

**University of Oslo
Department of Informatics**

**A Formal and
Executable Model
for Comparing
XML Documents**

Arild B. Torjusen

Cand. Scient. thesis

May 2003



Abstract

The main result in this thesis is the definition of a difference operator for XML documents and the implementation of a corresponding algorithm. The difference operator is similar to the diff Unix command, but it applies to structured documents rather than to plain files. The intended usage for the operator is to compare XML documents or webpages to identify updates and changes.

It is suggested to model a simplified version of XML as *terms*. A partial order relation is defined for such terms and a difference operator is formally defined with basis in the partial order. The operator must satisfy four properties, one of which is a minimality criterion. With these properties the result of the operator is a unique term. A difference algorithm is presented and is proved to be correct with regards to the definition of the operator. The proofs are by structural induction on the terms. The algorithm is implemented in Haskell as a function which ranges over a type representing the simplified version of XML.

The operator is initially defined for a simplified version of XML to discover the principal properties for such an operator. The results obtained for the simple model can be generalised to a model for proper XML. The thesis does not contain a formal definition for such an extended model but a Haskell function which ranges over a type representing XML without simplifications is implemented. Some examples of the application of the latter function to XHTML documents is presented.

Acknowledgements

Bjarte M. Østvold has been the primary supervisor for this Cand. Scient. project and I owe him great thanks for his devotion to the task. I have through the whole process always been able to rely on Bjarte for quick response and insightful comments to any question or problem which have occurred along the way. My second supervisor Olaf Owe gave valuable feedback in the final stage of the preparation of the thesis. Anders Moen has also contributed with important comments to the formal apparatus. My present employer, Integrate AS, deserves mention for allowing me great flexibility with regards to working hours.

My wife Line Bergem deserves a special thanks not only for her support and angelic patience in this period when I have devoted all my spare time to finish the thesis but also for her substantial suggestions for improvements. She is a sharp critic and my most valued discussion partner.

Contents

Preface	9
1 Background	13
1.1 Documents and structure	13
1.2 Document representation technology.	13
1.3 Document Processing	15
1.4 Data structures	16
1.5 Functional programming and Haskell	16
1.6 Two recent approaches to document processing using Haskell	19
1.6.1 Haskell and XML	20
1.6.2 Modelling HTML in Haskell	23
1.6.3 Comparison and relevance	25
1.7 A methodological remark	26
2 A term model for simplified XML	27
2.1 Terms and positions	29
2.1.1 Notation	29
2.1.2 Terms	30
2.1.3 Positions and subterms	32
2.2 Implementation of term operations in Haskell	34
2.3 Basic operations	35
2.4 Utility functions	37
3 A difference operator for terms	43
3.1 The cut relation for terms	45
3.2 Domain restrictions	52
3.3 The definition of term difference	54
4 The term difference algorithm	61
4.1 Correctness for the algorithm	64
4.2 The list cut relation	65
4.3 Auxiliary lemmas	67
4.4 Proof of left-cut	77
4.5 Proof of not right-cut	80
4.6 Proof of no junk	83
4.7 Proof of minimal left-cut	89
4.8 An alternative algorithm for term difference	92

5	Implementations	95
5.1	Implementation of utility functions	95
5.2	Implementation of term difference	96
5.3	An alternative implementation of term difference	97
6	Application to real-life web pages	99
6.1	The Haskell type <code>Element</code>	99
6.2	Utility functions	100
6.3	A difference operator for <code>Element</code>	104
6.4	Implementation of <code>Element</code> difference	106
6.5	Alternative implementation of <code>Element</code> difference	107
6.6	Examples	109
7	Conclusion	111
7.1	Results and assessments	111
7.2	Improvements and further work	112
A	Figures	115

Preface

The present thesis for the Cand. Scient. degree at the University of Oslo originally started out as a more broadly focused project with the working title “Structure-based operations on Web Documents”. The main goal for the project was to describe and implement structural operations on web documents. The practical motivation for the project was a belief that by studying how structured documents can be analysed we will be able to gain insights and also hopefully be able to implement tools which may be used for automatic analysis and processing of such documents. In this context “structured documents” should be understood to mean some form for SGML documents, more specifically either HTML or XML with the main focus on XML.

The process which eventually have lead to the present work was originally intended to proceed through four steps: The first step was to find criteria for comparing structured documents i.e. criteria for when two documents are equal or similar with regard to structure. When comparing documents there will be a continuum of cases ranging from complete structural isomorphy, to partial similarity. Two documents might for instance be partially similar to some degree in cases where parts of the document is isomorphic with parts of another document. The second step was to study and implement basic operations on structured documents. The basic operations correspond to the basic set-theoretic operations *Union*, *Intersection*, and *Difference*. Thirdly, I wanted to study how these basic operations could be combined into high-level operations. Finally I wanted to find out how and whether implementation of high-level operations or *filters* can be automated. E.g. how a filter that performs a specific manipulation/conversion on document instances can be automatically deduced by the system from the input documents together with sample documents which have the desired resulting properties. However these initial ambitions had to be adjusted in the course of the project, as the work implied turned out to be both more difficult and time-consuming than anticipated. The present work therefore is confined to the two first steps.

The list below contains some examples of things I hoped to be able to do with these filters or tools:

- To make summaries of several documents as e.g. a list of abstracts for a range of web pages.
- To merge two documents, so that only the differing parts are displayed in the result set.

- To extract only the elements which are different from a series of documents which are quite similar to each other.
- One could also imagine more complex operations, like taking two documents which each contains an HTML/XML table, and perform a join on specific fields in these tables and present the result as a new HTML/XML document for viewing in a browser. Other table-related operations could be to add lines or columns to tables in several HTML/XML documents.
- To filter away unwanted content from a web page. This include banner ads, “pop-up boxes” or any other elements we can single out as unwanted. Unwanted content in this context might also be content that repeated across a series of documents when we only wish to see the parts that are different from one document to the next.
- On a news-site (as a newspaper, news portal, message bulletin board, etc.), I would like to be able to “filter away” old material so that only the stories which are new, or changed since the previous visit to the page, are displayed.

The title “A Formal and Executable Model for Comparing XML Documents” is meant to indicate that this work includes both the framework of a formal model for XML, and executable implementations of operations on the basis of this framework. The main result in the thesis is the definition and implementation of a difference operator for XML documents. It can be used to achieve the last of the results listed above.

I start by defining a model for a simplified version of XML. Simplified XML is modelled as terms, and a term difference operator \setminus is formally defined for such terms on the basis of a more primitive term relation called cut (\preceq). In the course of the work with \setminus I also define a concept of intersection \cap which I however discuss in lesser detail. An algorithm (tmd) for computing term difference is then presented and is proved to be correct according to the definition of \setminus . The algorithm tmd is implemented as the Haskell function `tmDiff`. `tmDiff` ranges over the Haskell type `SElem` which corresponds to the term model for simplified XML.

The simplifications done in the model are of two kinds. The first kind regards the types of the objects in the model. In proper XML, the structure of a document is built up by two basic types, element and content. The element type consists of tags enclosing any number of objects of the content type. The content type may again be an element, but it may also be e.g.

a string, a comment etc. In the simplified model we only use one type a *simple-element* which consists of tags enclosing any number of objects of that same simple-element type. The second kind of simplification is that we do not allow these simple-elements to have attributes.¹

The reason that I have chosen to start from a simplified model, rather than modelling proper XML right away is that I want to focus on the principal properties of the tree-structure of XML documents without having to worry about the additional bookkeeping of attributes and comments etc. The results obtained in the simplified model are transferable to the general case and to illustrate how the results may be applied for non-simplified XML, I implement the Haskell function `e1Diff`. `e1Diff` is similar to `tmDiff` but ranges over the type `Element`. `Element` is a type which corresponds to XML without simplifications, and `e1Diff` can therefore be used to identify changes on XHTML pages.

Section 1 gives a background for the subject matter and discusses related work on markup language processing using Haskell. Section 2 presents the term model for simplified XML and introduce basic definitions. Section 3 introduces the cut-relation \preceq and some formal properties which are necessary for the definition of term difference \parallel are proved. Some restriction which must be enforced on the domain for \parallel are discussed and a definition of \parallel is given. The algorithm (`tmd`) for simplified XML and proofs of its correctness follows in Section 4. The Haskell implementation follows in the subsequent section. Section 6 discusses how the results so far may be applied to proper XML, presents the implementation of `e1Diff` and some examples of its application on real-life web pages.

¹The details are laid out in Section 2 below.

1 Background

1.1 Documents and structure

Almost any document has some form of structure, be it only that it has a header and a body and is divided into sections or paragraphs. A document could of course contain one long sequence of characters, but such a document would be unsuitable for conveying information (at least to human beings that is). Human cognition however is not the primary topic of this essay and I will not pursue this any further, my intention is just to state a point to which I believe you will agree, that structure plays an important role when we read and interpret texts, that we use the information inherent in the structure of the document as an aid to process the information in the document.

For a document intended for machine processing the demands to structure are much stricter. A human being who creatively can adapt the rules which governs his processing of a document as he proceeds processing it, can more easily cope with lack of strictness in the structure of a document than a rule-based computer can. So when we deal with documents meant for computing machines, a more strict method for keeping track of the document structure is called for. A very handy mean to bundle information about the structure of a document together with the document itself is by using some form for markup language.

1.2 Document representation technology.

The main standard for representing structured documents is SGML (Standard Generalized Markup Language, which is an ISO standard (ISO 8879:1986)). One main point with SGML is that it enables decomposition of a document into named elements. Thus one can refer to elements in a document. The elements may be arbitrarily named and are defined in a DTD (Document Type Definition) which may be a part of the SGML document or a separate document instance referred to by it. In any case, there will always correspond a DTD to an SGML document instance. The DTD specifies the elements which may be used in the given document and it specifies rules for how the different elements may relate to each other. For example, which and how many elements that are allowed inside another element.

Because SGML is very flexible, SGML document instances easily becomes quite complex and parsing and validating an SGML document according to a given DTD can be a very challenging and resource-demanding task.

1 BACKGROUND

One can describe SGML as a system for defining markup languages. A markup language defined in SGML is called an SGML *application* and is characterised by four different elements: an SGML declaration, a DTD, a specification which describes the semantics to be ascribed to the markup, and finally, document instances containing data and markup. HTML is such an SGML application. Or, more easily put, an HTML document is an SGML document which conforms to a specific DTD, namely the HTML DTD.

XML (Extensible Markup Language) is a subset of SGML, and could be described as a “simplification”, an “abbreviated version” or a “simple dialect” of SGML. Much of the motivation behind the definition of XML springs from a desire to be able to serve proper SGML on the web:

Its [the XML-specification] goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML. (W3C, 2000a, abstract)

From the perspective of simplification, one great advantage of XML over SGML is that an XML document does not necessarily need a DTD. There is a difference between a *well-formed* and a *valid* XML document. The XML standard quoted above specifies this: “Definition: A data object is an XML document if it is *well-formed*, as defined in this specification. A well-formed XML document may in addition be *valid* if it meets certain further constraints.” (W3C, 2000a, sec. 2)

The conditions for a document’s *well-formedness* are relatively straightforward: The document must contain one or more elements, there must be exactly one root element, and the elements should nest properly within each other. An *element* is a part of a document either delimited by a pair of start- and end-tags, thus: `<tag>content</tag>` or it may be an empty element tag thus: `</tag>` (which is equivalent with the empty element `<tag></tag>`). An element always has a name, and may have a list of name-value attributes. The content of an element may be any element, except the *root* element, or a sequence of characters.

For a document to be *valid* it must, in addition to be well-formed, also conform to a DTD: “Definition: An XML document is *valid* if it has an associated document type declaration and if the document complies with the constraints expressed in it.” (W3C, 2000a, sec. 2.8) The constraints that the DTD expresses describes a *grammar* for the XML document. It specifies the elements which may be used (a vocabulary) and it specifies which relations between elements are legal (the grammatical rules).

Even if serving pages to the web was an important motivation for the development of the XML standard, the relative simplicity and the generality of the standard have made it a popular choice in other kinds of applications where structuring of data is an issue. With a valid XML document instance you have a package of information, together with a structural description of it (the DTD). Hence XML can be used to encapsulate and describe information thus enabling portability. XML is now widely used as a file-storage format in applications such as word-processors, spread-sheet applications etc. XML is also used as a format for data exchange between applications. It is also a handy format for configuration files such as for instance deployment descriptors for web applications. XML is also very well suited for keeping catalogue information, as e.g. a list of bibliographic records, or any other kinds of catalogues of items. XML-catalogues can thus often be considered a sort of “light” databases, and is often utilised as such.

XHTML is a quite recent addition to the family of markup languages. It was constructed to overcome certain problems connected to the fact that HTML has some irregular properties. XHTML is a re-formulation of HTML as an XML application, i.e. XHTML is HTML reformulated to conform to the rules for XML.

1.3 Document Processing

The processing of a document may be considered a three tiered process. First the file/document must be parsed into some data structure, second operations may be performed on the data structure, thereby transforming it, or filtering it, according to given criteria. Finally the resulting data structure must be displayed or written to a file in a proper format. In other words the three steps involved are *parsing*, *transformation* and *pretty-printing*.

Parsing in general and parsing of XML in particular are very well-studied disciplines, especially within the field of functional programming: “The design of parsers has been a favourite application of functional programming for many years, . . .” (Bird, 1998, p. 373) I use HaXML, (available from <http://www.cs.york.ac.uk/fp/HaXml/>), which is a collection of utilities for using Haskell and XML together, including a parser and a pretty-printer for XML. HaXML is the toolkit developed and used in Wallace and Runciman (1999). In our case when working with web documents, the work of rendering the results of the document transformations will eventually be done by web-browsers. Either by transformation from XML to HTML, or by utilising browsers with XML capabilities. The main task for our project lies in the middle layer of this process, the transformation or manipulation

1 BACKGROUND

of data structures according to some given criteria. And although in the implementation of the transformation functions I do rely on the two other steps, the subjects of parsing and pretty-printing are not covered in this work. The interested reader may consult Bird (1998), Hughes (1995) and Hutton and Meijer (1998) for more information.

1.4 Data structures

To be able to work with structured documents, the document instances must be transformed into some kind of data structure for processing and manipulation. If we look at the criteria for well-formedness of XML above, it is easy to see that a well-formed XML document basically is a tree structure. So the data structures we will work on will most likely be some form of trees (or *terms* in a context free grammar) We might also work with graphs of some kind. There might be special issues related to structured hyper-text documents, as how to treat hyper-links, which might not be trivial.

So for instance the task of finding the *intersection* of two documents, i.e. finding the largest part of the documents which are similar, would amount to finding the largest common sub-tree in the two trees representing the documents.

1.5 Functional programming and Haskell

In contrast to a procedural or imperative language where the execution of a program consists of sequential execution of instructions or commands, the execution of a functional program consists of the evaluation of a function. To quote a textbook on the subject which states the main points quite concentrated:

Programming in a functional language consists of building definitions and using the computer to evaluate expressions. The primary role of the programmer is to construct a function to solve a given problem. This function [...] is described in a notation that obeys normal mathematical principles. The primary role of the computer is to act as an evaluator or calculator; its job is to evaluate the expression and print the result. (Bird, 1998, p. 1)

Haskell, which is the functional language I will use in this project, has some interesting features: It is strongly typed, which means that there are no exceptions to type rules and that all types are known at compile time, i.e. are statically bound. For any expression its type must be deducible from the types of the constituents of the expression. A major consequence of this is that it is possible at compilation time to check for type correctness

1.5 Functional programming and Haskell

and detect type errors before evaluation and thus dynamic type-errors are avoided. Both syntax checking and type checking takes place before the evaluation of a Haskell program begins.

The concept of polymorphic types is also central in Haskell. A polymorphic type is a type that contains type variables. (Bird, 1998, p. 23). Consider for instance a function `plus` with this type signature:

```
plus :: Integer -> Integer -> Integer
```

the type of `plus` is not polymorphic. But a type `fst` with the type signature

```
fst :: (α, β) -> α,
```

is polymorphic, α and β are place-holders for any type. So `fst` can be used to pick the first element of a pair of any types. The importance of type variables is apparent when we define functions on functions like e.g. `flip`:

```
flip      :: (β -> α -> γ) -> α -> β -> γ
flip f x y = f y x
```

Thus completely generic operations on functions can be defined and strong typing is preserved.

Sometimes we do not want to define types over arbitrary type variables, but we want to put some form for restriction on them. Then we would make use of type classes. Instead of giving `plus` the type signature above, we could specify the type as:

```
plus :: Num α => α -> α -> α
```

`Num α` specifies a so-called context which is a constraint on α in this type definition. The constraint states that α cannot be a totally arbitrary type but must be an instance of the type class `Num α`. Exploitation of type classes is integral to both of the two document processing strategies we describe below.

The underlying theoretical framework for Haskell is λ -calculus² Haskell really *is* λ -calculus with what is often characterised as “syntactic sugar” on top.³ A main achievement of λ -calculus is that it gives a fruitful notation for talking about functions. Thus the function $f(x) = 2x + 3$ can be expressed as $\lambda x.2x + 3$. The λ -operator “binds” the argument in the expression following the `.`, the main importance of this notation is, as Davie states, that “This

² λ -calculus was developed by Alonso Church (Church, 1941) on basis of his work on Frege in the thirties.

³See e.g. (Davie, 1992, p. 7 or p. 79)

1 BACKGROUND

allows us to put functions on a par with other kinds of object” (Davie, 1992, p. 7). With λ -calculus we have a formal framework that allows us, among other things, to describe very complex functions (which might be higher order), in a compact and relatively straightforward manner.

Evaluation of an expression in Haskell consists of reducing it to its normal form and print the result. Because Haskell in its foundation is λ -calculus, theoretical results in λ -calculus also applies to Haskell. This is relevant to the fact that Haskell uses lazy evaluation. Lazy evaluation means that a value is not computed until it is needed. This implementation of evaluation in Haskell as lazy evaluation is a consequence of the fact that Haskell does evaluation in so-called *normal order*. When evaluating an expression in normal order, one always starts with the outermost expressions. (In λ -calculus this corresponds to always starting with the leftmost redex in the λ -expression). This way of performing reduction guarantees that if a normal form exists the reduction sequence will end up with that form.⁴

In imperative languages we have variables which may be assigned values during the execution of a program. Evaluation of expressions in an imperative language may also have *side effects*. There is nothing strange with this, it’s just the way imperative languages work, as we all are familiar with. However programs written in an imperative language thus becomes order dependent, the meaning of the program depends on the order in which the statements are executed. This must be taken into consideration when reasoning about programs and consequently makes it very difficult to prove propositions about imperative programs. It is necessary to employ special schemes, as e.g. Hoare Logic (Hoare, 1969) to be able to take into consideration that values of variables change in the course of execution of the program. Proofs in Hoare logic tends to become lengthy and difficult both to understand and construct.

Haskell does not allow dynamic assignment of variables (this also rules out *side effects*). The execution of a Haskell program is, as before mentioned, the evaluation of a function, The order of evaluation does not matter. When evaluating e.g. an expression equivalent to the function $f(x, y, z)$ it does not matter which of x , y and z are evaluated first as we are guaranteed that evaluation of one of them cannot change the values of the others. In Haskell, the value of an expression depends only on the values of its well-formed sub-expressions. A language with this property is called referentially transparent, after Quine.⁵ Standard logic, being truth functional is based

⁴The second Church-Rosser theorem.

⁵“I call a mode of containment Φ referentially transparent if, whenever an occurrence

1.6 Two recent approaches to document processing using Haskell

on the same principle of transparency. (The truth value of a complex term depends only on the truth values of its constituents.)

So the advantage of the constraint that Haskell must be referentially transparent is that we can employ traditional logic to prove propositions about programs, and a proof of a complex functional expression can be broken down to proofs of its constituents. We will also be able to use well-known proof techniques, most important of which will be mathematical induction.

Several of the above-mentioned aspects of Haskell contributes to making it a suitable tool/language to use in our project:

- The close proximity between Haskell and mathematical language, and Haskell's referential transparency, which enables the use of standard proof strategies for proving properties about programs.
- Higher order functions can be used to build complex filters from basic functions.
- The type checking capabilities of Haskell is useful when checking validity of DTDs and documents.
- Haskell has proved very able to deal with tree structures. And there exists toolkits for parsing and pretty-printing XML.

1.6 Two recent approaches to document processing using Haskell

To give some background I will look into two recent projects in this field. The first one, Wallace and Runciman (1999) deals with representing XML using Haskell. Two main approaches to the task are pursued and compared: First a generic one, whereby an internal data structure suitable for representing any well-formed XML (I.e. regardless of DTDs) document is defined. The other approach investigated takes into consideration the type of the XML document at hand, and consists of defining internal Haskell types corresponding to the DTD for the document.

The other project Thiemann (2000) deals with the modelling of HTML using Haskell.

of a singular term t is purely referential in a term or sentence $\Psi(t)$ it is purely referential also in the containing term or sentence $\Phi(\Psi(t))$." (Quine, 1960, p. 144) where the concept of 'purely referential' is specified thus: "Here we have a criterion for what may be called *purely referential position*: the position must be subject to the *substitutivity of identity*" (ibid p. 142). Where substitutivity of identity means that co-referential terms may be substituted *salva veritate*

1 BACKGROUND

1.6.1 Haskell and XML

Wallace and Runciman (1999) discuss two approaches to XML processing using Haskell. The first approach is generic, i.e. it describes a procedure for processing any well-formed XML document regardless of its DTD. As mentioned above, any well-formed XML document exhibits a tree-structure, and it is this common property of XML documents which enables this generic approach. The task is to

(1) Define an internal data structure that represents content of *any* XML document, independent of all DTDs. (Wallace and Runciman, 1999, p. 1)

The second approach takes as its point of departure the DTD for a specific document type and uses this to derive Haskell data types:

(2) Given the DTD for some XML documents of interest, systematically *derive* definitions for internal Haskell data types too represent them. These definitions are closely based on the specific DTD. (ibid.)

The two main advantages of the first approach are *genericity*, i.e. the ability to deal with any XML instance without having to know anything about the DTD, and *function-level scripting*.

An example of such a generic application is pattern-matching in XML-documents.⁶ That an application is generic does not necessarily mean that it does not take the document type into consideration at all. A typical situation would be that an application will have *some* knowledge or presuppositions about the document type it is supposed to process. (A very simple such presupposition could for instance be that an element with a specific name (tag) is present in the document.)

Function level scripting means that we can hide the underlying details about the data structures from the programmer. This is accomplished by defining high-level combinators:⁷

All details of data structure manipulation can be hidden in a library of high-level combinators. In effect, combinators serve as an extensible domain-specific language. (Wallace and Runciman, 1999, p. 2)

There are two basic groups of functions involved: *content filters*, and *combinators*. A content filter is the basic type for functions processing the

⁶Wallace and Runciman have implemented an interpreter for a regular XML query language called *Xtract*

⁷Combinators is here taken to mean functions operating on functions or “higher order functions”

1.6 Two recent approaches to document processing using Haskell

internal representation of an XML document, and combinators are used to combine filters into more complex filters. There is also a top-level wrapper function:

```
processXmlwith :: CFilter -> IO()
```

We assume a top-level wrapper function, which gets command-line arguments, parses an XML file into the `Content` type, applies a filter, and pretty-prints the output document. (Wallace and Runciman, 1999, p. 3)

The basic types in the representation of an XML document are `Element` and `Content` which together forms a multi-branch tree structure.

```
data Element = Elem Name [Attribute] [Content]
data Content = CElem Element
              | CText String
```

The content filter processes elements of the type `Content`. The type `CFilter` takes some content as input and returns (a possibly empty list of content):

```
type CFilter = Content -> [Content]
```

This same basic type is used both to filter the input by selecting parts of it and to construct output. Consequently there are two groups of filters. *Predicate and selection* filters⁸ are for instance filters for checking whether a `Content` is an `Element` or a `String` or filters that return a list of the children of a `Content`. Some examples of *Construction* filters are: `literal s` which makes a content element containing the `String s` and the `mkElem t fs` which builds a content element with the tag `t` and a list of filters `fs`.

As mentioned the combinators is used to combine filters, the most important is the `'o'` combinator. It plugs two filters together such that the left filter is applied to the result of the right filter. An example is:

```
txt 'o' children 'o' tag 'title'
```

which returns the plain-text children of the current element provided the current element has the `title` tag. Other combinators are designed to append the result of two filters, or to enable one filter to function as a guard on another filter, etc. There is also an if-then-else combinator:

```
tag 'foo' ? none :> keep
```

⁸Predicate filters are selection filters that always gives as result either the empty list or a list with one element.

1 BACKGROUND

This filter would return nothing if the current element has the tag name `foo` or else return the element itself. There are also recursive combinators, of which `foldXml f` is worth mentioning. It applies the filter `f` to every level of the tree.

There is also a special `LabelFilter` type which together with a special combinator `'oo'` enables a rich variety of possibilities for attaching labels to items. Thus it is possible to make numbered lists etc.

The other approach is based upon the existence of a DTD for the XML document to be processed. The DTD defines a grammar for the document and document validation becomes a check that the structure of the document conforms to this grammar. XML document validation is an integral part of any XML application but Wallace and Runciman launch an idea to take the validation a bit further, namely the idea of *valid document processing*:

A valid processing script is one which produces a valid document as output given a valid document as input. We achieve this by demonstrating a correspondence between the DTD of the document and the definition of a set of algebraic types in Haskell, and the consequent correspondence between the document's content and a structured Haskell (Wallace and Runciman, 1999, p. 8)

The authors define a set of rules for the translation from a DTD to Haskell type declaration. The process may on the surface seem quite straightforward but a few tricky issues is involved.⁹

In the second approach the primary advantage is the validity checking capabilities. By drawing on the type-checking inherent in the Haskell language, validity of the processed XML documents is ensured. But in addition it also gives a more direct programming style than when you use combinators. You can for example use pattern matching, which may produce programs that are simpler and easier to understand. One disadvantage is a high initial costs, because one has to process the DTD before doing any transformations, a task that might be difficult. Another disadvantage (which maybe just as well could be considered to be an open possibility) is that the DTD grammar is small and restrictive compared to the type capabilities of Haskell, so we cannot take advantage of its full power.

Pros and Cons of the two schemes According to Wallace and Runciman their combinator scheme for document processing has several advantages compared to the current mainstream solution (which they take to be “new domain specific languages for expressing and scripting transformations”):

⁹I will not go into this, but cf. (Wallace and Runciman, 1999, p. 8)

1.6 Two recent approaches to document processing using Haskell

- A combinatorial library can easily be extended and patterns occurring in a combinator program can be isolated and re-used.
- The fact that the scripts themselves are Haskell programs implies that we can take advantage of the full power of the programming language and are not restricted by a maybe limited scripting language.
- Related to the above point is the fact that when using Haskell as a scripting language we have available things we need such as interpreters and compilers.
- When using Haskell we have a safe foundation for proving semantic properties about our scripts.

The disadvantages are that the syntax of Haskell combinator script will be quite unlike XML-syntax, and might be difficult to work with for people unfamiliar with Haskell.

Further work Wallace and Runciman point out three main directions for further work. One is to work further with the generality of the combinators, extending their functionality so they can handle e.g. deletion and also even transformation and editing of elements. An aspect highly relevant to my project, namely enabling handling of multiple input instances at once, is also put forth as an interesting extension. It is also mentioned that the properties of some of the recursive combinators are not fully investigated.

A second main direction for further work is to enhance efficiency through normalisation, by making processing trees more space efficient, and by using knowledge of the DTD for the document being processed to make the processing more efficient.

1.6.2 Modelling HTML in Haskell

Thiemann (2000) exemplifies another approach to using Haskell for processing markup documents, the focus here is on HTML. He defines HTML as an *embedded domain specific language* in Haskell and has defined a library of combinators to create and modify HTML elements. He makes extensive use of the mechanism of type classes to enforce well-formedness of HTML and he accentuates this as the main contribution of the paper.

Due to our use of type classes, the Haskell type checker guarantees the well-formedness of the generated HTML to a large degree. This exploitation of the type system is in our view the main contribution of the paper. (Thiemann, 2000, p. 2)

1 BACKGROUND

All HTML elements are modelled by its own data type and each element type is member of the type class `ELEMENT` which has methods to get and set contents and attributes and to render the element as a (HTML) string.¹⁰

The process of creating HTML elements uses constructors of the form `make_[tagname]` (as e.g. `make_body` which makes an element of type `BODY`) together with an `add` function to add contents.¹¹ to the element, for instance:

```
make_head 'add' (make_title 'add' 'Hello World')
```

This creates an HTML element of the type `HEAD` which contains an element of type `TITLE` which contains the text “Hello World”, in HTML this would be:

```
<HEAD><TITLE>Hello World</TITLE></HEAD>
```

This is not unlike the strategy of Wallace and Runciman (1999) where elements can be constructed by using the `mkElem` construction filter which takes as its argument a tag name and a list of `Content` elements, the element above would be constructed like this:

```
mkElem "HEAD" [mkElem "TITLE" [(literal "Hello World")]]
```

The main difference is that in the generic approach of Wallace and Runciman (1999) there is no checking on whether an element of a certain kind is allowed to be a part of another element.¹² Thieman on the other hand uses type classes to check which elements are allowed inside other elements. The method `add s t`, is specified in the type class `AddTo s t` (`s` is the type of the container and `t` is the type of the content) and has a default implementation in the type class. This default implementation is not overridden:

```
class (ELEMENT s, ELEMENT t) => AddTo s t where
  add :: s -> t -> s
  add element subelement =
    set_contents element (ELEMENT_Cons subelement (contents element))
```

To state that an element is a legal subelement of another, one only have to declare an instance of `AddTo` e.g.:

¹⁰Attributes are modelled along the same line, with a type for each attribute and a type class `ATTRIBUTE`

¹¹The contents of an element is a list of elements specified by the special type `ELEMENT_list` which is an heterogeneous list of values of type class `ELEMENT`:

```
data ELEMENT_List =
  forall e . ELEMENT e => ELEMENT_cons e ELEMENT_List | ELEMENT_Nil
```

(Thiemann, 2000, p. 7)

¹²This is quite natural as they are dealing with unrestricted XML.

1.6 Two recent approaches to document processing using Haskell

```
instance AddTo HEAD TITLE
```

So if an element¹³ is a non-legal sub element of some element, i.e. the element and its sub element is not specified as an instance of `AddTo`, Haskell's type checking will raise an error. The instance declarations needed to enforce well-formedness for a specific document type corresponds closely to the DTD for the document.

Thieman also shows how it is possible with the aid of these basic type constructors to define patterns or templates for parameterised documents. You can for instance define a new type, say `myDoc` which takes a title and a text and generates something of type `HTML`. I will not go into details on how this is done, but it is easy to imagine that this can be a useful feature, and it is a feature that Wallace and Runciman (1999) lacks.¹⁴

1.6.3 Comparison and relevance

Thieman points out, that the generic approach of Wallace and Runciman lacks validity checking of the elements (which no-one denies, it is just the price paid for genericity.) He further states that the type-based encoding of Wallace and Runciman on the other hand lacks flexibility for treatment of parameterised documents, and he suggests that a combination of the his and theirs work might prove fruitful: “[the combinator library of Wallace and Runciman (1999)] could be ported to our representation, thus reaping the benefits of their combinators while enjoying strongly typed XML at the same time” Thiemann (2000).

I will not enter into a discussion about these statements. Maybe the Wallace and Runciman (1999) approach could be accommodated to handle parameterised documents better, and maybe a combination of the work done in the two papers will give interesting results. In any case, what I can say in view of the subjects I intend to investigate in my thesis is that Thieman does not seem to focus particularly on the subject of filtering or processing of documents. Though he mentions that “Haskell can . . . be used as a meta-language to . . . extract information from the documents.” his main focus seems to be on *generation* of HTML.

I will not enter into a discussion about these statements. Maybe the Wallace and Runciman (1999) approach could be accommodated to handle parameterised documents better, and maybe a combination of the work done

¹³Attributes are treated analogously, with a type class `AddAttr e a` which specifies the method `attr e a` (`e` being the type of the element and `a` the type of the attribute).

¹⁴At least in the same straightforward manner as in Thieman's paper.

1 BACKGROUND

in the two papers will give interesting results. In any case, what I can say in view of the subjects I intend to investigate in my thesis is that Thieman does not seem to focus particularly on the subject of filtering or processing of documents. Though he mentions that “Haskell can . . . be used as a meta-language to . . . extract information from the documents.” his main focus seems to be on *generation* of HTML.

In this work I pursue more generic approach along the same lines as the approach in Wallace and Runciman (1999).

1.7 A methodological remark

The approach to the subject matter in this thesis follows two main directions. On the one hand it is necessary with a thorough understanding of the basic operations. The “basic operations” in this context are typically operations on terms. These basic functions should be modelled in a precise language and their formal properties studied. On the other hand it is important to know how to apply the functions in real-life cases. This points out two possible directions for the investigation. Either one could start with basic operations on elementary types, study their formal properties, and then step-by-step introduce more complex types and operations/functions on these. Thus a library of operations for processing of content can be compiled. This is a “bottom-up” approach to implementation.

The other approach starts out by looking at “real-life” web-pages and consider what would be interesting to be able to do with them (in terms of processing and transformations). From that point of departure one can proceed to implement the functions and tools that is thought to be necessary to solve the computational challenges at hand.

I believe that the interplay between these two directions will turn out to be germinal. My intention has therefore been to proceed in both directions. In short, on the one hand a thorough understanding of the basic operations is necessary to be able to put them to proper use on real-life cases, and on the other hand knowledge of the challenges involved in processing real-life web pages, is necessary for knowing where to direct the focus in the analysis and construction of basic operators and tools. Although this is in fact the way I have approached the subject matter it is mainly the “bottom up” approach which is documented in the present text.

2 A term model for simplified XML

I take as the point of departure the XML 1.0 specification published by the W3C (World Wide Web Consortium)¹⁵. A well-formed XML document contains one or more elements, there should be exactly one *root* element,¹⁶ and the elements should nest properly within each other. In other words: for a textual object to be a well-formed XML document it should match the production labelled `document`. The production for `document` (§2.1) is:

```
document ::= prolog element Misc
```

My focus is on the `element` so I just comment shortly on the two others:

```
prolog ::= XMLDecl? Misc* (doctypeddecl Misc*)?
```

The `prolog` may contain an `XMLDecl` which specifies which version of XML is being used. It may also contain a `doctypeddecl` which either contains or points to the document type definition, or DTD. `Misc` is either `Comment`, `S` or `PI`. The first is a comment, it starts with `<!--` and ends with `-->`. `S` is whitespace (I.e. space characters, carriage returns, line feeds, or tabs). `PI` is a processing instruction which is passed through to the application interpreting the XML document. a `PI` item starts with `<?` and ends with `?>`.

`element` is defined (§3) as:

```
element ::= EmptyElemTag
          | STag content ETag
```

which means that an element either consists of some `content` delimited by a pair of start- and end-tags, thus: `<tag>content</tag>` or it may be an empty element tag, which will look like this: `</tag>` (which is equivalent to `<tag></tag>`). The definitions for the three different kinds of tag elements (§3.1) are:

```
EmptyElemTag ::= '<' Name (S Attribute)* S? '/>'
STag         ::= '<' Name (S Attribute)* S? '>'
ETag        ::= '</' Name S? '>
```

An `Attribute` is a name/value pair. An important additional well-formedness constraint is: “Element Type Match: The Name in an element’s end-tag must match the element type in the start-tag.” (§3) The `content` is defined (§3.1) as:

¹⁵I.e. W3C (2000a) The section references below are to this document if not otherwise stated.

¹⁶“[Definition: There is exactly one element, called the *root*, or document element, no part of which appears in the content of any other element.]” (§2.1)

2 A TERM MODEL FOR SIMPLIFIED XML

```
content ::= CharData?(element|Reference|CDSect|PI|Comment)CharData?)*
```

Of the items which make up a `content` we will focus on the `element`. I will just comment shortly on the other parts: `CharData` is any text which is not *markup*¹⁷. A `Reference` is either an `EntityRef` i.e. a reference to a previously defined entity¹⁸ or a `CharRef`.¹⁹ A `CDSect` (i.e. CDATA section) “... may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognised as markup.” (§2.7). A CDATA section starts with `<![CDATA[` and ends with `]]>`.

To be able to get started with the analysis of XML documents, and to be able to formulate basic properties and functions, I start by doing some simplifications.

The first simplification is to disallow the occurrence of a list of name/value pairs in the `S`Tag and `EmptyElemTag`. In other words we do not allow attributes in our elements. So the simplified versions of the definitions for the first two kinds of tag elements are:

```
SEmptyElemTag ::= '<' Name S? '/>'
SSTag          ::= '<' Name S? '>'20
```

Our definition of the simplified element, `Selem` becomes:

```
Selem ::= SEmptyElemTag
        | SSTag Scontent ETag
```

Where `Scontent` is the simplified definition of `content`. We get at this definition by disregarding anything except `element` from the `content` production. The definition is:

```
Scontent ::= (Selem)*
```

An `Selem` will either be an empty element like this: `<foo/>` or an element with some content `<bar>Scontent</bar>`, where the `Scontent` is any number of `Selems`. What remains here is a rather simplistic grammar, and it

¹⁷“Text consists of intermingled character data and markup. [Definition: Markup takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, processing instructions, XML declarations, text declarations, and any white space that is at the top level of the document entity (that is, outside the document element and not inside any other markup).]” (§2.4)

¹⁸In the prologue of the document one might define entities like this: `<!ENTITY myname "Arild">`. If so defined one may refer to this entity further down in the document using: `&myname`

¹⁹A `CharRef` is a string on the form `&#n` or `&#xn` where `n` is a number representing a character code point in ISO/IEC 10646 in respectively decimal or hexadecimal notation.

²⁰We keep the well-formedness constraint “Element Type Match” mentioned above.

2.1 Terms and positions

might be objected, too simplistic to be interesting. However this is a methodological trick which we do to be able to focus on the basic properties of the tree-structure of an XML instance without having to worry about the additional bookkeeping of comments and attributes etc. When we have the basic functions for this simple grammar in place, it is not too difficult to modify these to handle standard XML.

The Haskell type we use to represent this simplified element is `SElem` defined like this:

```
data SElem = SElem Name [SElem]
type Name = String
```

The `Name` of an `SElem` corresponds to the tag of the XML element. Any string may be a tag.²¹ The well-formedness constraint for XML stating that there should be exactly one *root* element could also have been enforced by adding a type `Doc` for documents:

```
data Doc = RootElem [SElem]
```

However we will keep the definition as unrestricted as possible, and will not enforce a distinction between a `Doc` and an `SElem`. The data structure for `SElem` is a multi-branching tree.²²

Example 2.1. *The XML-instance:*

`<f><g><a/></g><h><c/><d/></h></f>` corresponds to the Haskell object:

```
SElem "f" [SElem "g" [SElem "a" [],SElem "b" []],
          SElem "h" [SElem "c" [],SElem "d" []]
        ]
```

2.1 Terms and positions

2.1.1 Notation

I use meta-variables to stand for elements in the object-language. This means that for example the meta variable x stands for an arbitrary (object) variable. The naming conventions are as follows:

²¹For modelling HTML `Name` would be restricted to only allow for a set of predefined tags

²²A data type for representing multi-branching trees (called “rose trees”) is discussed in (Bird, 1998, p. 195). Note that our `SElem` type is a `String` rose tree.

```
data Rose a = Node a [Rose a]
```

2 A TERM MODEL FOR SIMPLIFIED XML

- x, y, z for variables.
- a, b, c, d, e for function symbols of zero arity (constants).
- f, g, h for function symbols of any arity.
- s, t, u, v, w for terms
- k, l, r for lists of terms. An over-lined term symbol may be used to denote a list of terms. (For example: $f(\bar{t})$ instead of $f(t_1, \dots, t_n)$).
- p, q for positions and P for lists of positions.

Teletype is used for concrete terms. As e.g. `a()`, or `html()`.

- A calligraphic font (e.g. \mathcal{F}, \mathcal{G}) is used for functions over terms.

$(x \neq y)$ is shorthand for $\neg(x = y)$. ‘ \Rightarrow ’ and ‘ \Leftrightarrow ’ denote (meta-)implications. For all kinds of symbols subscripts and superscripts may be used.

2.1.2 Terms

A *ranked alphabet* is a finite nonempty set of symbols each of which has a unique nonnegative *arity* (or *rank*). Let Σ be a ranked alphabet. For each $n \geq 0$ the set of n -ary symbols in the ranked alphabet Σ is denoted by Σ_n . ε denotes the empty string.

Definition 2.2 (Terms, T_Σ). *The set T_Σ of Σ -terms is defined inductively as the smallest set of strings such that:*

$$\Sigma_0 \cup \{\varepsilon\} \subseteq T_\Sigma \tag{i}$$

$$\text{If } f \in \Sigma_n \wedge 1 \leq n \wedge \bigwedge_{i=1}^n t_i \in T_\Sigma \text{ then } f(t_1, \dots, t_n) \in T_\Sigma \tag{ii}$$

If $c \in \Sigma_0$, c is a function symbol of zero arity or a *constant*, for simplicity I write just c for a term $c()$ and also T for T_Σ .²³

It will often be the case that in an XML document, the same tag occurs at several places, but with a different number of children at each occurrence. In an HTML document this is the typical situation. (E.g. for TD, P or A tags). So when modelling (X/HT)ML documents as terms, it might seem that the

²³The definitions of terms and term properties are based on the following sources: Gécseg and Steinby (1997), Comon et al. (1998) and Ölveczky (2003)

constraint that all function symbols have a unique and finite arity might cause some problems. A typical HTML BODY element could for example be modelled by the term:

```
body(p(a, a, a),
p,
table(tr(td, td(a)), tr(td, td)),
table(tr(td(a, td, td))))
```

Here the symbols `p`, `tr` and `table` have varying arities, which is in conflict with the assumption above.

However one can ensure that any function symbol has a unique arity by indexing any function symbol with the arity of the function, and adopt the convention that this index is a part of the function symbol. Thus the symbol ' f^i ' denotes the function symbol f with the arity-index i . The term above, properly indexed, would become:

```
body4(p3(a0, a0, a0),
p0,
table2(tr2(td0, td1(a0)), tr2(td0, td0)),
table1(tr1(td3(a0, td0, td0))))
```

However, to simplify the notation and arguments in this thesis I have chosen to skip over this difficulty. I will for example allow the terms $f(g(x, y))$ and $f(g(x, y, z))$ to occur in one and the same context even if to be strict this should not be allowed. The correct thing would be to write $f^1(g^2(x, y))$ and $f^1(g^3(x, y, z))$. The reason that I allow this sloppiness in the treatment of function symbols is that I have confidence that it can be overcome by an amount of extra bookkeeping of indexes.

There is one occasion though, where I have to be more precise with regards to function symbol arities. And that is when considering the concepts of “tag-equality” and “function-equality”. I say that two terms are tag-equal if both terms’ outermost function symbols matches without regards to the index on the function symbol. The notion of “function-equality” is slightly stricter: Two terms are function-equal if their outermost function symbols are equal (with the indexes, or the arity of the function symbol, taken into consideration). I will give formal definitions below.

In the term $f(t_1, \dots, t_n)$, the expression (t_1, \dots, t_n) denotes a *list*, i.e. the ordered set of the elements t_1, \dots, t_n . The concept of list is defined by the empty list ‘()’ and a “cons” operator ‘:’.

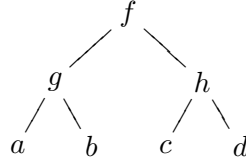
Definition 2.3 (List). *The concept of list is defined by the following two statements where $()$ denotes the empty list*

- (i) $()$ is a list
- (ii) if l is a list then $t:l$ is a list for any term t

The notation (t_1, \dots, t_n) is used as a more convenient way of denoting the list $t_1:(t_2:(\dots t_n:(\dots)))$. Note particularly that the list $t:()$ can be written: (t) .

Example 2.4. *A special case of a member of T is the term \mathbf{a} . It models the XML-instance $\langle \mathbf{a} \rangle$.*

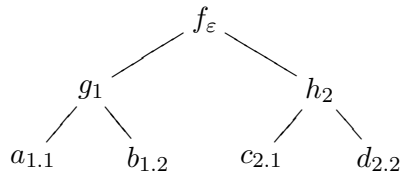
Example 2.5. *The XML-instance in Example 2.1 is modelled by the term $\mathbf{f}(g(\mathbf{a}, \mathbf{b}), h(\mathbf{c}, \mathbf{d}))$. The term can be depicted as a tree like this:*



2.1.3 Positions and subterms

Definition 2.6 (Position). *A position is either the empty string, denoted by ε , or a finite string consisting of positive integers separated by dots. ε is right identity for positions (i.e. $p.\varepsilon = p$). ε denotes the root position in a term. $i.p$ denotes the position p in the i -th subterm of a term.*

Example 2.7. *The term above, with positions:*



The set of positions in a term t denoted $\mathcal{P}(t)$ can be described intuitively as follows. For a term t , $\varepsilon \in \mathcal{P}(t)$, and if p is the position of f in $t = g(\dots f(t_1, \dots, t_n) \dots)$ then $p.1, \dots, p.n$ are the positions of t_1, \dots, t_n in t .

Definition 2.8 (Positions in term ($\mathcal{P}(t)$)). $\mathcal{P}(t)$ is defined inductively:

$$\mathcal{P}(\varepsilon) = \emptyset \quad (3)$$

$$\mathcal{P}(c) = \{\varepsilon\} \quad (4)$$

$$\mathcal{P}(f(t_1, \dots, t_n)) = \{\varepsilon\} \bigcup_{i=1}^n \{i.p \mid p \in \mathcal{P}(t_i)\} \quad (5)$$

The following observation, which follows directly from the definition is used in a later proof.

Observation 2.9. For any term $t = f(t_1, \dots, t_n)$ where $\mathcal{H}(t) > 1$, if $p \in \mathcal{P}(t)$, then if $p \neq \varepsilon$, p is $i.p'$, where $1 \leq i \leq n$ and $p' \in \mathcal{P}(t_i)$

With the definition of *positions in term* in place, we can define the concept of a *subterm of a term at position p* , denoted by $t|_p$.

Definition 2.10 (Subterm). If $p \in \mathcal{P}(t)$ then the term $t|_p$ is called a *subterm of t* and is defined inductively as follows:

$$t|_\varepsilon = t$$

$$f(t_1, \dots, t_n)|_{i.p} = t_i|_p$$

If $p \neq \varepsilon$, then $t|_p$ is called a *proper subterm* of t .

Example 2.11.

$$f(g(a, b), h(c, d))|_1 = g(a, b)$$

$$f(g(a, b), h(c, d))|_{2.1} = c$$

A position is a string and we can define the *length of a position*:

Definition 2.12 (Length of position).

$$\text{len}(\varepsilon) = 0$$

$$\text{len}(i.p) = 1 + \text{len}(p).$$

From the length of a position it can be decided how “far down” or at what “depth” in a term a certain subterm is located. For terms t, u and position $p \in \mathcal{P}(t)$: u is a subterm of t at depth n iff:

$$\exists p \ p \in \mathcal{P}(t) \wedge t|_p = u \wedge \text{len}(p) = n.$$

We introduce a more compact notation to denote the set of positions at a certain depth in a term:

Definition 2.13 (Positions at depth ($\mathcal{P}_n(t)$)).

$$\mathcal{P}_n(t) = \{p \mid p \in \mathcal{P}(t) \wedge \text{len}(p) = n\} \quad (6)$$

With this notation we can define:

Definition 2.14 (Subterm at depth). For terms t, u and position p :

$$u \text{ is a subterm of } t \text{ at depth } n \text{ iff } \exists p \ p \in \mathcal{P}_n(t) \wedge t|_p = u. \quad (7)$$

The expression “immediate subterm” denotes a subterm at depth 1 of a term. The expression “children” might also be used instead of “immediate subterms”.

Example 2.15.

$$g(a, b) \text{ and } h(c, d) \text{ are the immediate subterms of } f(g(a, b), h(c, d))$$

Subterm replacement is an important concept since it plays a crucial role in the definition of the cut relation and thereby also in the definition of term difference.

Definition 2.16 (Subterm replacement). If t and u are terms, and p is a position in t , then $t[u]_p$ is the term t , where $t|_p$ is replaced by u . $t[u]_p$ is defined inductively as follows:

$$\begin{aligned} t[u]_\varepsilon &= u \\ f(t_1, \dots, t_i, \dots, t_n)[u]_{i.p} &= f(t_1, \dots, t_i[u]_p, \dots, t_n) \end{aligned}$$

Example 2.17.

$$\begin{aligned} f(g(a, b), h(c, d))[x]_\varepsilon &= x \\ f(g(a, b), h(c, d))[x]_1 &= f(x, h(c, d)) \\ f(g(a, b), h(c, d))[x]_{2.1} &= f(g(a, b), h(x, d)) \end{aligned}$$

2.2 Implementation of term operations in Haskell

Just to illustrate the correspondence between the terms defined above and the Haskell type `SElem`, some of the definitions are implemented.

Positions are implemented as a list of `Ints`

```
type Position = [Int]
```

2.3 Basic operations

The function `positions` is an implementation of Definition 2.8 above. It takes an `SElem` as argument and returns a list of the positions in the term in lexicographic order.

```
positions :: SElem -> [Position]
positions (SElem t []) = [[]]
positions (SElem t elms) =
  [] : (concat (map g (zip [1..] elms)))
  where g (n,elm) = [n : pos | pos <- positions elm]
```

The function `subterm` implements Definition 2.10. It takes an `SElem` (`tm`) and a `Position` (`pos`) as arguments and returns the subterm of `tm` at the given position. Note that the function is undefined if not $p \in \mathcal{P}(t)$

```
subterm :: SElem -> Position -> SElem
subterm tm pos
  | (pos `elem` (positions tm)) = subterm' tm pos
  | otherwise = undefined
where
  subterm' tm [] = tm
  subterm' (SElem t elems)(p:ps) = subterm' (elems!!(p-1)) ps
```

Finally the function `subtSubst` implements Definition 2.16 above.

```
subtSubst :: SElem -> SElem -> Position -> SElem
subtSubst t u [] = u
subtSubst (SElem t elms) u (p:ps) =
  (SElem t (mapAtp (p-1) (tn -> subtSubst tn u ps) elms))
```

`mapAtp p f xs` applies the function `f` to the `p`-th member in the list `xs` (indexed from 0), and leaves the rest of the list as it is:²⁴

```
mapAtp 0 f (x:xs) = ((f x) : xs )
mapAtp p f [] = []
mapAtp p f (x:xs) = (x:(mapAtp (p-1) f xs))
```

2.3 Basic operations

The goal for this work is to define some operations which in turn can be implemented as functions, useful for analysis and processing of XML-documents. The intention is that it should be possible to use the operations to compare two XML-instances D_{t_1} and D_{t_2} to see how they differ from each other. An intended application for such an operation might be to compare the XML-document downloaded from an URL at time t_1 with the document downloaded from the same URL at a later time t_2 .

²⁴For example: `mapAtp 2 (*10) [1,2,3,4]` evaluates to `[1,2,30,4]`.

2 A TERM MODEL FOR SIMPLIFIED XML

If we are able to do such a comparison we can from D_{t_2} either remove the content we already have seen in D_{t_1} or we could keep it and highlight the content that is new since the last viewing. If nothing has changed we could also issue a message to the viewer about that, so he wouldn't have to bother to view the page a second time.

So we seek to define two operators \parallel and \parallel^m . $D_{t_2} \parallel D_{t_1}$ should yield the new parts in D_{t_2} compared to D_{t_1} . \parallel^m is similar to \parallel but instead of only yielding the difference of the two documents it will produce a new term D'_{t_2} with the new parts highlighted or marked in some way.

The operator sought for corresponds to the \setminus (set-minus) operator for sets. A first naïve approach might therefore be to define operations on terms corresponding to the set-theoretic basic operations: \cup, \cap and \setminus . The term-variants of these operators are marked with the index t : The definition of \cup_t would be:

$$\begin{aligned} f(\dots) \cup_t g(\dots) &= \perp \\ f(v_1, \dots, v_n) \cup_t f(u_1, \dots, u_m) &= f((v_1, \dots, v_n) \cup_l (u_1, \dots, u_m)). \end{aligned}$$

As (v_1, \dots, v_n) and (u_1, \dots, u_m) are lists, \cup_l denotes a union operator for lists, comparable to the union operation for sets.²⁵ The same applies to \cap_l and \setminus_l . The definition of \cap_t would be:

$$\begin{aligned} f(\dots) \cap_t g(\dots) &= \perp \\ f(v_1, \dots, v_n) \cap_t f(u_1, \dots, u_m) &= f((v_1, \dots, v_n) \cap_l (u_1, \dots, u_m)), \end{aligned}$$

and \setminus_t :

$$\begin{aligned} f(\dots) \setminus_t g(\dots) &= \perp \\ f(v_1, \dots, v_n) \setminus_t f(u_1, \dots, u_m) &= f((v_1, \dots, v_n) \setminus_l (u_1, \dots, u_m)). \end{aligned}$$

However, whereas these definitions yields the intended result for terms of height up to two²⁶ it is not obvious how they can be extended to cover the general case of terms of arbitrary size.

One solution to this might have been to define suitable functions à la

²⁵I will not go further into how such an operator should be defined (cf. e.g. (Jones et al., 1999b, sec. 7) for examples).

²⁶I.e. terms which have children, but no "grandchildren". (A term with height 1 is a term with no children.) See below for a formal definition of height of terms.

map and *fold*²⁷ and somehow combine the basic operations with these higher level functions to get the desired result. I do not believe that this is the right way to proceed at this point, as it is difficult to see directly how this can be done. I will therefore start from my intuitions regarding the functionality for \parallel (and \parallel^m). By analysing these intuitions I hope to arrive at the definition of a more complex function, capable of dealing with the general case.

2.4 Utility functions

To be able to define a term difference operator, I need to define some basic concepts and operators for terms. First a concept of *height* for a term. A term with no children has height 1 and generally the height of a term is 1+ the height of the highest of its children.

Definition 2.18 (Height of a term).

$$\begin{aligned}\mathcal{H}(\varepsilon) &= 0 \\ \mathcal{H}(f(t_1, \dots, t_n)) &= 1 + \max\{\mathcal{H}[t_i]\}_{i=1}^n\end{aligned}$$

Example 2.19.

$$\begin{aligned}\mathcal{H}(a()) &= 1 \\ \mathcal{H}(f(g(a, b), h(c, d))) &= 3\end{aligned}$$

The size of a term t denoted by $|t|$ is the number of symbols in the term and is inductively defined by:

²⁷The functions *map* and *foldr/foldl* are defined as follows for lists (the notation is function application without parentheses):

$$\begin{aligned}\text{map } \mathcal{F} (x_1, \dots, x_n) &= (\mathcal{F} x_1, \dots, \mathcal{F} x_n) \\ \text{foldr } \mathcal{F} e (x_1, x_2 \dots, x_n) &= \mathcal{F} x_1 (\mathcal{F} x_2 (\dots (\mathcal{F} x_n e))) \\ \text{foldl } \mathcal{F} e (x_1, x_2 \dots, x_n) &= \mathcal{F} (\dots (\mathcal{F} (\mathcal{F} e x_1)x_2) \dots)x_n\end{aligned}$$

These would of course look different for terms. For example a map function for terms: map_t , which can be used to apply a function \mathcal{F} to all nodes in a term could be defined as:

$$\text{map}_t \mathcal{F} g(v_1, \dots, v_n) = \mathcal{F} g(\text{map}_t \mathcal{F} t_1, \dots, \text{map}_t \mathcal{F} t_n)$$

implemented in Haskell this corresponds to:

$$\text{mapTree } f \text{ (SElem } t \text{ elms)} = f \text{ (SElem } t \text{ (map (mapTree } f) \text{ elms))}$$

It is somewhat more difficult to get a clear intuition about what the *fold* operations on terms would look like.

Definition 2.20 (Size of a term).

$$|\varepsilon| = 0$$

$$|f(t_1, \dots, t_n)| = 1 + \sum_{i=1}^n |t_i|$$

Example 2.21.

$$|a| = 1$$

$$|f(g(a, b), h(c, d))| = 7$$

We should be able to compare terms in different ways and therefore I define a range of comparison operators of different strength. We start with *isomorphy*. Two terms s and t are isomorphic if their structure match, i.e. they must have the same number of children, and for each child of s it must again be isomorphic to the corresponding child of t :

Definition 2.22 (Term isomorphism, \sim). For terms $t_1, \dots, t_n, u_1, \dots, u_n$ and function symbols f, g :

$$\bigwedge_{i=1}^n (t_i \sim u_i) \Rightarrow f(t_1, \dots, t_n) \sim g(u_1, \dots, u_n)$$

We also need to be able to decide whether two terms are “tag-equal”:

Definition 2.23 (Root tag equality, \approx). For terms $t_1, \dots, t_n, u_1, \dots, u_m$ and function symbol f :

$$f(t_1, \dots, t_n) \approx f(u_1, \dots, u_m) \quad (m, n \geq 0)$$

For easier notation when comparing terms for (root) tag equality, let the function \mathcal{S} be a function which picks out the outermost function symbol in a term.

Definition 2.24 (Function symbol of term, $\mathcal{S}(t)$).

$$\mathcal{S}(f(t_1, \dots, t_n)) = f$$

The relation to \approx is obvious:

$$s \approx t \text{ iff } \mathcal{S}(s) = \mathcal{S}(t). \quad (8)$$

The notion of function equality is stricter than root tag equality. For two terms to be function equal, not only the function symbol, but also the arities of the two functions—or in other words—the number of children must be equal.

Definition 2.25 (Function equality, $=_{\mathcal{F}}$). For terms $t_1, \dots, t_n, u_1, \dots, u_n$ and function symbol f :

$$f(t_1, \dots, t_n) =_{\mathcal{F}} f(u_1, \dots, u_n) \quad (n \geq 0)$$

Example 2.26.

$$f(a, g(b)) \sim f'(c, h(d))$$

$$f(b, c, d) \approx f(e)$$

$$f(b, c, d) \neq_{\mathcal{F}} f(e)$$

$$f(b_1, c_1, d_1) \approx f(b_2, c_2, d_2)$$

$$f(b_1, c_1, d_1) =_{\mathcal{F}} f(b_2, c_2, d_2)$$

Equality proper for terms is defined inductively as follows:

Definition 2.27 (Term equality, $=$).

$$\varepsilon = \varepsilon$$

$$\bigwedge_{i=1}^n (t_i = u_i) \Rightarrow f(t_1, \dots, t_n) = f(u_1, \dots, u_n) \quad (n \geq 0)$$

It is useful also to define a concept of *relative equality*. The function we use to check for relative equality differs from the function for equality proper in that the former is parametrised with an integer as parameter. This integer n specifies how thorough two terms should be checked against each other, or in other words, how far down the tree we descend to look for differences. In the comparison of the two terms s and t :

- if the parameter $n = 0$ we do not care about the children of the two terms. We only check whether the tags on s and t match. This corresponds to the concept of root tag equality defined above.
- if $n = 1$ the tags on s and t must match, and the arity of the function symbols of the two terms must match (i.e. they must both have the same number of children), and the tags of the children of the two terms must match pairwise.
- if $n = k$ the tags on s and t must match, the number of children of the two terms must match, and the children again must pairwise match each other to the grade of $k - 1$.

2 A TERM MODEL FOR SIMPLIFIED XML

Note that the second case above ($n = 1$) is just a special case of the more general third case, and therefore superfluous. Relative equality is defined inductively as follows:

Definition 2.28 (Relative term equality $=_{R(k)}$).

$$s \approx t \Rightarrow s =_{R(0)} t \quad (9)$$

$$\bigwedge_{i=1}^n (t_i =_{R(k-1)} u_i) \Rightarrow f(t_1, \dots, t_n) =_{R(k)} f(u_1, \dots, u_n) \quad (10)$$

Example 2.29.

$$\begin{aligned} f(b, c, d) &=_{R(0)} f(e) \\ f(b, c, d) &\neq_{R(1)} f(e) \\ f(g(a), h(b)) &=_{R(1)} f(g(d), h(e)) \\ f(g(a), h(b)) &\neq_{R(2)} f(g(d), h(e)) \\ f(g(a), h(b)) &=_{R(2)} f(g(a), h(b)) \end{aligned}$$

Equality proper for terms $s, t \neq \varepsilon$ can be stated in terms of relative equality thus:

Corollary 2.30.

$$s = t \Leftrightarrow s =_{R(\max(\mathcal{H}[s], \mathcal{H}[t]))} t$$

Proof. The proof is by induction on the height of terms. □

It can be noted that:

Corollary 2.31.

$$s =_{R(k+1)} t \Rightarrow s =_{R(k)} t \quad (k \geq 0) \quad (11)$$

Proof. The proof is induction on the degree of equality k . □

We can also introduce the concept of a parametrised difference of terms:

Definition 2.32 (n -difference). *We say that two terms t and u “ n -differs” or that they are “ n -different” iff n is the smallest integer such that $\neg(t =_{R(n)} u)$*

Observe that if two terms t, u are not equal they will be n -different for some n , ($0 \leq n \leq (\max(\mathcal{H}[t], \mathcal{H}[u]))$)

Observation 2.33.

$$\forall t, u \ (t \neq u) \Rightarrow \exists n_{(0 \leq n \leq (\max(\mathcal{H}[t], \mathcal{H}[u])))} \neg(t =_{R(n)} u)$$

Proof. The proof is by induction on the height of the terms. □

A relation R on a set X is a subset of $X \times X$, and is an equivalence relation if and only if R has the following properties (xRy is shorthand for $(x, y) \in R$ and is read “ x is related to y ”):

1. Reflexivity: $\forall a \in X \ aRa$
2. Symmetry: $\forall a, b \in X \ aRb \rightarrow bRa$
3. Transitivity: $\forall a, b, c \in X \ aRb \wedge bRc \rightarrow aRc$

Conjecture 2.34. *The operators defined above (\sim , \approx , $=_{\mathcal{F}}$, $=$ and $=_{R(k)}$) are all equivalence relations and for each of the operators, equivalence classes might be defined which will form a partition of the domain of terms.*

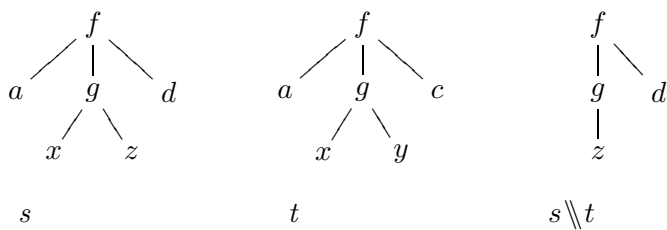
2 A TERM MODEL FOR SIMPLIFIED XML

3 A difference operator for terms

With the basic properties and definitions on terms in place, we can now turn the attention to the main operator \parallel (term difference). As mentioned in Section 2.3 \parallel is intended to be a subtraction operator for terms. $s \parallel t$ is meant to be the term which is the difference between s and t .

Loosely speaking one might say that in $s \parallel t$ we only want to keep the immediate subterms, or children, from s that are new, or which have descendants that are new compared to t . In addition to this, we only keep those *parts* of a term which are new. An example might elucidate what is meant. The terms are displayed as trees for illustrative purposes.²⁸

Example 3.1.



In outline the following should be the result of applying \parallel on the two terms s and t : if the new term s differs from the old one t , $s \parallel t$ is built from s . I.e. we use the function symbol of s , and build a new list of immediate subterms for this function symbol. As immediate subterms in $s \parallel t$ we include

1. those immediate subterms from s which are not immediate subterms in t . (As for d in the example.)
2. if an immediate subterm in s *partially matches* some immediate subterm in t we include in the list of immediate subterms in $s \parallel t$ the result of applying \parallel on those two subterms. (As for $b(x, z)$ in the example.)

²⁸As mentioned above, I am sloppy with the function symbols and write

$$f(a, g(x, z), d) \parallel f(a, g(x, y), d) = f(g(z), d)$$

where it strictly speaking should have been

$$f^3(a^0, g^2(x, z), d^0) \parallel f^3(a^0, g^2(x, y), d^0) = f^2(g^1(z), d^0).$$

3 A DIFFERENCE OPERATOR FOR TERMS

So what does it mean that two terms match each other partially? It means that they are equal in some way. Several concepts of equality were defined above, so there are several candidates for the job of deciding whether two terms match each other partially. It must be decided how strict the match condition should be. The equality notions from the previous chapter can be ordered in terms of strictness like this:²⁹

$$\{=_{R(0)}, \approx\} \supseteq \{=_{R(1)}, =_{\mathcal{F}}\} \supseteq =_{R(2)} \supseteq \dots$$

Which of them to choose might depend on the domain for which we intend to compare terms. It makes a difference whether we intend to use it to compare two web-pages in HTML, i.e. documents of which we might know little about the structure, and for which the concept of function arity does not make much sense, or if we intend to use it to compare two XML documents which has a strict DTD.

For now \approx (i.e. $=_{R(0)}$) is chosen as the condition for *partial match*. This is for two reasons.

First, remember that some of the motivation behind defining a operator for term difference for simple terms is a belief that \parallel might be used as a starting point to define a corresponding operator for more general comparison of terms. Whereas the domain for \parallel is terms modelling simplified XML as presented above, the domain for a general version of the difference operator could be terms modelling proper XML (or HTML). I have the intuition that the $=_{R(0)}$ notion is the one that will turn out to be the best starting point in the general case.

Second, it can from a methodological perspective be considered reasonable to choose this condition as it is the least strict one. It is better to start with a weak condition and strengthen it later than to cut of potentially interesting results prematurely.

A formal definition of \parallel will be given below. But first a few words about what is to follow. \parallel can be defined by the more primitive concept *cut*, which will be introduced and formally defined in the next section. In Section 3.2 some domain restrictions for \parallel are introduced and considered. The definition of \parallel follows in Section 3. An algorithm for computing $s \parallel t$ is presented in Section 4, and in sections 4.4–4.7 I will prove that the algorithm is correct with regards to the definition of \parallel .

²⁹In other words, the partitions which can be generated by the operators gets more fine grained from the left to the right.

3.1 The cut relation for terms

Intuitively speaking, a *cut* of a term is what one gets if a subterm is removed from a term. *Cut* is a more primitive concept than term difference and will be used in the definition of \parallel . I will therefore first formalise and study this concept. The *cut*-relation can be thought of as a “less than” relation for terms. And I will use the symbol \prec to denote this relation. In this section I start by defining *proper cut* \prec which is an irreflexive relation on terms. Next I define *cut* \preceq which is a generalisation of \prec . If we think of \prec as an operation, \preceq can be explained by saying that $t \preceq u$ if t can be made from u by applying a cut-operation 0– n times. Finally I prove that \preceq is a partial order relation for terms. This is a result which is vital for subsequent argument.

Definition 3.2 (Partial order). *A relation R on a set X is a subset of $X \times X$, and is a partial order if it has the following properties:³⁰*

1. *Reflexivity:* $\forall a \in X \ aRa$
2. *Transitivity:* $\forall a, b, c \in X \ aRb \wedge bRc \rightarrow aRc$
3. *Antisymmetry:* $\forall a, b \in X \ aRb \wedge bRa \rightarrow a = b$

The observations and theorems which follows in this section are integral to the definition of \parallel and to the discussion of the properties of \parallel , which I return to in the following sections.

To remove a subterm from a term is the same as substituting that subterm with the empty string ε and *cut* can therefore be defined as follows:

Definition 3.3 (Proper cut \prec). *For terms t and u*

$$t \prec u \text{ if } \exists p \ p \in \mathcal{P}(u) \wedge t = u[\varepsilon]_p \quad (12)$$

Lemma 3.4. *\prec is irreflexive. For any term s :*

$$\neg(s \prec s) \quad (13)$$

which is equivalent to

$$\neg \exists p \ p \in \mathcal{P}(s) \wedge s = s[\varepsilon]_p. \quad (14)$$

³⁰ xRy is shorthand for $(x, y) \in R$ and is read “ x is related to y ”

3 A DIFFERENCE OPERATOR FOR TERMS

Proof. The proof is by induction on the height of s .

Basis: $\mathcal{H}(s) \leq 1$. For $\mathcal{H}(s) = 0$, $s = \varepsilon$ and $\neg(\varepsilon \prec \varepsilon)$ follows directly from Definition 3.3 since $\mathcal{P}(\varepsilon) = \emptyset$. If $\mathcal{H}(s) = 1$, by Definition 2.8 $\mathcal{P}(s) = \{\varepsilon\}$:

$$\begin{aligned}
\neg(s \prec s) \wedge \mathcal{H}(s) = 1 & \Leftrightarrow \text{Def. 3.3, Def. 2.8} & \text{(i)} \\
\neg \exists p \ p \in \{\varepsilon\} \wedge s = s[\varepsilon]_p & \Leftrightarrow PC^{31} & \text{(ii)} \\
\forall p \ p \in \{\varepsilon\} \rightarrow s \neq s[\varepsilon]_p & \Leftrightarrow PC & \text{(iii)} \\
s \neq s[\varepsilon]_\varepsilon & \Leftrightarrow \text{Def. 2.16} & \text{(iv)} \\
s \neq \varepsilon & \Leftrightarrow \text{True} & \text{(v)}
\end{aligned}$$

Induction step: Assume as the induction hypothesis that for any terms s such that $1 \leq \mathcal{H}(s) \leq n$:

$$\forall p \in \mathcal{P}(s) \ s \neq s[\varepsilon]_p \quad \text{(i)}$$

Consider an arbitrary term $t = f(t_1, \dots, t_n)$, where $\mathcal{H}(t) = n+1$, and assume for contradiction that $t \prec t$.

$$\begin{aligned}
t \prec t \wedge \mathcal{H}(t) = n+1 & \quad \text{Assumption} \quad \Rightarrow & \text{(ii)} \\
\exists p \ p \in \mathcal{P}(t) \wedge t = t[\varepsilon]_p & \quad \text{Def 3.3, ii} & \text{(iii)}
\end{aligned}$$

If $p = \varepsilon$ then $t = t[\varepsilon]_p = \varepsilon$ which contradicts the assumption (ii) The other case to consider is if $p \neq \varepsilon$.

$$\exists p' \ p' \in \mathcal{P}(t) \wedge t = t[\varepsilon]_{i,p'} \quad \text{Obs 2.9} \quad \Rightarrow \quad \text{(iv)}$$

But then for this p' :

$$\begin{aligned}
f(t_1, \dots, t_i, \dots, t_n) & = \\
f(t_1, \dots, t_i[\varepsilon]_{p'}, \dots, t_n) & \quad \text{Obs. 2.9, Def. 2.16} \Rightarrow & \text{(v)} \\
t_i = t_i[\varepsilon]_{p'} \wedge \mathcal{H}(t_i) \leq n & \quad PC, v \Rightarrow & \text{(vi)} \\
\perp & \quad vi, i \Rightarrow & \text{(vii)} \\
\neg(t \prec t) & \quad PC, ii-vii & \text{(viii)}
\end{aligned}$$

and irreflexivity of \prec follows by general induction. \square

The following lemma will be used in the proof that \preceq is antisymmetric:

³¹“PC” in a derivation indicates usage of standard first-order predicate calculus.

3.1 The cut relation for terms

Lemma 3.5. *For any term s :*

$$\neg \exists p \ p \in \mathcal{P}(s) \wedge s \prec s[\varepsilon]_p \quad (15)$$

Proof. As above, the proof is by induction on the height of s .

Basis: $\mathcal{H}(s) \leq 1$. For $\mathcal{H}(s) = 0$, $s = \varepsilon$ and (15) follows directly from Definition 3.3 since $\mathcal{P}(\varepsilon) = \emptyset$. If $\mathcal{H}(s) = 1$, by Definition 2.8 $\mathcal{P}(s) = \{\varepsilon\}$:

$$\begin{aligned} \mathcal{H}(s) = 1 \rightarrow \neg \exists p \ p \in \mathcal{P}(s) \wedge s \prec s[\varepsilon]_p &\Leftrightarrow \text{Def. 3.3, Def. 2.8} & \text{(i)} \\ \neg \exists p \ p \in \{\varepsilon\} \wedge s \prec s[\varepsilon]_p &\Leftrightarrow PC & \text{(ii)} \\ \forall p \ p \in \{\varepsilon\} \rightarrow \neg(s \prec s[\varepsilon]_p) &\Leftrightarrow PC, \text{Def. 3.3, Def. 2.8} & \text{(iii)} \\ \neg(s \prec s[\varepsilon]_\varepsilon) \wedge \mathcal{H}(s) = 1 &\Leftrightarrow \text{Def. 2.16} & \text{(iv)} \\ \neg(s \prec \varepsilon) \wedge \mathcal{H}(s) = 1 &\Leftrightarrow \text{Def. 3.3, Def. 2.8} & \text{(v)} \\ \neg \exists p \ p \in \emptyset \wedge s = \varepsilon[\varepsilon]_\varepsilon &\Leftrightarrow \text{True} & \text{(vi)} \end{aligned}$$

Induction step: Assume as the induction hypothesis that (15) holds for any terms s such that $1 \leq \mathcal{H}(s) \leq n$. Consider an arbitrary term $t = f(t_1, \dots, t_n)$, where $\mathcal{H}(t) = n + 1$, and assume for contradiction that $\exists p \ p \in \mathcal{P}(t) \wedge t \prec t[\varepsilon]_p$.

$$\exists p \ p \in \mathcal{P}(t) \wedge t \prec t[\varepsilon]_p \quad \text{Assumption} \quad (i)$$

If $p = \varepsilon$ then $t \prec t[\varepsilon]_p$ is equivalent to $t \prec \varepsilon$ which is a contradiction. If $p \neq \varepsilon$ then:

$$\exists p' \ p' \in \mathcal{P}(t) \wedge t \prec t[\varepsilon]_{i.p'} \quad \text{Obs 2.9} \quad \Rightarrow \quad (ii)$$

But then for this p' :

$$t \prec f(t_1, \dots, t_i[\varepsilon]_{p'}, \dots, t_n) \quad \text{Obs. 2.9, Def. 2.16} \quad \Rightarrow \quad (iii)$$

$$\exists q \in \mathcal{P}(t) \ t = f(t_1, \dots, t_i[\varepsilon]_{p'}, \dots, t_n)[\varepsilon]_q \quad (iv)$$

- If $q = \varepsilon$ then $t = f(t_1, \dots, t_i[\varepsilon]_{p'}, \dots, t_n)[\varepsilon]_\varepsilon = \varepsilon$ which is a contradiction.
- If $q = j.q'$ where $j \neq i$ then $t = f(t_1, \dots, t_i[\varepsilon]_{p'}, \dots, t_n)[\varepsilon]_q$ implies $t_i = t_i[\varepsilon]_{p'}$, which contradicts (14) since $p' \in \mathcal{P}(t_i)$.
- If $q = i.q'$, then $t = f(t_1, \dots, t_i[\varepsilon]_{p'}, \dots, t_n)[\varepsilon]_q$ implies $t_i = (t_i[\varepsilon]_{p'})[\varepsilon]_q$, which implies that $t_i \prec (t_i[\varepsilon]_{p'})$ and that contradicts the induction hypothesis(15).

3 A DIFFERENCE OPERATOR FOR TERMS

Any choice for q leads to a contradiction and it may be concluded that:

$$\forall p \in \mathcal{P}(t) \quad \neg(t \prec t[\varepsilon]_p) \quad (\text{v})$$

and (15) follows by general induction. \square

The generalisation of proper cut is the relation \preceq which is the reflexive transitive closure of \prec :

Definition 3.6 (Cut \preceq). For terms t, u

$$t \preceq u \quad \text{if} \quad t = u \quad \text{or} \quad (16)$$

$$\exists t' \quad t \prec t' \wedge t' \preceq u \quad (17)$$

Example 3.7.

$$f(g(a, b), h(d)) \prec f(g(a, b), h(d, e)) \quad (18)$$

$$f(g(a, b)) \prec f(g(a, b), h(d, e)) \quad (19)$$

$$f(g(b)) \preceq f(g(a, b), h(d, e)) \quad (20)$$

Observation 3.8. Note that for any terms $f(\bar{t})$ and $g(\bar{u})$, if $f \neq g$ then $g(\bar{u}) \not\preceq f(\bar{t})$. This holds because one cannot make $f(\bar{t})$ equal to $g(\bar{u})$ by removing subterms from $f(\bar{t})$. The two terms will always differ in their outermost function symbol. Hence for function symbols f, g and lists of terms \bar{t}, \bar{u} :

$$f \neq g \Rightarrow f(\bar{t}) \not\preceq g(\bar{u}) \quad (21)$$

which is equivalent to

$$f(\bar{t}) \preceq g(\bar{u}) \Rightarrow f = g. \quad (22)$$

Corollary 3.9 (Distance between terms). If $s \preceq t$ then there exists a least n and terms $u_0 \dots u_n$ such that

$$s = u_0 \wedge u_0 \prec u_1 \wedge \dots \wedge u_{n-1} \prec u_n \wedge u_n = t \quad (23)$$

and this n is the distance between s and t , written $\mathcal{D}(s \preceq t)$.

Example 3.10.

$$\text{If } s = t \text{ then } \mathcal{D}(s \preceq t) = 0$$

$$\text{If } s \prec t \text{ then } \mathcal{D}(s \preceq t) = 1$$

$$\text{If } s \neq u \wedge u \neq t \wedge s \preceq u \wedge u \preceq t \text{ then } \mathcal{D}(s \preceq t) = 2$$

3.1 The cut relation for terms

For a term v , the set of all terms u such that $u \preceq v$ is called the cut-set of v .

Definition 3.11 (Cut-set $\mathcal{C}(v)$). For terms u, v :

$$\mathcal{C}(v) = \{u \mid u \preceq v\} \quad (24)$$

Example 3.12.

$$\mathcal{C}(f(g(a, b))) = \{\varepsilon, f, f(g), f(g(a)), f(g(b)), f(g(a, b))\}$$

I intend to prove that for any term, \preceq is a partial order on the set $\mathcal{C}(v)$. Each of the three properties reflexivity, transitivity and antisymmetry is proved separately:

Theorem 3.13. \preceq is reflexive.

Proof. For reflexivity it must be proved that:

$$\forall t \in \mathcal{C}(v) \quad t \preceq t. \quad (25)$$

This follows directly from Definition 3.6. \square

Theorem 3.14. \preceq is transitive.

Proof. The proposition that must be proved for transitivity is:

$$\forall s, t, u \in \mathcal{C}(v) \quad s \preceq t \wedge t \preceq u \rightarrow s \preceq u., \quad (\text{Transitivity})$$

If $s = t$ then obviously transitivity holds. ($s = t \wedge t \preceq u \rightarrow s \preceq u$). If $s \neq t$ transitivity is proved by induction on $\mathcal{D}(s \preceq t)$.

Basis: For the basis, $\mathcal{D}(s \preceq t) = 1$, i.e. $s \prec t$.

$$\begin{array}{lll} s \prec t \wedge t \preceq u & \text{Assumption} & \Rightarrow \quad \text{(i)} \\ \exists t \quad s \prec t \wedge t \preceq u & \text{PC} & \Rightarrow \quad \text{(ii)} \\ s \preceq u & \text{Definition 3.6} & \Rightarrow \quad \text{(iii)} \\ s \prec t \wedge t \preceq u \rightarrow s \preceq u & \text{PC, i-iii} & \Rightarrow \quad \text{(iv)} \end{array}$$

Induction step: Assume as the induction hypothesis that for any terms s, t, u :

$$s \preceq t \wedge t \preceq u \rightarrow s \preceq u \quad 0 < \mathcal{D}(s \preceq t) \leq n \quad \text{(v)}$$

3 A DIFFERENCE OPERATOR FOR TERMS

Given the induction hypothesis, consider:

$$\begin{array}{llll}
s \preceq t \wedge t \preceq u & \mathcal{D}(s \preceq t) = n + 1 & \text{Assumption} & \Rightarrow \quad \text{(vi)} \\
\exists s' \ s \prec s' \wedge s' \preceq t \wedge t \preceq u & \mathcal{D}(s' \preceq t) = n & \text{Def 3.6} & \Rightarrow \quad \text{(vii)} \\
\exists s' \ s \prec s' \wedge s' \preceq u & & v, vii & \Rightarrow \quad \text{(viii)} \\
s \preceq u & & viii, \text{Def 3.6} & \Rightarrow \quad \text{(ix)} \\
s \preceq t \wedge t \preceq u \rightarrow s \preceq u & & PC, vi-ix & \Rightarrow \quad \text{(x)}
\end{array}$$

and transitivity for \preceq follows by general induction. \square

Theorem 3.15. \preceq is antisymmetric.

Proof. The proposition that must be proved for antisymmetry is

$$\forall s, t \in \mathcal{C}(v) \ s \preceq t \wedge t \preceq s \rightarrow s = t. \quad (26)$$

If $s = t$, obviously antisymmetry holds. If $s \neq t$ the proof is by induction on $\mathcal{D}(s \preceq t)$ and includes a second proof by induction on $\mathcal{D}(t \preceq s)$.

Basis: For the basis, $\mathcal{D}(s \preceq t) = 1$, i.e. $s \prec t$ and it must be proved that:

$$s \prec t \wedge t \preceq s \rightarrow s = t. \quad (i)$$

The proof of (i) is by induction on $\mathcal{D}(t \preceq s)$:

Basis: For the basis, $\mathcal{D}(t \preceq s) = 1$, i.e. $t \prec s$, the argument is *ad absurdum*:

$$\begin{array}{llll}
\neg(s \prec t \wedge t \prec s \rightarrow s = t) & \text{Assumption} & \Rightarrow & \text{(ii)} \\
s \prec t \wedge t \prec s \wedge s \neq t & PC, ii & \Rightarrow & \text{(iii)} \\
t \prec s \rightarrow \exists q \in \mathcal{P}(s) \ t = s[\varepsilon]_q & \text{Def 3.3} & \Rightarrow & \text{(iv)} \\
\exists q \in \mathcal{P}(s) \ t = s[\varepsilon]_q & iii, iv & \Rightarrow & \text{(v)} \\
\exists q \in \mathcal{P}(s) \ s \prec s[\varepsilon]_q & PC, iii, v & \Rightarrow & \text{(vi)} \\
\forall p \in \mathcal{P}(s) \ s \prec s[\varepsilon]_p & PC, Lem 3.5 & \Rightarrow & \text{(vii)} \\
\perp & PC, vi, vii & \Rightarrow & \text{(viii)} \\
s \prec t \wedge t \prec s \rightarrow s = t & PC, ii-viii & \Rightarrow & \text{(ix)}
\end{array}$$

Induction step: This is also a *reductio* argument. The induction hypothesis is that:

$$s \prec t \wedge t \preceq s \rightarrow s = t, \quad \text{if } 0 < \mathcal{D}(t \preceq s) \leq n. \quad (i)$$

3.1 The cut relation for terms

Assume for contradiction that:

$$\begin{array}{llll}
\neg(s \prec t \wedge t \preceq s \rightarrow s = t) \wedge & & & \\
\mathcal{D}(t \preceq s) = n + 1 & \text{Assumption} & \Rightarrow & \text{(ii)} \\
s \prec t \wedge t \preceq s \wedge s \neq t & \text{PC,ii} & \Rightarrow & \text{(iii)} \\
\exists u \ s \prec t \wedge t \prec u \wedge u \preceq s \wedge s \neq t \wedge & & & \\
\mathcal{D}(u \preceq s) = n & \text{Def 3.3} & \Rightarrow & \text{(iv)} \\
\exists u t \prec u \wedge u \preceq s & & & \\
\mathcal{D}(u \preceq s) = n \wedge & \text{PC,iv} & \Rightarrow & \text{(v)} \\
t = s & \text{IH(i),v} & \Rightarrow & \text{(vi)} \\
t = s \wedge s \neq t & \text{v,vi} & \Rightarrow & \text{(vii)} \\
\perp & \text{vi} & \Rightarrow & \text{(viii)} \\
s \prec t \wedge t \preceq s \rightarrow s = t & \text{PC,ii-viii} & \Rightarrow & \text{(ix)}
\end{array}$$

And $s \prec t \wedge t \preceq s \rightarrow s = t$ follows by general induction. This concludes the basis step in the inductive proof of (26). *Induction step:* This is also a *reductio* argument. The induction hypothesis is that:

$$s \preceq t \wedge t \preceq s \rightarrow s = t \quad \text{if } 0 < \mathcal{D}(s \preceq t) \leq n \quad \text{(i)}$$

$$\begin{array}{llll}
\neg(s \preceq t \wedge t \preceq s \rightarrow s = t) \quad \mathcal{D}(s \preceq t) = n + 1 & \text{Assumption} & \Rightarrow & \text{(ii)} \\
s \preceq t \wedge t \preceq s \wedge s \neq t \quad \mathcal{D}(s \preceq t) = n + 1 & \text{PC,ii} & \Rightarrow & \text{(iii)} \\
\exists u \ s \preceq u \wedge u \prec t \quad (\mathcal{D}(s \preceq u) = n) & \text{Cor. 3.9,iii} & \Rightarrow & \text{(iv)} \\
\exists v \ t \prec v \wedge v \preceq s & \text{Cor. 3.9,iii} & \Rightarrow & \text{(v)} \\
\exists u, v \ s \preceq u \wedge u \preceq v \wedge v \preceq s \quad (\mathcal{D}(s \preceq u) = n) & \text{PC,Def. 3.3,iv,v} & \Rightarrow & \text{(vi)} \\
\exists u, v \ s \preceq u \wedge u \preceq s \quad (\mathcal{D}(s \preceq u) = n) & \text{Trans. of } \preceq, \text{vi} & \Rightarrow & \text{(vii)} \\
\exists u, v \ s = u & \text{PC,IH(i),vii} & \Rightarrow & \text{(viii)} \\
\exists u, v \ s \preceq s \wedge s \prec t & \text{PC,iv,viii} & \Rightarrow & \text{(ix)} \\
\exists u, v \ s \preceq s \wedge s \prec t \wedge t \prec v \wedge v \preceq s & \text{PC,ix,v} & \Rightarrow & \text{(x)} \\
\exists u, v \ s \prec t \wedge t \preceq s & \text{Def. 3.3,x} & \Rightarrow & \text{(xi)} \\
s \prec t \wedge t \preceq s & \text{PC,xi} & \Rightarrow & \text{(xii)} \\
s = t & \text{Basis (i),xii} & \Rightarrow & \text{(xiii)} \\
\exists u \ s = u & \text{PC,viii} & \Rightarrow & \text{(xiv)} \\
s = t \wedge s \neq t & \text{iii,xiii} & \Rightarrow & \text{(xv)} \\
s \preceq t \wedge t \preceq s \rightarrow s = t & \text{PC,ii,xv} & \Rightarrow & \text{(xvi)}
\end{array}$$

3 A DIFFERENCE OPERATOR FOR TERMS

Antisymmetry for \preceq follows by general induction. \square

Corollary 3.16. *\preceq is reflexive, transitive and antisymmetric, hence for any term v , the relation \preceq is a partial order on the set $\mathcal{C}(v)$.*

3.2 Domain restrictions

Before \parallel can be defined, some domain restrictions need to be considered. From a methodological point of view it may be noted that we have to make these restrictions “to get started” so to say. As we progress I will discuss and evaluate these restrictions.

The first restriction is that the two arguments to \parallel must be isomorphic.

Restriction 1.

$$s \not\sim t \Rightarrow s \parallel t = \perp \tag{R1}$$

The initial motivation for this restriction is that while it is relatively straight forward to understand what a difference operation on two isomorphic terms means, it is much harder to get to grips with what it would mean to perform a difference operation on two terms with quite diverging structures.

The second restriction is that $s \parallel t$ is only defined for terms such that for any subterm in the term there is no repetition of function symbols in the list of immediate subterms.³²

Restriction 2. *$s \parallel t$ is only defined for terms s and t such that for any subterm $f(t_1, \dots, t_n)$ in s and t the following holds:*

$$\forall i, j \quad \mathcal{S}(t_i) = \mathcal{S}(t_j) \Rightarrow i = j \quad 1 \leq i, j \leq n \tag{R2}$$

Example 3.17.

$$\begin{aligned} f(g(z), b, w) \parallel f(g(x), g(y), c) &= \perp \\ f(a, g(h(x), h(y))) \parallel f(b, c) &= \perp \end{aligned}$$

The reason for this second restriction is a little more complex, and hints at some interesting problems related to comparison of terms. I will consider them briefly.

³²In other words: At no place in a term may the same function symbol occur more than once in the same list of subterms.

There are two related but different problems concerning comparison of terms which contain several occurrences of the same function symbol. The first might be called “the problem of choice”: In the computation of $\text{tmd}(s, t)$, where $t = f(t_1, \dots, t_n)$ and $s = g(u_1, \dots, u_m)$, the subterms of s will be compared to the subterms of t , as sketched above (43). The first difficulty occurs if an u_i matches several t ’s. It must then be decided whether the result for the i -th subterm should be $\text{tmd}(u_i, t_j)$ or $\text{tmd}(u_i, t_k)$, as in this example: What is the result of

$$f(g(z), b, w) \parallel f(g(x), g(y), c)?$$

Depending on which of $g(x)$ or $g(y)$ is chosen as the subterm to compare $g(z)$ against, the result might be either: $f(g(z \parallel x), b, w)$ or $f(g(z \parallel y), b, w)$.

The second problem is complementary to the first, but slightly different. It might be called “the question of keeping”. Generally speaking, it must be decided whether in the calculation of $s \parallel t$ a subterm in the list of subterms of the “old” term t should be removed after it has been “used” once in the comparison, or whether it should be kept and may be used again.

Example 3.18. *What should the result be for the following expression?*

$$f(g(x), g(y), c) \parallel f(g(z), b, w)$$

If a term is removed from the list of “old” terms after it has been used once³³ the result is:

$$f(g(x \parallel z), g(y), c).$$

If the subterm $g(z)$ is not removed after the first match, the result will be:

$$f(g(x \parallel z), g(y \parallel z), c).$$

By introducing Restriction 2, I do not try to sweep the problems under the carpet, but as long as we work inside the simplified term model for XML-terms (which corresponds to the type `SElem`) we have too little information to be able to make the right decisions about which choices to make. When we extend the model to be a model for actual XML/HTML (which corresponds to the Haskell type `Element`) the situation might be different. As `Elements` can carry more information they might be compared in many more respects than just by comparing the tags (i.e. function symbols

³³And assuming that iteration over the list of subterms in the first argument starts with the leftmost term ($g(x)$)

3 A DIFFERENCE OPERATOR FOR TERMS

in the current model). `Elements` can differ not only in their tag, but also in their content and attributes. In an extended model we shall certainly have to allow two children in the same term to have the same tag.³⁴ And the results from the analysis of the simplified term model must be adapted in some way.

I also believe I might be in a better position to see what the right choices should be after we have gained some more experience with the simplified model, and after looking at some top-down examples of operations on “real” XML/HTML-documents. The model we work with now does not give many clues as to how to choose between terms in such a situation. The model is too coarse to give us any reasons to prefer one choice over the other. So I postpone this question by assuming that Restriction 2 holds.

It also turns out to be considerably more difficult to give a proper formal definition for \backslash without this restriction, among other things, \backslash would become indeterministic. We might not even find any definite solution to the two above-mentioned problems later either, and end up with a strategy which involves generating all the solutions and then use some selection criteria to choose from the set of candidates afterwards. I will come back to this question when I discuss the implementation of a difference function for non-simplified XML.

It should be noted that the definition of Restriction 2 is strictly related to which concept of equality is used as the basis for deciding whether two terms match each other partially.³⁵ As mentioned above this might be a domain sensitive question.

3.3 The definition of term difference

Term difference (\backslash) can now be defined on the basis of \prec . In the search of a definition I will identify the properties which \backslash must have in order for it to fulfil the intended purpose, namely to compute the difference between two terms (or rather “subtracting” one term from the other). This example might illustrate the purpose of \backslash :

³⁴As the reader will know this is more often than not the situation for HTML/XML documents.

³⁵As I have chosen $=_{R(0)}$ (i.e. \approx) as the condition for partial matching \approx is also used in Restriction 2. Remember that $s=_{R(0)} t \Leftrightarrow s \approx t \Leftrightarrow \mathcal{S}(s) = \mathcal{S}(t)$.

3.3 The definition of term difference

Example 3.19. *The following should hold:*

$$a \parallel a = \varepsilon \quad (27)$$

$$a \parallel b = a \quad (28)$$

$$f(a, b) \parallel f(a, b, c) = \varepsilon \quad (29)$$

$$f(a) \parallel f(b) = f(a) \quad (30)$$

$$f(a, d, e) \parallel f(a, b, c) = f(d, e) \quad (31)$$

$$f(a, g(x, z), c) \parallel f(a, g(x, z), c) = f(g(z), c) \quad (32)$$

Elementary set theory turns out to be an interesting source of inspiration in the search for a definition of \parallel . If we look to set theory we find a clue in the definition of set-minus (\setminus) in terms of the subset relation \subseteq . The definition of \setminus is as follows: For sets A, B, C, D

$$\begin{aligned} C = A \setminus B & \quad \text{if} \\ & C \subseteq A \\ & C \not\subseteq B \vee C = \emptyset \\ & \forall D (D \subseteq A \wedge D \not\subseteq B \rightarrow D \subseteq C) \end{aligned}$$

We take this definition as a starting point for the definition of \parallel . The \preceq relation will play the role of \subseteq . This approach also seems to fit well together with the intuitions about which properties \parallel should have. Not unexpectedly however, we will see that the analogy is not complete and that we therefore have to do some modifications to succeed.

Before we start to identify the properties which should hold for $s \parallel t$ note that the special case that $s \preceq t$ is treated separately. If $s \preceq t$ I suggest that it makes sense to claim that $s \parallel t = \varepsilon$.³⁶ This suggestion can be motivated by the following: I said earlier (Section 2.3) that \parallel is intended to have the property that $s \parallel t$ yields the new parts from s compared to t . If $s \preceq t$ there are no new parts in s compared to t , since s is in a sense “smaller” than t . Translated from the model perspective to the implementation perspective this is equivalent to the situation where we compare an old version of a web page to a new version of the same web page where something has been removed since the old version. Note especially that $s = t$ is a special case of $s \preceq t$, so this proviso also covers the case where nothing has changed on the web page.³⁷

³⁶In this sense \parallel can be compared to modified subtraction ($\dot{-}$) for natural numbers, for which we have e.g: $ss0 \dot{-} s0 = s0$, but $s0 \dot{-} sss0 = 0$

³⁷I added this special proviso for the case $s \preceq t$ at the final stage of the preparation

3 A DIFFERENCE OPERATOR FOR TERMS

The first property corresponds to the idea that $s \parallel t$ should not contain anything that is not already in s but always is smaller in some sense. In other words, \parallel should not add anything to the result $s \parallel t$ which did not come from s . For the property of “being smaller” we use the concept of cut. This property—which is called *left-cut*³⁸—can be formulated as follows:

$$s \parallel t \preceq s \quad (\textit{left-cut})$$

Next the property corresponding to $C \not\subseteq B \vee C = \emptyset$ is a property which ensures that what should have been removed from s in fact *is* removed in $s \parallel t$. I said earlier that in $s \parallel t$ only the parts of s which are “new” compared to t , or which have descendants which are new, should be kept. This can be expressed by saying that $s \parallel t$ should *not* be a cut of t . To understand why this is so, suppose $s \parallel t$ were a cut of t , then there would be (one or several) subterms in t which could be removed to get $s \parallel t$, which is contrary to the intention with \parallel . We call this property *not right-cut*.

Note that there is a special case for $s \preceq t$. If $s \preceq t$ then $s \parallel t = \varepsilon$ and ε is a cut of any term so we allow $s \parallel t$ to be a cut of t in this case. It is therefore assumed that the property *not right-cut* only holds when $s \not\preceq t$. *not right-cut* therefore becomes:

$$\text{If } s \not\preceq t \text{ then } s \parallel t \not\preceq t \quad (\textit{not right-cut})$$

The term equivalent to the third property in the definition of \backslash above, is a sort of a “minimality” criteria for left-cut. The purpose of this criteria is to restrict the set of solutions in a way such that the solution is not excessively cutted. $s \parallel t$ should be the largest term which has both of the above properties, or in other words the term which is “most equal” to, or “closest to” s without being a cut of t . Without this criteria, we would not be sure to get the solution we intended, but instead maybe “smaller” trivial solutions. This might be called a restriction “from below.” Another purpose with such a restriction criteria is to obtain uniqueness for \parallel .

A first attempt at formulating the minimality property is:

$$\text{If } s \not\preceq t \text{ then } \forall v (v \preceq s \wedge v \not\preceq t \rightarrow v \preceq s \parallel t) \quad (33)$$

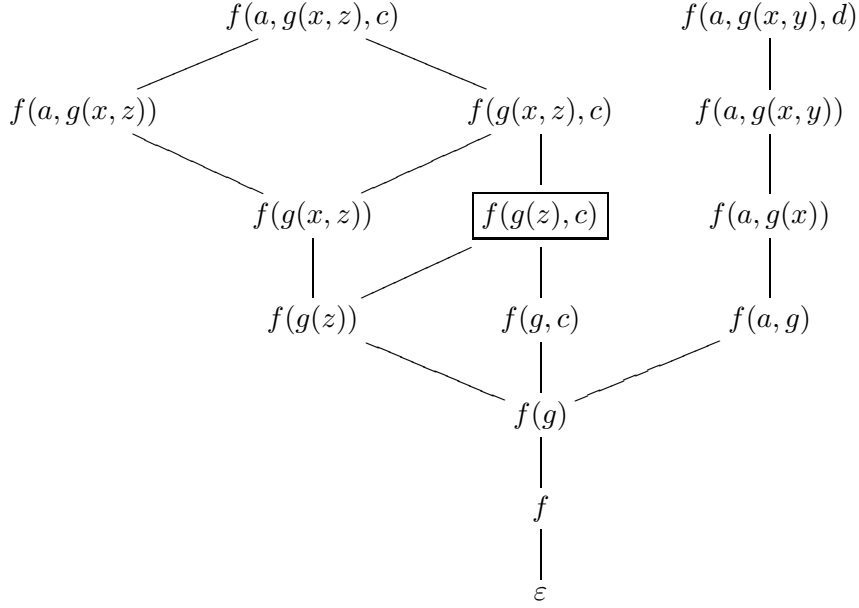
However it turns out that these three criteria are not sufficient to determine the intended solution. This can be seen from the following example:

of this thesis as I realised that it would be difficult to prove *not right-cut* without it. Initially only the case $s = t$ was treated as a special case and I believe that some of the proofs below regarding \parallel might be easier with this new proviso taken into consideration. Unfortunately however I did not have time to rewrite the proofs at such a late stage.

³⁸As it expresses that the solution is a cut of the left side argument to \parallel .

3.3 The definition of term difference

Consider the following diagram which displays the cut relationships for the terms in the example (32) above. An *upward* line from v to u signifies $v \preceq u$. (The diagram does not show all cut relationships, only enough to illustrate the point. The intended solution is framed in.)



The property (33) will for example prevent $f(g(z))$ from being a solution, because there exists a term $v = f(g(z), c)$, (namely the intended solution) such that $v \preceq s \wedge v \not\preceq t$ but also $(v \not\preceq f(g(z)))$, and (33) is false. At first sight this seems to be the sort of restriction from below which we are after. Unfortunately however (33) also prevents the intended solution $f(g(z), c)$ from being a solution due to the existence of $f(g(x, z), c)$. (Because $f(g(x, z), c) \preceq s \wedge (f(g(x, z), c) \not\preceq t)$ but also $\neg(f(g(x, z), c) \preceq f(g(z), c))$, and (33) is false.

The problem occurs because we try to enforce a restriction from below on a too large set of candidates. The set of candidates eligible to be a solution must be restricted by some additional property prior to the restriction from below. The question then is, on what criteria should a term like $f(g(x, z), c)$ be made ineligible? If we compare $f(g(x, z), c)$ to $f(a, g(x, y), d)$ we see that although the former is not a *cut* of the latter it contains a subterm, namely x which also occurs in $f(a, g(x, y), d)$, and accordingly is not “new” in s . So we need to find a way to express that we do not want to have “junk” like this in the result.

3 A DIFFERENCE OPERATOR FOR TERMS

In other words, the term that is the solution to $s \parallel t$ is not only a term which is not a cut of t , it is also *the* term which has the *least* in common with t compared to any other candidate. To be able to find a way to express this property we shall need the following auxiliary concept:

Definition 3.20 (Term intersection, $s \pitchfork t$). *Let s, t be terms such that Restriction 2 holds. A term u is the intersection of s, t , written $s \pitchfork t$, if the following holds:*

$$u \preceq t \tag{34}$$

$$u \preceq s \tag{35}$$

$$\forall v (v \preceq t \wedge v \preceq s \rightarrow v \preceq u) \tag{36}$$

Intuitively $s \pitchfork t$ is the “largest” term which is both a cut of s and of t .

The reason that the Restriction 2 occurs in the definition is to secure a unique result for \pitchfork . Without it $s \pitchfork t$ is not unique, as can be seen from the following example with a term : $f(g(b), g(a))$ and $f(g(a), g(b))$ for which the restriction do not hold.

$$f(g(b), g(a)) \pitchfork f(g(a), g(b)) = f(g(b), g(a))$$

$$f(g(b), g(a)) \pitchfork f(g(a), g(b)) = f(g(a), g(b))$$

both solutions satisfy the definition of \pitchfork above.

With Restriction 2, \pitchfork is unique. However I do not have a proof for this so it remains a claim:

Claim 3.21 (Intersection uniqueness).

$$u = s \pitchfork t \wedge u' = s \pitchfork t \rightarrow u \equiv u'$$

Note that \pitchfork is associative:

Observation 3.22. $s \pitchfork t = t \pitchfork s$.

Example 3.23.

$$f(\dots) \pitchfork g(\dots) = \varepsilon$$

$$f() \pitchfork f(a, b, c) = f()$$

$$f(a, b) \pitchfork f(b, c) = f(b)$$

$$f(a, g(b)) \pitchfork f(g(a), b) = f(g())$$

Lemma 3.24.

$$s \pitchfork s = s$$

Proof. (Sketch) By contradiction. Take as witness $v = s$ and use antisymmetry. \square

With the concept of term intersection, the new property, which might be called *no junk* can be formulated thus:

$$\text{If } s \not\leq t \text{ then } \forall v (v \neq \varepsilon \wedge v \preceq s \wedge v \not\leq t \rightarrow s \pitchfork t \pitchfork t \preceq v \pitchfork t) \quad (\text{no junk})$$

The three properties together, secures that the term $s \pitchfork t$ is a term which satisfies *left-cut* and *not right-cut* and which also has less in common with t than any other term v which satisfies *left-cut* and *not right-cut*. *no junk* enforces what could be called a restriction “from above”.

With the *no junk* property, the term $f(g(x, z), c)$ is cut off as a solution candidate because it has more in common with t than has the intended solution $f(g(z), c)$:

$$\begin{aligned} f(g(x, z), c) \pitchfork f(a, g(x, y), d) &= f(b(x)) \\ f(g(z), c) \pitchfork f(a, g(x, y), d) &= f(b) \end{aligned}$$

and $f(b(x)) \not\leq f(b)$, thus $f(g(x, z), c)$ does not satisfy *no junk*.

With this third property in place, we can revise the minimality criteria. The minimality criteria now says that: If $u = s \pitchfork t$ then for any term w such that *left-cut*, *not right-cut* and *no junk* holds, $w \preceq u$. Together these four properties define *term difference*:

3 A DIFFERENCE OPERATOR FOR TERMS

Definition 3.25 (Term difference, $s \parallel t$). Let s, t, v, w be terms such that Restriction 2 holds. Then a term u is the difference of s, t , written $s \parallel t$, given as follows. If $s \preceq t$ then $u = \varepsilon$. Otherwise (i.e., $s \not\preceq t$) the following must hold:

$$\begin{aligned}
 u &\preceq s && \text{(left-cut)} \\
 u &\not\preceq t && \text{(not right-cut)} \\
 \forall v (v \neq \varepsilon \wedge v \preceq s \wedge v \not\preceq t \rightarrow u \pitchfork t \preceq v \pitchfork t) &&& \text{(no junk)} \\
 \\
 \forall w (w \neq \varepsilon \wedge w \preceq s \wedge w \not\preceq t \wedge &&& \text{(minimal left-cut)} \\
 \forall v (v \neq \varepsilon \wedge v \preceq s \wedge v \not\preceq t \rightarrow w \pitchfork t \preceq v \pitchfork t) \rightarrow &&& \\
 w \preceq u) &&&
 \end{aligned}$$

With these properties $s \parallel t$ is unique. This is important since if it were not $s \parallel t$ would be undetermined.

Lemma 3.26 (Difference uniqueness).

$$u = s \parallel t \wedge u' = s \parallel t \rightarrow u = u'$$

Proof. Assume that

$$u = s \parallel t \wedge u' = s \parallel t \tag{37}$$

but then by *minimal left-cut*

$$\forall w (w \preceq u) \wedge \forall w (w \preceq u') \tag{38}$$

and in particular

$$u' \preceq u \wedge u \preceq u'. \tag{39}$$

By antisymmetry of \preceq we get that:

$$u' = u. \tag{40}$$

□

4 The term difference algorithm

In the following sections I present an algorithm (tmd) for computing $s \parallel t$. If we assume that Restriction 1 and 2 above holds, the following can be noted about tmd:

1. The algorithm is only defined for terms $s, t \neq \varepsilon$.
2. If s and t are identical, it makes no sense to calculate their difference, so we must consider what the value of $\text{tmd}(s, t)$ should be if $s = t$. One choice could be s , as that is what we probably would want an implementation of the function to display to the user. However that is an implementation detail, which should be handled at the implementation layer. If we look at the intended meaning of tmd (i.e. a function which shows what is new in s compared to t) it seems to make more sense to let the value be nothing, since nothing has changed. A similar argument applies to the more general case that $s \preceq t$ ³⁹ If $s \preceq t$ there are no new parts in s compared to t , since s is in a sense “smaller” than t . We could have let the result of $\text{tmd}(s, t)$ be the empty string in both cases but I prefer to say that $\text{tmd}(s, t)$ is undefined when $s \preceq t$.⁴⁰
3. According to Restriction 1 $s \parallel t$ is undefined if t and s are not isomorphic. We shall say that $\text{tmd}(s, t)$ also is undefined for such cases.
4. According to Restriction 2 $s \parallel t$ is undefined if any of the terms have subterms where several children are tag equal to each other. In these cases $\text{tmd}(s, t)$ is undefined.
5. If none of the above is the case, the situation is that there is no conflict with the domain restrictions, s and t differs in some way and $s \preceq t$ We then build a new term by taking the function symbol of s and build a list of children in this new term by comparing the children of s against the children in t .

The definition of tmd is:

³⁹ $s = t$ implies $s \preceq t$

⁴⁰ For the corresponding operator \parallel , $s \parallel t = \varepsilon$ if $s \preceq t$.

4 THE TERM DIFFERENCE ALGORITHM

Term difference (tmd)

Let $s = g(u_1, \dots, u_n), t = f(t_1, \dots, t_n)$ be terms such that:

$$\begin{aligned} s &\not\sim t \\ s &\sim t \end{aligned} \tag{R1}$$

For any subterm $f(t_1, \dots, t_n)$ in s and t the following holds:

$$\forall i, j \quad \mathcal{S}(t_i) = \mathcal{S}(t_j) \Rightarrow i = j \quad 1 \leq i, j \leq n \tag{R2}$$

$$\begin{aligned} \text{tmd}(g(u_1, \dots, u_n), f(t_1, \dots, t_n)) = \\ g(\text{tmd}_l((u_1, \dots, u_n), (t_1, \dots, t_n))) \end{aligned} \tag{tmd}$$

The algorithm tmd_l has as its domain lists of terms and as its co-domain lists of terms. It is defined as follows:

Term list difference (tmd_l)

$$\text{tmd}_l((), (t_1, \dots, t_n)) = () \tag{tl_0}$$

$$\begin{aligned} \text{tmd}_l((u_1, \dots, u_m), (t_1, \dots, t_n)) = \\ \text{tmd}_l((u_2, \dots, u_m), (t_1, \dots, t_n)) \end{aligned} \tag{tl_1}$$

if $\exists i(u_1 = t_i)$ and (tl₀) does not apply,

$$\text{tmd}(u_1, t_i) : \text{tmd}_l((u_2, \dots, u_m), (t_1, \dots, t_n)) \tag{tl_2}$$

if $\exists i(\mathcal{S}(u_1) = \mathcal{S}(t_i))$ and (tl₀, tl₁) do not apply,

$$u_1 : \text{tmd}_l((u_2, \dots, u_m), (t_1, \dots, t_n)) \tag{tl_3}$$

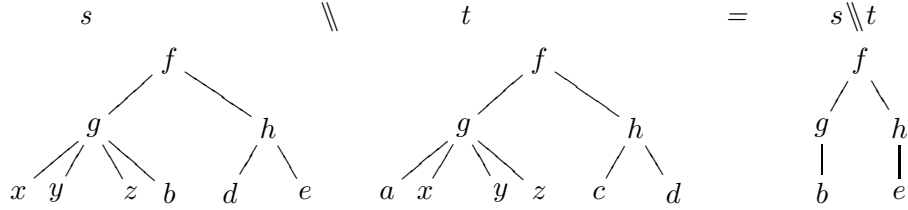
if none of the above apply

i.e. $\forall i(u_1 \neq t_i \wedge \mathcal{S}(u_1) \neq \mathcal{S}(t_i))$.

So what is the meaning of (tl₀–tl₃)? They describe the computation of the list of terms which becomes the children in $\text{tmd}(s, t)$. As mentioned before, this list of children results from comparing the children of s (i.e. u_1, \dots, u_m) and the children of t (i.e. t_1, \dots, t_n). Each u_l is compared to t_1, \dots, t_n . tmd_l is recursive and if the list of us is—or has become—empty, the empty list is returned. This is the recursion bottom (tl₀). If the list is not empty, the first u in the list is compared to the list of ts . If this u_l is identical to some t_i , it is dropped (tl₁). If the u_l is not equal to any t , but has the same outermost function symbol as some t , we know that one

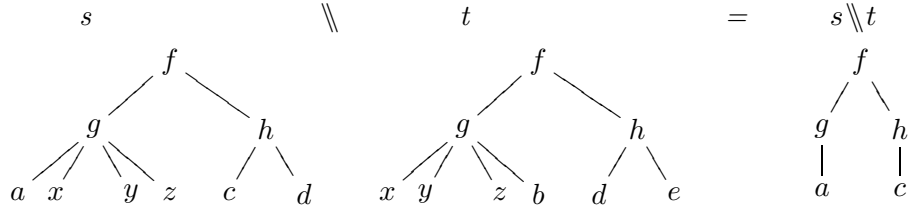
or several of their subterms differ. In this case a new term ($\text{tmd}(u_l, t_k)$) is added to the beginning of the result list (tl_2). Finally if an u_l is not equal, nor has the same outermost function symbol as any t_k , this u_l is regarded as “new” and is included in the result list (tl_3).

Example 4.1.



tmd is not commutative, as is seen from the following example:

Example 4.2.



It can also be noted that even if t and s are isomorphic, $\text{tmd}(s, t)$ is in general neither isomorphic to t nor to s .

As I mentioned in the discussion of the domain restrictions, if it were not for Restriction 2, there could have been several t_i s which matched (fully, or partially) one and the same u_l . We decided to postpone this difficulty to a later stage. However if Restriction 2 was dropped, and we had a method for choosing the right t_i from a list of candidates, it makes sense to assume that that particular t_i which were chosen to use as the term to compare the u_l against would be used only at one occasion. Thus that particular t_i should be excluded from the list of candidates to compare the rest of the u s against. This leads to alternative formulations of (tl_1) and (tl_2):

$$\begin{aligned}
 & \text{tmd}_l((u_1, \dots, u_m), (t_1, \dots, t_n)) = \\
 & \quad \text{tmd}_l((u_2 \dots u_m), (t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n)) \quad (\text{tl}_1^*) \\
 & \quad \quad \text{if } \exists i (u_1 = t_i) \text{ and } (\text{tl}_0) \text{ does not apply,} \\
 & \quad \text{tmd}(u_1, t_i) : \text{tmd}_l((u_2 \dots u_m), (t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n)) \quad (\text{tl}_2^*) \\
 & \quad \quad \text{if } \exists i (\mathcal{S}(u_1) = \mathcal{S}(t_i)) \text{ and } (\text{tl}_0, \text{tl}_1) \text{ does not apply.}
 \end{aligned}$$

4 THE TERM DIFFERENCE ALGORITHM

Note that as long as Restriction 2 is in effect, the starred and unstarred versions of (tl_1) and (tl_1) gives the same result.

4.1 Correctness for the algorithm

I propose that the algorithm tmd is correct with regards to the definition of \parallel . In other words:

Theorem 4.3. *For terms s, t such that Restriction 1 and Restriction 2 holds and for term $u \neq \varepsilon$:*

$$\text{If } u = tmd(s, t) \text{ then } u = s \parallel t \quad (41)$$

Note that due to the restrictions discussed above tmd is a *partial* algorithm, (it is not defined for any pair of terms).

When I first introduced the restriction on the domain for tmd , that the two arguments to the algorithm have to be isomorphic it was more for the sake of understanding the operation rather than for strictly formal reasons. Since Restriction 1 is quite strict and therefore narrows the domain for tmd considerably I had hoped that Theorem 4.3 could be proved to hold without regard to the restriction. However as the work progressed I realised that I was unable to prove the property *no junk* without it, so it had to be included in the formulation of Theorem 4.3. For a later work it is interesting to investigate what it takes to get rid of it.

In accordance with the definition of \parallel Theorem 4.3 can be divided into four theorems where each corresponds to one of the four defining properties in Definition 3.25.

Theorem 4.3.1 (left-cut). *For terms s, t such that Restriction 2 holds and for term $u \neq \varepsilon$:*

$$\text{If } u = tmd(s, t) \text{ then } u \preceq s \quad (42)$$

Theorem 4.3.2 (not right-cut). *For terms s, t such that Restriction 2 holds and for term $u \neq \varepsilon$:*

$$\text{If } u = tmd(s, t) \text{ then } u \not\preceq t \quad (43)$$

Theorem 4.3.3 (no junk). *For terms s, t such that Restriction 1 and Restriction 2 holds, and for term $u \neq \varepsilon$:*

$$\begin{aligned} \text{If } u = tmd(s, t) \text{ then} & \quad (44) \\ \forall v (v \neq \varepsilon \wedge v \preceq s \wedge v \not\preceq t \rightarrow u \pitchfork t \preceq v \pitchfork t) & \end{aligned}$$

Theorem 4.3.4 (minimal left-cut). *For terms s, t such that Restriction 1 and Restriction 2 holds, and for term $u \neq \varepsilon$:*

$$\begin{aligned}
 & \text{If } u = \text{tmd}(s, t) \text{ then} & (45) \\
 & \forall w(w \neq \varepsilon \wedge w \preceq s \wedge w \not\preceq t \wedge \\
 & \quad \forall v(v \neq \varepsilon \wedge v \preceq s \wedge v \not\preceq t \rightarrow w \text{ m } t \preceq v \text{ m } t) \rightarrow \\
 & \quad w \preceq u)
 \end{aligned}$$

Before we can prove the theorems we need to prove some lemmas, therefore the proofs of the four theorems follow in sections 4.4–4.7.

4.2 The list cut relation

To facilitate the proof of Theorem 4.3 I will define the new relation *proper list cut* \prec_l and its generalisation *list cut* \preceq_l . They are both relations on lists, analogous to \prec and \preceq for terms and play important roles in the arguments that follow. I start by defining some more basic concepts with regards to lists, these concepts are necessary for the definitions of \prec_l and \preceq_l .

Definition 4.4 (Positions in list (\mathcal{P}^l)). *A position in a list is a positive integer.*

$$\begin{aligned}
 \mathcal{P}^l() &= \emptyset \\
 \mathcal{P}^l(t_1, \dots, t_n) &= \{1, \dots, n\}
 \end{aligned}$$

Definition 4.5 (List replacement). *If l is a list, $i \in \mathcal{P}^l(l)$ and u is a term, then $l[u]_i^l$ is the list l where the i -th element is replaced by u :*

$$(t_1, \dots, t_i, \dots, t_n)[u]_i^l = (t_1, \dots, u, \dots, t_n)$$

Primarily for ease of notation I also define the *length* of a list, and the concept of *an element in a list at position i* .

Definition 4.6 (Length of list).

$$\begin{aligned}
 \mathcal{L}() &= 0 \\
 \mathcal{L}(t_1, \dots, t_n) &= n
 \end{aligned}$$

Definition 4.7 (Element in list).

$$(t_1, \dots, t_n)!i = t_i \quad \text{if } 1 \leq i \leq n.$$

4 THE TERM DIFFERENCE ALGORITHM

With regards to the relation between positions in a list and positions in a term, note that for any list l , and for any function symbol f , if p is a position in the list l then p is a position in the term $f(l)$.

Observation 4.8.

$$\text{If } p \in \mathcal{P}^l(l) \text{ then } p \in \mathcal{P}(f(l)) \quad (46)$$

Also note that if t is a term at position p in the list (t_1, \dots, t_n) then t is the subterm at position p in the term $f(t_1, \dots, t_n)$.

Observation 4.9.

$$\forall l, f, p \text{ If } l!p = t \text{ then } f(l)|_p = t \quad (47)$$

And finally note this relation between list substitution and term substitution for positions of length 1:

Observation 4.10.

$$\forall l, f, p, t \text{ len}(p) = 1 \rightarrow f(l[t]_p^l) = f(l)[t]_p \quad (48)$$

The idea with the \prec_l relation is that a list l is said to be a list cut of another list r if: Either l can be made from r by removing a term from the list. Or if l can be made from r by replacing some term u in l with a term t , such that $t \preceq u$: The definition for \prec_l is:

Definition 4.11 (Proper list cut \prec_l). For lists $l = (u_1, \dots, u_m)$ and $r = (t_1, \dots, t_n)$:

$$l \prec_l r \quad \text{if} \quad \exists i \ l = r[\varepsilon]_i^l \quad (0 < i \leq \mathcal{L}(r)) \quad \text{or} \quad (49)$$

$$\exists i, t \ l = r[t]_i^l \wedge t \preceq (r!i) \quad (0 < i \leq \mathcal{L}(r)) \quad (50)$$

Example 4.12.

$$(a, b) \prec_l (a, b, c) \quad (51)$$

$$(a, f(x)) \prec_l (a, f(x, y)) \quad (52)$$

\preceq_l is the reflexive transitive closure of \prec_l :

Definition 4.13 (List cut \preceq_l). For lists l and r :

$$l \preceq_l r \quad \text{if} \quad l = r \quad \text{or} \quad (53)$$

$$\exists l' \ l \prec_l l' \wedge l' \preceq_l r \quad (54)$$

Example 4.14.

$$(a) \preceq_l (a, b, c) \tag{55}$$

Corollary 4.15 (Distance between lists). *If $l \preceq_l r$ then there exists a least n and lists $l_0 \dots l_n$ such that*

$$l = l_0 \wedge l_0 \prec_l l_1 \wedge \dots \wedge l_{n-1} \prec_l l_n \wedge l_n = r \tag{56}$$

and this n is the distance between l and r , written $\mathcal{D}_l(l \preceq_l r)$.

Example 4.16.

$$\begin{aligned} \mathcal{D}_l(l \preceq_l l) &= 0 \\ \mathcal{D}_l((a, f(x)) \preceq_l (a, f(x, y))) &= 1 \\ \mathcal{D}_l((a, b) \preceq_l (a, b, c)) &= 1 \\ \mathcal{D}_l((a) \preceq_l (a, b, c)) &= 2 \end{aligned}$$

Theorem 4.17. \preceq_l is reflexive.

Proof. Follows directly from Definition 4.13 (53). □

Theorem 4.18. \preceq_l is transitive.

Proof. The proof is similar to the proof of Theorem 3.14 and is left out. □

4.3 Auxiliary lemmas

For the proof of Theorem 4.3 we need some lemmas. They are presented and proved below. The proof of Theorem 4.3 follows in Section 4.5. The lemmas and their order of presentation are as follows: First I prove Lemma 4.19 and Lemma 4.20 they regard properties of \prec and \preceq_l . They will be used in the proof of Lemma 4.21. The latter expresses an important relation between \preceq and \preceq_l . The lemmas 4.22–4.25 and Lemma 4.28 express properties of \prec and \preceq_l which are needed in the proof of Theorem 4.3. Lemma 4.26 and Lemma 4.27 expresses some useful properties of tmd .

Lemma 4.19 states that if in a term s a *subterm* is replaced with a cut of itself, then the resulting term is a cut of the original term s :

Lemma 4.19.

$$\text{If } \exists p \ p \in \mathcal{P}(s) \wedge t \preceq s|_p \text{ then } s[t]_p \preceq s. \tag{57}$$

4 THE TERM DIFFERENCE ALGORITHM

Proof. The proof is by induction on $\mathcal{D}(t \preceq s|_p)$.

Basis: For $\mathcal{D}(t \preceq s|_p) = 0$, $t = s|_p$ and then $s[t]_p = s[s|_p]_p = s$ and $s \preceq s$ holds by (16). If $\mathcal{D}(t \preceq s|_p) = 1$, $t \prec s|_p$. Consider a term s with a subterm in a position p , and a term t such that $t \prec s|_p$. If

$$\exists p \ p \in \mathcal{P}(s) \wedge t \prec s|_p \quad (58)$$

then by Definition 3.6

$$\exists p \exists p' \ p \in \mathcal{P}(s) \wedge p' \in \mathcal{P}(s|_p) \text{ such that } t = (s|_p)[\varepsilon]_{p'}. \quad (59)$$

For these p and p' :

$$s[t]_p = s[(s|_p)[\varepsilon]_{p'}]_p. \quad (60)$$

But as $p \in \mathcal{P}(s)$, $s|_p$ is a subterm of s and $p' \in \mathcal{P}(s|_p)$ we can from p and p' find a position $p'' \in \mathcal{P}(s)$ ⁴¹ such that

$$s[t]_p = s[\varepsilon]_{p''}. \quad (61)$$

By the definition of \preceq it follows that

$$s[t]_p \preceq s. \quad (62)$$

Induction step: Assume as the induction hypothesis that (57) holds for terms s and $s|_p$ as described above, where $\mathcal{D}(t \preceq s|_p) = n$. If

$$\exists p \ p \in \mathcal{P}(s) \wedge t \preceq s|_p \quad \mathcal{D}(t \preceq s|_p) = n + 1 \quad (63)$$

then by Definition 3.6:

$$\exists p \exists t' \ p \in \mathcal{P}(s) \wedge t \prec t' \wedge t' \preceq s|_p \quad \mathcal{D}(t' \preceq s|_p) = n. \quad (64)$$

By the induction hypothesis (since $s|_p$ is a subterm in position p in s and $t' \preceq s|_p$):

$$s[t']_p \preceq s \quad (65)$$

t' is obviously a subterm in position p in $s[t']_p$ and $t \prec t'$ by (64). Therefore, as was just proved for the basis case

$$\begin{aligned} (s[t']_p)[t]_p \preceq s[t']_p &\Rightarrow \\ (s[t]_p) \preceq s[t']_p & \end{aligned} \quad (66)$$

and

$$(s[t]_p) \preceq s \quad (67)$$

follows from (65) and (66) by transitivity of \preceq . \square

⁴¹In fact $p'' = p.p'$.

Lemma 4.20 states that if in a term s some subterm is substituted with ε the list of immediate subterms in the resulting term is a list cut of the list of immediate subterms in the original term s :

Lemma 4.20.

$$\exists p \ p \in \mathcal{P}(g(r)) \wedge g(l) = g(r)[\varepsilon]_p \rightarrow l \prec_l r \quad (68)$$

Proof. By case exhaustion on the length of the position p .

- If $p = \varepsilon$ then

$$g(l) = g(r)[\varepsilon]_\varepsilon = \varepsilon$$

this makes the antecedent of (68) false and consequently (68) true.

- If $\text{len}(p) = 1$:

$$g(l) = g(r)[\varepsilon]_p = g(r[\varepsilon]_p) \text{ By Obs 4.10} \Rightarrow \quad (i)$$

$$l = r[\varepsilon]_p \quad PC \quad \Rightarrow \quad (ii)$$

$$l \prec_l r \quad \text{by Def 4.11} \quad (iii)$$

- If $\text{len}(p) > 1$, then p is on the form $i.p'$, where $i \neq \varepsilon$:

$$g(l) = g(u_1, \dots, u_m)[\varepsilon]_{i.p'} \quad \text{Assumption} \Rightarrow \quad (iv)$$

$$g(l) = g(u_1, \dots, u_i[\varepsilon]_{p'}, \dots, u_n) \text{ by Def 2.16} \Rightarrow \quad (v)$$

$$l = (u_1, \dots, u_i[\varepsilon]_{p'}, \dots, u_n) \quad \Rightarrow \quad (vi)$$

$$l = (u_1, \dots, u_m)[u_i[\varepsilon]_{p'}]_i^l \quad \text{by Def 4.5} \Rightarrow \quad (vii)$$

$$u_i[\varepsilon]_{p'} \prec u_i \quad \text{by Def 3.3} \Rightarrow \quad (viii)$$

$$l \prec_l (u_1, \dots, u_m) \quad \text{by vii, viii, 50} \quad (ix)$$

□

Lemma 4.21 expresses an important relation between \preceq and \preceq_l and the lemma is integral to the proof of Theorem 4.3.

Lemma 4.21. For an arbitrary (non-empty) function symbol g and for lists l, r :

$$l \preceq_l r \text{ if and only if } g(l) \preceq g(r). \quad (69)$$

Each direction, from the left to the right and vice versa is proved in turn. We start with (70):

$$\text{If } l \preceq_l r \text{ then } g(l) \preceq g(r) \quad (70)$$

4 THE TERM DIFFERENCE ALGORITHM

Proof. If $l = r$, then also $g(l) = g(r)$ and $g(l) \preceq g(r)$ follows by Definition 3.6. If $l \neq r$ the proof is by induction on the distance between l and r .

Basis: The basis case is where $\mathcal{D}(l \preceq_l r) = 1$, i.e. where $l \prec_l r$, and the following must be proved:

$$\text{If } l \prec_l r \text{ then } g(l) \preceq g(r) \quad (71)$$

Assume that $l \prec_l r$, then from the definition of \prec_l (Definition 4.11) we know that either (49) or (50) is the case.

• If $l \prec_l r$ holds by (49) then:

$$\exists i \ l = r[\varepsilon]_i^l \quad (0 < i \leq \mathcal{L}(r)) \quad (72)$$

But then for this i :

$$\begin{aligned} g(l) &= g(r[\varepsilon]_i^l) \\ &= g(r)[\varepsilon]_i \quad i \in \mathcal{P}(g(r)) \text{ by (46) and (48)}. \end{aligned} \quad (73)$$

But if

$$\exists i \ g(r)[\varepsilon]_i \quad i \in \mathcal{P}(g(r)), \quad (74)$$

then

$$g(r)[\varepsilon]_i \preceq g(r) \quad (75)$$

by Definition 3.6. And

$$g(l) \preceq g(r) \quad (76)$$

follows from (75) and (73)

• If $l \prec_l r$ holds by (50) then:

$$\exists i, t \ l = r[t]_i^l \wedge t \preceq (r!i) \quad (0 < i \leq \mathcal{L}(r)) \quad (77)$$

But then for these i and t :

$$\begin{aligned} g(l) &= g(r[t]_i^l) \\ g(l) &= g(r)[t]_i \quad i \in \mathcal{P}(g(r)) \quad \text{by (46) and (48)}. \end{aligned} \quad (78)$$

And as

$$(r!i) = g(r)|_i \quad \text{by (47)}$$

and

$$t \preceq (r!i) \quad \text{by (77)}$$

then

$$t \preceq g(r)|_i. \quad (79)$$

From (57) and (79) it follows that:

$$g(r)[t]_i \preceq g(r) \quad (80)$$

which, (since $g(l) = g(r)[t]_i$) is equivalent to

$$g(l) \preceq g(r) \quad (81)$$

This concludes the basis part of the proof of (70).

Induction step: Assume as the induction hypothesis that (70) holds for any lists l, r with distance n between them. Assume

$$l \preceq_l r, \quad (82)$$

where $l \neq r$ and the distance between l and r is $n + 1$. From (82) we get:

$$\exists l' \quad l \prec_l l' \wedge l' \preceq_l r \quad \text{by Definition 4.13} \quad (83)$$

But then:

$$\exists l' \quad g(l) \prec g(l') \wedge g(l') \preceq g(r) \quad (84)$$

where the first conjunct follows from the first conjunct of (83) as was just proved above.⁴², and the second follows from (83) by the induction hypothesis. And

$$g(l) \preceq g(r).$$

follows from (84) by Definition 4.13 (54). This concludes the induction step of the proof of (70). \square

For the other direction of Lemma 4.21 it must be proved that:

$$\text{If } g(l) \preceq g(r) \text{ then } l \preceq_l r \quad (85)$$

⁴²In the basis part of this inductive argument.

4 THE TERM DIFFERENCE ALGORITHM

Proof. For the case that $g(l) = g(r)$, then also $l = r$ and $l \preceq_l r$ follows by Definition 4.13. If $g(l) \neq g(r)$ the proof is by induction on $\mathcal{D}(g(l) \preceq g(r))$.

Basis: The basis is where $\mathcal{D}(g(l) \preceq g(r)) = 1$, i.e. $g(l) \prec g(r)$:

$$\begin{array}{lll}
 g(l) \prec g(r) & \text{Assumption} \Rightarrow & \text{(i)} \\
 \exists p \ p \in \mathcal{P}(g(r)) \wedge g(l) = g(r)[\varepsilon]_p & \text{By Def 3.3} \Rightarrow & \text{(ii)} \\
 \exists p \ p \in \mathcal{P}(g(r)) \wedge g(l) = g(r)[\varepsilon]_p \rightarrow l \prec_l r & \text{Lemma 4.20} \Rightarrow & \text{(iii)} \\
 l \prec_l r & \text{PC} \Rightarrow & \text{(iv)} \\
 g(l) \prec g(r) \rightarrow l \prec_l r & \text{PC} \Rightarrow & \text{(v)}
 \end{array}$$

This concludes the basis part of the proof of (85).

Induction step: Assume as the induction hypothesis that (85) holds for any terms $g(l), g(r)$ such that $\mathcal{D}(g(l) \preceq g(r)) = n$.

Assume that:

$$g(l) \preceq g(r) \quad \mathcal{D}(g(l) \preceq g(r)) = n + 1$$

From Definition 3.6:

$$\exists l' \ g(l) \prec g(l') \wedge g(l') \preceq g(r) \quad \mathcal{D}(g(l') \preceq g(r)) = n$$

It was shown above that:

$$g(l) \prec g(l') \rightarrow l \prec l'$$

and

$$l' \preceq_l r$$

follows from the induction hypothesis. And with transitivity of \preceq_l ,

$$l \preceq r$$

follows. This concludes the proof of (85) and hence of Lemma 4.21. \square

Lemma 4.22 states a property about \preceq_l .

Lemma 4.22. *For any lists l, r and any term t :*

$$\text{If } l \preceq_l r \text{ then } (t:l) \preceq_l (t:r), \tag{86}$$

4.3 Auxiliary lemmas

Proof. If $l = r$, then certainly $t:l = t:r$. If $l \neq r$, the proof is by induction on the distance between l and r . There are two base cases (i.e. where $l \prec_l r$) corresponding to the two disjuncts in the definition of \prec_l .

Basis:

$$\text{If } l \prec_l r \text{ then } (t:l) \preceq_l (t:r) \quad (87)$$

• If $l \prec_l r$ by (49) then:

$$\exists i \ l = r[\varepsilon]_i^l \quad (0 < i \leq \mathcal{L}(r)).$$

But then it is also the case that

$$\begin{aligned} \exists i \ (t:l) &= t:(r[\varepsilon]_i^l) & (0 < i < \mathcal{L}(t:r)) &\Rightarrow \\ \exists j \ (t:l) &= (t:r)[\varepsilon]_j^l & (j = i + 1 \wedge 0 < j \leq \mathcal{L}(t:r)). \end{aligned}$$

By (49) and Definition 4.13:

$$(t:l) \preceq_l (t:r)$$

follows.

• If $l \prec_l r$ by (50) then

$$\exists i, u \ l = r[u]_i^l \wedge u \preceq r!i \quad (0 < i \leq \mathcal{L}(r)).$$

But then it is also the case that

$$\begin{aligned} \exists i, u \ t:l &= t:r[u]_i^l \wedge u \preceq r!i & (0 < i < \mathcal{L}(t:r)) &\Rightarrow \\ \exists i, u \ t:l &= (t:r)[u]_j^l \wedge u \preceq (t:r)!j & (j = i + 1 \wedge 0 < j \leq \mathcal{L}(t:r)). \end{aligned}$$

By (50) and Definition 4.13:

$$(t:l) \preceq_l (t:r)$$

follows.

Induction step: As the induction hypothesis assume that Lemma 4.22 holds for any lists l, r with distance n between them, and assume:

$$l \preceq_l r \quad (88)$$

where the distance between l and r is $n + 1$. By Definition 4.13 it follows that

$$\exists l' \ l \prec_l l' \wedge l' \preceq_l r. \quad (89)$$

4 THE TERM DIFFERENCE ALGORITHM

And for this l'

$$\exists l' \quad t:l \prec_l t:l' \tag{90}$$

follows from the first conjunct of (89) as was just proved above for the basis case. And

$$\exists l't:l' \preceq_l t:r. \tag{91}$$

follows from the second conjunct (89) by the induction hypothesis (86). By Definition 3.6 we get:

$$t:l \preceq_l t:r.$$

□

Lemma 4.23 states that if there is an immediate subterm in $g(t_1, \dots, t_n)$ which is not a cut of any immediate subterm s_j in $g(s_1, \dots, s_m)$ then $g(t_1, \dots, t_n)$ cannot be a cut of $g(s_1, \dots, s_m)$.

Lemma 4.23. *For terms $g(t_1, \dots, t_n)$ and $g(s_1, \dots, s_m)$*

$$(\exists i(1 \leq i \leq n) \forall j(1 \leq j \leq m) \quad t_i \not\preceq s_j) \rightarrow g(t_1, \dots, t_n) \not\preceq g(s_1, \dots, s_m). \tag{92}$$

Proof. I prove the contrapositive of (92). If

$$g(t_1, \dots, t_n) \preceq g(s_1, \dots, s_m)$$

then by Lemma 4.21

$$t_1, \dots, t_n \preceq_l s_1, \dots, s_m$$

but then

$$\forall i \exists j \quad t_i \preceq s_j \tag{93}$$

For if (93) is not the case, s_1, \dots, s_m cannot be made equal to t_1, \dots, t_n by removing subterms from s_1, \dots, s_m or replacing a number of subterms with a cut of themselves.⁴³ □

⁴³Remember that if $t_i = s_j$ then $t_i \preceq s_j$.

Corollary 4.24. For lists t_1, \dots, t_n and s_1, \dots, s_m

$$(\exists i(1 \leq i \leq n) \forall j(1 \leq j \leq m) t_i \not\leq s_j) \rightarrow t_1, \dots, t_n \not\leq s_1, \dots, s_m. \quad (94)$$

Proof. Follows from Lemma 4.23 by Lemma 4.21. \square

Lemma 4.25 is an intermediate result which is needed in the proof of *no junk*.

Lemma 4.25. If

$$g(\bar{w}) \not\leq g(\bar{u}) \wedge g(\bar{w}) \preceq g(u:\bar{u}) \quad (95)$$

then

$$\begin{aligned} \bar{w} &= u:\bar{u} && \text{or} \\ \bar{w} &= (u':\bar{u}) \wedge u' \preceq u. \end{aligned} \quad (96)$$

Proof. From (95) and Lemma 4.21 it follows that:

$$\bar{w} \not\leq_l \bar{u} \wedge \bar{w} \preceq_l (u:\bar{u}) \quad (97)$$

And (96) follows from the definition of \preceq_l . \square

On quite a few occasions we will be doing induction on the height of terms and the following properties of tmd are therefore convenient:

Lemma 4.26. For terms s, t such that $s \neq t$:

$$\text{If } \mathcal{H}(s) = 1 \wedge \mathcal{S}(s) \neq \mathcal{S}(t) \text{ then } \text{tmd}(s, t) = s. \quad (98)$$

Proof. Assume $\mathcal{H}(s)=1$. Then s is on the form $a()$. Take $t = g(u_1, \dots, u_m)$. If $g = a$, $s \preceq t$ and $\text{tmd}(s, t)$ is undefined, else: ⁴⁴

$$\begin{aligned} \text{tmd}(a(), g(u_1, \dots, u_m)) &= a(\text{tmd}_l((), (u_1, \dots, u_m))) && \text{by (tmd)} \\ &= a() && \text{by (tl}_0\text{)}. \end{aligned}$$

\square

⁴⁴Strictly speaking $\text{tmd}(a(), g(u_1, \dots, u_m))$ is undefined according to the definition of tmd (p. 62) due to the restriction R1. However as mentioned above this restriction is not presupposed in the proof of Theorem 4.3.

4 THE TERM DIFFERENCE ALGORITHM

Lemma 4.27. *For terms s, t such that $s \neq t$:*

$$\text{If } \mathcal{H}(t) = 1 \text{ then } \text{tmd}(s, t) = s. \quad (99)$$

Proof. Assume $\mathcal{H}(t)=1$. Then t is on the form $a()$. Take $s = g(u_1, \dots, u_m)$:

$$\begin{aligned} \text{tmd}(g(u_1, \dots, u_m), a()) &= g(\text{tmd}_l((u_1, \dots, u_m), ())) && \text{by (tmd)} \\ &= g(u_1 : \text{tmd}_l((u_2, \dots, u_m), ())) && \text{by (tl}_3) \\ &\vdots \\ &= g(u_1 : (u_2 : (\dots u_n : (\text{tmd}_l((), ())) \dots)) && \text{by (tl}_3) \\ &= g(u_1 : (u_2 : (\dots u_n : () \dots)) && \text{by (tl}_0) \\ &= g(u_1, \dots, u_m). \end{aligned}$$

□

Lemma 4.28 below plays an important role in the proof of *no junk*.

Lemma 4.28. *For arbitrary terms $g(\bar{v})$ and $f(\bar{t})$*

$$g(\bar{v}) \neq f(\bar{t}) \wedge f(\bar{t}) \neq \varepsilon \wedge \bar{t} \neq () \rightarrow f(\bar{t}) \not\leq \text{tmd}(g(\bar{v}), f(\bar{t})) \quad (100)$$

Proof. The proof is by induction on the height of $g(\bar{v})$. Note that if $f \neq g$ the truth of (100) follows by Observation 3.8 since $\mathcal{S}(\text{tmd}(g(\bar{v}), f(\bar{t}))) = g$. Thus we may assume that $f = g$ and prove

$$g(\bar{v}) \neq g(\bar{t}) \wedge g(\bar{t}) \neq \varepsilon \wedge \bar{t} \neq () \rightarrow g(\bar{t}) \not\leq \text{tmd}(g(\bar{v}), g(\bar{t})) \quad (101)$$

Basis: For the basis case ($\mathcal{H}(g(\bar{v}))=1$) then $g(\bar{v}) = h()$. $g(\bar{t}) \neq g(\bar{v})$, $\text{tmd}(g(\bar{v}), g(\bar{t})) = g(\bar{v}) = h()$ and (101) holds since if $g(\bar{t}) \leq h()$ then either $g(\bar{t}) = g(\bar{v}) = h()$ or $g(\bar{t}) = \varepsilon$ but each possibility is negated in the antecedent to (101).

Induction step: Assume as the induction hypothesis that the following holds for any $g(\bar{v})$ such that $\mathcal{H}(g(\bar{v})) \leq n$:

$$g(\bar{t}) \neq \varepsilon \wedge \bar{t} \neq () \rightarrow g(\bar{t}) \not\leq \text{tmd}(g(\bar{v}), g(\bar{t})) \quad (102)$$

And consider the following term:

$$\text{tmd}(g(\bar{v}), g(\bar{t})) \quad \mathcal{H}(g(\bar{v})) = n + 1 \quad (103)$$

Since $\mathcal{H}(g(\bar{v})) = n + 1$, $\bar{v} \neq ()$ and may be written as $u : \bar{u}$, and (103) may be rewritten as

$$g(\text{tmd}_l((u : \bar{u}), (\bar{t}))) \quad (104)$$

In the calculation of $\text{tmd}_l((u:\bar{u}), (\bar{t}))$, the algorithm tmd recurses through the list $u:\bar{u}$. For each u one of the preconditions in the definition of tmd_l (tl₁–tl₃) must hold. Let $\bar{t} = t_1, \dots, t_n$:

- If $\exists i(u = t_i)$, by (tl₁), (104) becomes $g(\text{tmd}_l(\bar{u}, \bar{t}))$. But then u will not be an immediate subterm in the term $g(\text{tmd}_l((u:\bar{u}), (\bar{t})))$ this can be seen from the following argument: By (R2) $\mathcal{S}(u)$ occurs only once in $u:\bar{u}$. And we know that in a term: $\text{tmd}_l((g(\bar{u}), (g(\bar{t})))) = g(\text{tmd}_l(\bar{u}, \bar{t}))$, no immediate subterm will have an outermost function symbol which does not occur as the outermost function symbol for some term in list \bar{u} .⁴⁵ Hence \bar{t} will contain a term $t_i = u$ which is not an immediate subterm of the term $\text{tmd}(g(\bar{v}), g(\bar{t}))$ and from Lemma 4.23 it follows that:

$$g(\bar{t}) \not\leq \text{tmd}(g(\bar{v}), g(\bar{t}))$$

- If $\forall j(u \neq t_j) \wedge \exists i(\mathcal{S}(u) = \mathcal{S}(t_i))$, by (tl₁), (104) becomes

$$g(\text{tmd}(u, t_i)\text{tmd}_l((u:\bar{u}), (\bar{t})))$$

and consequently the term $\text{tmd}(u, t_i)$ becomes an immediate subterm in the term $\text{tmd}(g(\bar{v}), g(\bar{t}))$. From the induction hypothesis it follows that $t_i \not\leq \text{tmd}(u, t_i)$ and thus \bar{t} will contain a term $t' = \text{tmd}(u, t_i)$ which is not an immediate subterm of the term $\text{tmd}(g(\bar{v}), g(\bar{t}))$ and from Lemma 4.23 it follows that:

$$g(\bar{t}) \not\leq \text{tmd}(g(\bar{v}), g(\bar{t}))$$

- If there are no terms in \bar{u} such that $\exists i(u = t_i)$ or $\exists i(\mathcal{S}(u) = \mathcal{S}(t_i))$, then $\text{tmd}(g(\bar{v}), g(\bar{t})) = g(\bar{v})$. And since it is assumed that $\bar{t} \neq ()$, $g(\bar{t})$ must contain at least one immediate subterm which does not occur in $g(\bar{u})$ and consequently $g(\bar{t}) \not\leq g(\bar{v})$. \square

4.4 Proof of left-cut

With the definition of \prec_l and the preliminary results established, we can continue with the proof of Theorem 4.3, starting with Theorem 4.3.1

$$\text{If } u = \text{tmd}(s, t) \text{ then } u \preceq s. \tag{42}$$

Proof. The proof is by induction on the height of t .

Basis: $\mathcal{H}(t) \leq 1$. If $\mathcal{H}(t) = 0$, $\text{tmd}(s, t)$ is undefined. So for the basis we

⁴⁵This can be seen by considering the algorithm tmd .

4 THE TERM DIFFERENCE ALGORITHM

only have to consider a term t such that $\mathcal{H}(t) = 1$. t is then on the form $a()$. We must show that:

$$\text{tmd}(g(u_1, \dots, u_m), a()) \preceq g(u_1, \dots, u_m) \quad (105)$$

The left side reduces to $g(u_1, \dots, u_m)$ by Lemma 4.27 and

$$g(u_1, \dots, u_m) \preceq g(u_1, \dots, u_m)$$

holds by Definition 3.6.

Induction step: Assume as the induction hypothesis that for any terms s, u and t such that $\mathcal{H}(t) \leq k$:

$$\text{tmd}(s, t) \preceq s \quad (\mathcal{H}(t) \leq k) \quad (106)$$

I will show that (106) also holds for any term t where $\mathcal{H}(t) = k + 1$, in other words that:

$$\begin{aligned} \text{tmd}(g(u_1, \dots, u_m), f(t_1, \dots, t_n)) &\preceq g(u_1, \dots, u_m) \\ \mathcal{H}(f(t_1, \dots, t_n)) &= k + 1. \end{aligned} \quad (107)$$

(107) is (by the definition of tmd), equivalent to:

$$g(\text{tmd}_l((u_1, \dots, u_m), (t_1, \dots, t_n))) \preceq g(u_1, \dots, u_m). \quad (108)$$

The rest of the proof can be split into two main parts. If, under the assumption of the induction hypothesis, it can be proved that

$$\text{tmd}_l((u_1, \dots, u_m), (t_1, \dots, t_n)) \preceq_l (u_1, \dots, u_m), \quad (109)$$

and that for arbitrary function symbols g and lists l, r :

$$\text{If } l \preceq_l r \text{ then } g(l) \preceq g(r) \quad (110)$$

Then, taking l to be $\text{tmd}_l((u_1, \dots, u_m), (t_1, \dots, t_n))$ and r to be (u_1, \dots, u_m) , it can be concluded that (108) holds. We know from Lemma 4.21, that (110) is true, so we only need to establish (109). Note first, that given the assumption that $\mathcal{H}(f(t_1, \dots, t_n)) = k + 1$:

$$\forall i (0 < i \leq n) \quad \mathcal{H}(t_i) \leq k. \quad (111)$$

The proof of (109) is by subinduction on the length of (u_1, \dots, u_m) .

Subinduction basis: For the basis ($m = 0$) consider

$$\text{tmd}_l((), r) \preceq_l () \quad (112)$$

The left side of (112) reduces to $(\)$ by (tl_0) , $(\) \preceq_l (\)$ is true by (53) and consequently (112) holds.

Subinduction induction step: Assume as the subinduction induction hypothesis that for arbitrary lists l, r , where $\mathcal{L}(l) = m$:

$$\text{tmd}_l(l, r) \preceq_l l \quad (113)$$

holds and consider

$$\text{tmd}_l((u_1, \dots, u_{m+1}), (t_1, \dots, t_n)) \preceq_l (u_1, \dots, u_{m+1}) \quad (114)$$

For any non-empty list (u_1, \dots, u_{m+1}) , one of the preconditions in the definition of tmd_l (tl_1 – tl_3) must hold. I consider each case in turn:

- For the precondition $\exists i(u_1 = t_i)$, by (tl_1) , the left side of (114) reduces to:

$$\text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)). \quad (115)$$

By the induction hypothesis (113):

$$\text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)) \preceq_l (u_2, \dots, u_{m+1}). \quad (116)$$

By (49) we also have that:

$$(u_2, \dots, u_{m+1}) \prec_l (u_1, \dots, u_{m+1}), \quad (117)$$

(114) follows from (116) and (117) by transitivity of \preceq_l .

- For the precondition $\forall j(u_1 \neq t_j) \wedge \exists i(\mathcal{S}(u_1) = \mathcal{S}(t_i))$, by (tl_2) , (114) reduces to:

$$\text{tmd}(u_1, t_i) : \text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)) \preceq_l (u_1, \dots, u_{m+1}). \quad (118)$$

Since the assumption in (107) was that $\mathcal{H}(f(t_1, \dots, t_n)) = k + 1$, we know that $\mathcal{H}(t_i) \leq k$, and by the first induction hypothesis (106) we have that

$$\text{tmd}(u_1, t_i) \preceq u_1. \quad (119)$$

From (119) and the definition of \prec_l ⁴⁶ it follows that:

$$\begin{array}{l} \text{tmd}(u_1, t_i) : \text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)) \prec_l \\ u_1 : \text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)). \end{array} \quad (120)$$

⁴⁶Taking $\text{tmd}(u_1, t_i)$ to be t in (50).

4 THE TERM DIFFERENCE ALGORITHM

By the subinduction induction hypothesis (113):

$$\text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)) \preceq_l (u_2, \dots, u_{m+1}) \quad (121)$$

And from (121) and Lemma 4.22:

$$u_1 : \text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)) \preceq_l u_1 : (u_2, \dots, u_{m+1}). \quad (122)$$

(118) follows from (120) and (122) by transitivity of \preceq_l .

• For the precondition $\forall i (u_1 \neq t_i \wedge \mathcal{S}(u_1) \neq \mathcal{S}(t_i))$, by (tl₃), (114) reduces to:

$$u_1 : \text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)) \preceq (u_1, \dots, u_{m+1}) \quad \text{by (tl}_3\text{)} \quad (123)$$

By the subinduction induction hypothesis (113):

$$\text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)) \preceq_l (u_2, \dots, u_{m+1}) \quad (124)$$

From (124) and Lemma 4.22:

$$u_1 : \text{tmd}_l((u_2, \dots, u_{m+1}), (t_1, \dots, t_n)) \preceq_l u_1 : (u_2, \dots, u_{m+1}) \quad (125)$$

which is equivalent to (123) and consequently (114) holds. This concludes the induction step in the proof of (109) and hence the proof of Theorem 4.3.1. \square

4.5 Proof of not right-cut

The proposition to prove is:

$$\text{If } u = \text{tmd}(s, t) \text{ then } u \not\preceq t \quad (43)$$

Proof. We know that $s \not\preceq t$ and in particular that $s \neq t$ because u is undefined if $s \preceq t$. We also know that $u \neq \varepsilon$. As tmd is undefined for $s, t = \varepsilon$ we can take $s = g(\bar{u})$ and $t = f(\bar{t})$. A reformulation of (43) accordingly yields the following general proposition to prove:

$$\text{tmd}(g(\bar{u}), f(\bar{t})) \not\preceq f(\bar{t}) \quad (126)$$

This reduces to

$$g(\text{tmd}_l(\bar{u}, \bar{t})) \not\preceq f(\bar{t}) \quad (127)$$

by the definition of tmd . If $f \neq g$, (127) is true by Observation 3.8. We can therefore assume that $f = g$ and prove

$$g(\text{tmd}_l(\bar{u}, \bar{t})) \not\leq g(\bar{t}) \quad (128)$$

The proof of (128) is by induction on the height of $g(\bar{u})$.

Basis: For the basis, $\mathcal{H}(g(\bar{u})) \leq 1$. Since $g(\bar{u}) \neq \varepsilon$ we only have to consider the case that $\mathcal{H}(g(\bar{u})) = 1$. If $\mathcal{H}(g(\bar{u})) = 1$, $\text{tmd}(g(\bar{u}), g(\bar{t}))$ is undefined and (43) is trivially true.

Induction step: Assume as the induction hypothesis that (128) holds for any term w where $\mathcal{H}(w) \leq m$. We must show that (128) holds for any term w' where $\mathcal{H}(w') = m + 1$.

According to Lemma 4.21 it should suffice to show that:

$$\text{tmd}_l(\bar{u}, \bar{t}) \not\leq_l \bar{t} \quad (129)$$

$\bar{u} = ()$ is impossible since $\mathcal{H}(g(\bar{u})) = m + 1$. We may therefore assume that \bar{u} is non-empty. In the calculation of $\text{tmd}_l(\bar{u}, \bar{t})$, the algorithm tmd recurses through the list \bar{u} . For each u one of the preconditions in the definition of tmd_l (tl₁–tl₃) must hold. Let $\bar{t} = (t_1, \dots, t_n)$ and let $\bar{u} = u:l$.

• If for some u the precondition of (tl₃): $\forall i(u \neq t_i \wedge \mathcal{S}(u) \neq \mathcal{S}(t_i))$ holds, then

$$\text{tmd}_l((u:l), (t_1, \dots, t_n)) = u:\text{tmd}_l(l, (t_1, \dots, t_n)) \quad (130)$$

But then we know that the list $\text{tmd}_l((u:l), (t_1, \dots, t_n))$ will contain a term u such that

$$\forall i(1 \leq i \leq n) \rightarrow \mathcal{S}(u) \neq \mathcal{S}(t_i)$$

but then by Observation 3.8 this u cannot be a cut of any t_i :

$$\forall i(1 \leq i \leq n) \rightarrow u \not\leq t_i$$

By Corollary 4.24 it follows that:

$$\text{tmd}_l((u:l), (t_1, \dots, t_n)) \not\leq (t_1, \dots, t_n).$$

• If for some u the precondition of (tl₂): $\forall j(u \neq t_j) \wedge \exists i(\mathcal{S}(u) = \mathcal{S}(t_i))$ holds, then

$$\text{tmd}_l((u:l), (t_1, \dots, t_n)) = \text{tmd}(u, t_i):\text{tmd}_l(l, t_1, \dots, t_n) \quad (131)$$

4 THE TERM DIFFERENCE ALGORITHM

But then we know that the list $\text{tmd}_l((u:l), (t_1, \dots, t_n))$ will contain the term $\text{tmd}(u, t_i)$.⁴⁷ Since $\mathcal{H}(g(\bar{u})) = m + 1$, $\mathcal{H}(u) = m$ and by the induction hypothesis we have

$$\text{tmd}(u, t_i) \not\leq t_i. \quad (132)$$

By the definition of tmd we know that

$$\mathcal{S}(\text{tmd}(u, t_i)) = \mathcal{S}(t_i).$$

Since Restriction 2 holds for $g(t_1, \dots, t_n)$ we also know that the outermost function symbols of any term t' different from t_i is different from $\mathcal{S}(\text{tmd}(u, t_i))$ hence for this i :

$$\forall j \quad j \neq i \rightarrow \mathcal{S}(t_i) \neq \mathcal{S}(t_j). \quad (133)$$

And by Observation 3.8 we therefore also have that:

$$\forall j(1 \leq j \leq n) \quad j \neq i \rightarrow \text{tmd}(u, t_i) \not\leq t_j \quad (134)$$

By (132) and (134), we know that $\text{tmd}(u, t_i)$ cannot be a cut of any term t_j :

$$\forall j(1 \leq j \leq n) \quad \text{tmd}(u, t_i) \not\leq t_j \quad (135)$$

And by Lemma 4.23 it follows that:

$$\text{tmd}_l((u:l), (t_1, \dots, t_n)) \not\leq (t_1, \dots, t_n).$$

• The third possibility is that none of $(\text{tl}_2, \text{tl}_3)$ apply to any u in the list. Take $u:l = (u_1, \dots, u_m)$, we then have that

$$\forall j \exists i(1 \leq j \leq m \wedge 1 \leq i \leq n) \quad u_j = t_i$$

but then $(u_1, \dots, u_m) \preceq_l (t_1, \dots, t_n)$. This can be seen by the following informal argument: Since we know that (t_1, \dots, t_n) contains all the terms in (u_1, \dots, u_m) , (t_1, \dots, t_n) can be made equal to (u_1, \dots, u_m) by removing the terms from (t_1, \dots, t_n) which are not in (u_1, \dots, u_m) . If

$$(u_1, \dots, u_m) \preceq_l (t_1, \dots, t_n)$$

then

$$g(u_1, \dots, u_m) \preceq g(t_1, \dots, t_n)$$

by Lemma 4.21. But we assumed that this was not the case. This concludes the proof of Theorem 4.3.2 \square

⁴⁷The term $\text{tmd}(u, t_i)$ will be the first term in the list no matter what the rest of the list $\text{tmd}_l(l, t_1, \dots, t_n)$ becomes.

4.6 Proof of no junk

The proposition to prove is:

$$\begin{aligned} \text{If } u = \text{tmd}(s, t) \text{ then} & \quad (44) \\ \forall v(v \neq \varepsilon \wedge v \preceq s \wedge v \not\preceq t \rightarrow u \text{ } \text{\textcircled{m}} t \preceq v \text{ } \text{\textcircled{m}} t) & \end{aligned}$$

Proof. By induction on $\mathcal{H}(s)$.

Basis: For the basis, $\mathcal{H}(s) \leq 1$, since tmd is undefined for $s, t = \varepsilon$ we only have to consider the case $\mathcal{H}(s) = 1$. If $\mathcal{H}(s) = 1$ then $s = a$ so we must prove for some v that:

$$v \neq \varepsilon \wedge v \preceq a \wedge (v \not\preceq t \vee \mathcal{H}(v) = 1) \rightarrow \text{tmd}(a, t) \text{ } \text{\textcircled{m}} t \preceq v \text{ } \text{\textcircled{m}} t. \quad (136)$$

From the second conjunct in the antecedent of (136) we know that $v \preceq a$. If $v \preceq a$ then $v = a \vee v = \varepsilon$ and since we know that $v \neq \varepsilon$ we know that $v = a$. By Lemma 4.26 it follows that:

$$\text{tmd}(a, t) = a \quad (137)$$

and the consequent which must be proved becomes:

$$a \text{ } \text{\textcircled{m}} t \preceq a \text{ } \text{\textcircled{m}} t \quad (138)$$

which holds by the definition of \preceq .

Induction step: Take $s = g(\bar{u})$ and assume as the induction hypothesis that *no junk* holds for any $g(\bar{u})$ such that $1 \leq \mathcal{H}(g(\bar{u})) \leq n$. It suffices to prove that the following holds for a term $g(\bar{u})$ where $\mathcal{H}(g(\bar{u})) = n + 1$:

$$\begin{aligned} \forall v \ v \neq \varepsilon \wedge v \preceq g(\bar{u}) \wedge v \not\preceq t & \\ \rightarrow \text{tmd}(g(\bar{u}), t) \text{ } \text{\textcircled{m}} t \preceq v \text{ } \text{\textcircled{m}} t & \quad (139) \end{aligned}$$

Let $t = f(\bar{t})$ in (139), and assume for some v that:

$$v \neq \varepsilon \quad (140)$$

$$v \preceq g(\bar{u}) \quad (141)$$

$$v \not\preceq f(\bar{t}) \quad (142)$$

$$(143)$$

it must be proved that:

$$\text{tmd}(g(\bar{u}), f(\bar{t})) \text{ } \text{\textcircled{m}} f(\bar{t}) \preceq v \text{ } \text{\textcircled{m}} f(\bar{t}) \quad (144)$$

4 THE TERM DIFFERENCE ALGORITHM

By the definition of tmd this is equivalent to:

$$g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} f(\bar{t}) \preceq v \mathbin{\frown} f(\bar{t}). \quad (145)$$

If $f \neq g$ then $g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} f(\bar{t}) = \varepsilon$ and (145) holds since ε is a cut of any term (including ε). Therefore we can assume that $f = g$ and show that:

$$\text{tmd}(g(\bar{u}), g(\bar{t})) \mathbin{\frown} g(\bar{t}) \preceq v \mathbin{\frown} g(\bar{t}) \quad (146)$$

Since $v \preceq g(\bar{u})$ we know (from (22)) that $\mathcal{S}(v) = \mathcal{S}(g(\bar{u})) = g$ so we can take $v = g(\bar{v})$ for some list \bar{v} and we get that the proposition that must be proved is:

$$\text{tmd}(g(\bar{u}), g(\bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\bar{v}) \mathbin{\frown} g(\bar{t}) \quad (147)$$

which is equivalent to:

$$g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\bar{v}) \mathbin{\frown} g(\bar{t}) \quad (148)$$

With $v = g(\bar{v})$ we get that the assumptions are:

$$g(\bar{v}) \neq \varepsilon \quad (149)$$

$$g(\bar{v}) \preceq g(\bar{u}) \quad (150)$$

$$g(\bar{v}) \not\preceq g(\bar{t}) \quad (151)$$

I prove (148) by subinduction on the length of \bar{u} .

Subinduction basis: For the basis case ($\mathcal{L}(\bar{u})=0$). $g(\bar{v}) = g()$ because $g(\bar{v}) \preceq g(\bar{u})$ and $g(\bar{v}) \neq \varepsilon$. Therefore (147) becomes:

$$\text{tmd}(g(), g(\bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g() \mathbin{\frown} g(\bar{t}) \quad \Rightarrow \quad \text{by (tmd)}$$

$$g(\text{tmd}_l((), \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g() \mathbin{\frown} g(\bar{t}) \quad \Rightarrow \quad \text{by (tl}_0)$$

$$g() \mathbin{\frown} g(\bar{t}) \preceq g() \mathbin{\frown} g(\bar{t}).$$

which holds by the definition of $\mathbin{\frown}$. *Subinduction induction step:* For the induction step of the subinduction, assume as the induction hypothesis that (148) holds for arbitrary lists \bar{u} such that $\mathcal{L}(\bar{u}) \leq n$.

It must be proved that the following holds for the list $(u : \bar{u})$, where $\mathcal{L}(\bar{u}) = n$:

$$g(\text{tmd}_l((u : \bar{u}), \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\bar{v}) \mathbin{\frown} g(\bar{t}) \quad (152)$$

Since $(u : \bar{u})$ is non-empty, one of the preconditions in the definition of $\text{tmd}_l(\text{tl}_1\text{--tl}_3)$ must hold:

- If $\exists i(u = t_i)$, (152) reduces to

$$g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\bar{v}) \mathbin{\frown} g(\bar{t})$$

which is the induction hypothesis.

- I deal with the case for the third precondition before the second. For the precondition of (tl_3) : $\forall i(u \neq t_i \wedge \mathcal{S}(u) \neq \mathcal{S}(t_i))$. (152) reduces to:

$$g(u : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\bar{v}) \mathbin{\frown} g(\bar{t}) \quad (153)$$

If it can be proved that

$$g(u : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}). \quad (154)$$

then (153) follows from the induction hypothesis (148) and (154) by transitivity of \preceq .

For a more compact notation in the following let α stand for the list $\text{tmd}_l(\bar{u}, \bar{t})$. Assume for contradiction the negation of (154) namely that there exists a term s' such that:

$$s' = g(u : \alpha) \mathbin{\frown} g(\bar{t}) \quad (155)$$

and

$$s' \not\preceq g(\alpha) \mathbin{\frown} g(\bar{t}) \quad (156)$$

From (155) and the definition of $\mathbin{\frown}$ it follows that:

$$s' \preceq g(u : \alpha) \quad (157)$$

and

$$s' \preceq g(\bar{t}) \quad (158)$$

Either $s' \preceq g(\alpha)$ or $s' \not\preceq g(\alpha)$ must be the case. If $s' \preceq g(\alpha)$ we get a contradiction with (156) right away, as can be seen from the following argument: Assume that $s' \preceq g(\alpha)$, but then since also $s' \preceq g(\bar{t})$ it follows from the last equation in the definition of $\mathbin{\frown}$ (36), that $s' \preceq (g(\alpha) \mathbin{\frown} g(\bar{t}))$ ⁴⁸

⁴⁸Instantiated for this particular case (36) becomes

$$\forall v(v \preceq g(\alpha) \wedge v \preceq g(\bar{t}) \rightarrow v \preceq (g(\alpha) \mathbin{\frown} g(\bar{t})).$$

4 THE TERM DIFFERENCE ALGORITHM

which contradicts the assumption (156). Hence it remains to consider the other possibility, that $s' \not\leq g(\alpha)$. Since $s' \preceq g(u : \alpha)$ we also know (from Observation 3.8) that s' has the same outermost function symbol as $g(u : \alpha)$ and therefore we can take s' to be $g(\bar{w})$ for some list of terms \bar{w} . Hence the assumptions are:

$$g(\bar{w}) \not\leq g(\alpha),^{49} \quad (159)$$

$$g(\bar{w}) \preceq g(u : \alpha)^{50} \quad (160)$$

and

$$g(\bar{w}) \preceq g(\bar{t}).^{51} \quad (161)$$

but then, from (159) and (160), by Lemma 4.25 we get that either the term u or a term u' such that $u' \preceq u$, must occur in the list \bar{w} .

This contradicts (161), which can be seen by the following argument: From the precondition $(\forall i(u \neq t_i \wedge \mathcal{S}(u) \neq \mathcal{S}(t_i)))$ it follows that u does not occur in \bar{t} . And because no term in \bar{t} has the same outermost function symbol as u , no term in \bar{t} can be a cut of u , formally: $\neg \exists i(t_i \preceq u)$ (Cf. Observation 3.8 (21)) so neither u nor any term u' such that $u' \preceq u$ occurs in \bar{t} . When either u or u' occurs in \bar{w} , but none of them occur in \bar{t} , then by Corollary 4.24, $\bar{w} \not\leq_l \bar{t}$.

From $\bar{w} \not\leq_l \bar{t}$ it follows by Lemma 4.21 that $g(\bar{w}) \not\leq g(\bar{t})$ but this contradicts (171). An we may conclude that there is no s' such that

$$s' = g(u : \alpha) \mathbin{\frown} g(\bar{t}) \wedge s' \not\leq g(\alpha) \mathbin{\frown} g(\bar{t}) \quad (162)$$

and consequently that (154) holds. And (153) follows from (148) and (154) by transitivity of \preceq .

• For the precondition of tl_2 : $\forall j(u \neq t_j) \wedge \exists i(\mathcal{S}(u) = \mathcal{S}(t_i))$, (152) reduces to:

$$g(\text{tmd}(u, t_i) : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\bar{v}) \mathbin{\frown} g(\bar{t}) \quad (163)$$

If it can be proved that

$$\frac{g(\text{tmd}(u, t_i) : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t})}{\quad} \quad (164)$$

⁴⁹Which we get by substituting $g(\bar{w})$ for s' in $s' \not\leq g(\alpha)$.

⁵⁰Which we get by substituting $g(\bar{w})$ for s' in (157).

⁵¹Which we get by substituting $g(\bar{w})$ for s' in (158).

then (163) follows from the induction hypothesis (148) and (164) by transitivity of \preceq .

Take $\alpha = \text{tmd}_l(\bar{u}, \bar{t})$ as before, and assume for contradiction the negation of (164) namely that there exists a term s' such that:

$$s' = g(\text{tmd}(u, t_i):\alpha) \mathbin{\frown} g(\bar{t}) \quad (165)$$

and

$$s' \not\preceq g(\alpha) \mathbin{\frown} g(\bar{t}) \quad (166)$$

From (165) and the definition of $\mathbin{\frown}$ it follows that:

$$s' \preceq g(\text{tmd}(u, t_i):\alpha) \quad (167)$$

and

$$s' \preceq g(\bar{t}) \quad (168)$$

Either $s' \preceq g(\alpha)$ or $s' \not\preceq g(\alpha)$ must be the case. If $s' \preceq g(\alpha)$ we get a contradiction with (166) right away by a similar argument as above. Hence it remains to consider the other possibility, that $s' \not\preceq g(\alpha)$. Since $s' \preceq g(\text{tmd}(u, t_i):\alpha)$ we also know (from Observation 3.8) that $\mathcal{S}(s') = g$ and we can take s' to be $g(\bar{w})$ for some list of terms \bar{w} . Hence the assumptions are:

$$g(\bar{w}) \not\preceq g(\alpha),^{52} \quad (169)$$

$$g(\bar{w}) \preceq g(\text{tmd}(u, t_i):\alpha)^{53} \quad (170)$$

and

$$g(\bar{w}) \preceq g(\bar{t}).^{54} \quad (171)$$

But then, from Lemma 4.25 we get that either the term $\text{tmd}(u, t_i)$ or a term u' such that $u' \preceq \text{tmd}(u, t_i)$, must occur in the list \bar{w} . Given that either $\text{tmd}(u, t_i)$ or such a term u' occurs in \bar{w} , if we can prove that none of them occur in \bar{t} , then by Corollary 4.24, $\bar{w} \not\preceq_l \bar{t}$. From $\bar{w} \not\preceq_l \bar{t}$ it follows

⁵²Which we get by substituting $g(\bar{w})$ for s' in $s' \not\preceq g(\alpha)$.

⁵³Which we get by substituting $g(\bar{w})$ for s' in (167).

⁵⁴Which we get by substituting $g(\bar{w})$ for s' in (168).

4 THE TERM DIFFERENCE ALGORITHM

by Lemma 4.21 that $g(\bar{w}) \not\leq g(\bar{t})$ which contradicts (171). And we may conclude that there is no s' such that

$$s' = g(\text{tmd}(u, t_i):\alpha) \mathbin{\frown} g(\bar{t}) \wedge s' \not\leq g(\alpha) \mathbin{\frown} g(\bar{t}) \quad (172)$$

and consequently that (164) holds. And (163) follows from (148) and (164) by transitivity of \preceq .

Therefore the crux of the argument is to prove that neither the term $\text{tmd}(u, t_i)$ nor a term u' such that $u' \preceq \text{tmd}(u, t_i)$ is an element in the list \bar{t} . In other words it must be proved that

$$\neg \exists j (t_j = \text{tmd}(u, t_i)) \quad (173)$$

and

$$\neg \exists j (t_j \preceq \text{tmd}(u, t_i)) \quad (174)$$

Due to Restriction 2, t_i is the only child in $g(\bar{t})$ with the same outermost function symbol as u . Therefore t_i is the only term which might be equal to, or a cut of $\text{tmd}(u, t_i)$.

So it suffices to prove that:

$$t_i \neq \text{tmd}(u, t_i) \quad (175)$$

and

$$t_i \not\leq \text{tmd}(u, t_i)^{55} \quad (176)$$

To see that $t_i \neq \text{tmd}(u, t_i)$, assume for contradiction that $t_i = \text{tmd}(u, t_i)$ but then, since $\text{tmd}(u, t_i) \not\leq t_i$ by *not right-cut* we get that $t_i \not\leq t_i$ which is false.

If we can prove that $t_i \neq \varepsilon$ and that t_i is not on the form $h()$ for some function symbol h , (176) follows directly from Lemma 4.28. That $t_i \neq \varepsilon$ follows from $\mathcal{S}(u) = \mathcal{S}(t_i)$. That t_i is not on the form $h()$ can be seen from the following argument: Assume that $t_i = h()$ but then since $\mathcal{S}(u) = \mathcal{S}(t_i)$ and then $u = h(\bar{v})$ for some \bar{v} . If $\bar{v} = ()$ then $u = t_i$ but this is contrary to the precondition that $\forall i (u \neq t_i)$, so $\bar{v} \neq ()$ but this is prohibited by Restriction 1 which demands that $t_i \sim u$. □

⁵⁵Note that $t_i \not\leq \text{tmd}(u, t_i)$ implies $t_i \neq \text{tmd}(u, t_i)$ but the two cases are treated separately for the sake of the exposition.

4.7 Proof of minimal left-cut

The proposition to prove is:

$$\begin{aligned}
& \text{If } u = \text{tmd}(s, t) \text{ then} \\
& \forall w (w \neq \varepsilon \wedge w \preceq s \wedge w \not\preceq t \wedge \\
& \quad \forall v (v \neq \varepsilon \wedge v \preceq s \wedge v \not\preceq t \rightarrow w \pitchfork t \preceq v \pitchfork t) \rightarrow \\
& \quad w \preceq u)
\end{aligned} \tag{45}$$

Proof. Consider a w such that

$$\begin{aligned}
& w \neq \varepsilon && \text{(i)} \\
& w \preceq s && \text{(ii)} \\
& w \not\preceq t && \text{(iii)} \\
& \forall v (v \neq \varepsilon \wedge v \preceq s \wedge v \not\preceq t \rightarrow w \pitchfork t \preceq v \pitchfork t) && \text{(iv)}
\end{aligned}$$

In this proof we use what we already have proved for $\text{tmd}(s, t)$. Since we know that (i)–(iii) hold for the special case that $v = \text{tmd}(s, t)$ it follows from (iv) that

$$w \pitchfork t \preceq \text{tmd}(s, t) \pitchfork t. \tag{177}$$

Thus if it can be established for some w which satisfies (i)–(iv) that:

$$w \pitchfork t \preceq \text{tmd}(s, t) \pitchfork t \rightarrow w \preceq \text{tmd}(s, t) \tag{178}$$

we get $w \preceq \text{tmd}(s, t)$ from (177) and (178) by PC and we may conclude that (45) holds.⁵⁶ The proof of (178) is by induction on the length of the list of immediate subterms of s .

Basis: For the basis case (i.e. that the list of immediate subterms of s is empty) then $s = a() = a$. By (ii) $w \preceq a$. If $w \preceq a$ then $w = a \vee w = \varepsilon$, and since $w \neq \varepsilon$ by (i), $w = a$. From Lemma 4.26 it follows that $\text{tmd}(a, t) = a$ and (178) becomes

$$a \pitchfork t \preceq a \pitchfork t \rightarrow a \preceq a.$$

⁵⁶Note that since *no junk* holds for $\text{tmd}(s, t)$ and the conditions in the antecedent to *no junk* are satisfied by w , it is also the case that:

$$\text{tmd}(s, t) \pitchfork t \preceq w \pitchfork t$$

and further, from (177) and by antisymmetry of \preceq , that

$$\text{tmd}(s, t) \pitchfork t = w \pitchfork t.$$

4 THE TERM DIFFERENCE ALGORITHM

This holds by the definition of \preceq (16).

Induction step: As the induction hypothesis assume that (178) holds for any $s = g(\bar{u})$ where $\mathcal{L}(\bar{u}) \leq n$. I will prove that (178) holds for $s' = g(u : \bar{u})$. Take $\bar{t} = f(\bar{t})$, we want to prove:

$$w \preceq \text{tmd}(g(u : \bar{u}), f(\bar{t})) \quad (179)$$

from the assumption:

$$w \text{ m } f(\bar{t}) \preceq (\text{tmd}(g(u : \bar{u}), f(\bar{t})) \text{ m } f(\bar{t})). \quad (180)$$

The following proposition is an instance of the induction hypothesis (take $t = f(\bar{t})$ and $s = g(\bar{u})$):

$$w \text{ m } f(\bar{t}) \preceq \text{tmd}(g(\bar{u}), f(\bar{t})) \text{ m } f(\bar{t}) \rightarrow w \preceq \text{tmd}(g(\bar{u}), f(\bar{t})) \quad (181)$$

So if from the assumption (180) we can show that:

$$w \text{ m } f(\bar{t}) \preceq \text{tmd}(g(\bar{u}), f(\bar{t})) \text{ m } f(\bar{t}), \quad (182)$$

then from the induction hypothesis instance (181) and from (182) it follows that

$$w \preceq \text{tmd}(g(\bar{u}), f(\bar{t})). \quad (183)$$

Therefore if it also can be shown that

$$\text{tmd}(g(\bar{u}), f(\bar{t})) \preceq \text{tmd}(g(u : \bar{u}), f(\bar{t})), \quad (184)$$

we may from (183) and (184) conclude that (179) holds by transitivity of \preceq .

So we must prove (182) and (184), I take the latter first. By tmd (184) reduces to

$$g(\text{tmd}_l(\bar{u}, \bar{t})) \preceq g(\text{tmd}_l((u : \bar{u}), \bar{t})). \quad (185)$$

We can see that (185) holds by considering the definition of tmd_l . The formal details have been left out but consider the following argument: If u is equal to some term in t , the right-hand side of (185) becomes equal to the left-hand side (by (tl₁)). If u partially matches some t_i , the list of children on the right-hand side will contain the immediate subterm $u' = \text{tmd}(u, t_i)$ (by (tl₂)). And if none of the above apply the list of children on the right-hand side will contain the immediate subterm $u' = u$ (by (tl₃)). In any case this subterm u' can be replaced with ε to get the term $g(\text{tmd}_l(\bar{u}, \bar{t}))$ and hence by the definition of \preceq (185) is true.

4.7 Proof of minimal left-cut

The difficult part of *no junk* is to prove (182). First note that (182) by tmd reduces to:

$$w \mathbin{\frown} f(\bar{t}) \preceq g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} f(\bar{t}) \quad (186)$$

The assumption (180) was:

$$w \mathbin{\frown} f(\bar{t}) \preceq \text{tmd}(g(u:\bar{u}), f(\bar{t})) \mathbin{\frown} f(\bar{t})$$

For any non-empty list $(u:\bar{u})$, one of the preconditions in the definition of tmd_l (tl_1 – tl_3) must hold and I consider each case in turn:

- If $\exists i(u = t_i)$, the assumption (180) reduces to

$$w \mathbin{\frown} f(\bar{t}) \preceq g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} f(\bar{t})$$

which is (186), the proposition I wanted to prove.

- If $\forall j(u \neq t_j) \wedge \exists i(\mathcal{S}(u) = \mathcal{S}(t_i))$, the assumption (180) reduces to

$$w \mathbin{\frown} f(\bar{t}) \preceq g(\text{tmd}(u, t_i) : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} f(\bar{t}) \quad (187)$$

So if it can be proved that:

$$g(\text{tmd}(u, t_i) : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} f(\bar{t}) \preceq g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} f(\bar{t}) \quad (188)$$

then (186) follows by transitivity of \preceq .

If $f \neq g$ (188) is equivalent to

$$\varepsilon \preceq \varepsilon$$

which is true by the definition of \preceq .

If $f = g$ (188) is equivalent to

$$g(\text{tmd}(u, t_i) : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \preceq g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \quad (189)$$

This is the same equation as (163). The assumptions for (189) are also the same as the assumptions for (163). Therefore (189) can be proved by an argument similar to the one for (163). The details are left out.

- If $\forall i(u \neq t_i) \wedge (\mathcal{S}(u) \neq \mathcal{S}(t_i))$, the assumption (180) reduces to

$$w \mathbin{\frown} f(\bar{t}) \preceq g(u : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) \quad (190)$$

Since u does not occur immediately in \bar{t}

$$g(u : \text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t}) = g(\text{tmd}_l(\bar{u}, \bar{t})) \mathbin{\frown} g(\bar{t})$$

and (190) reduces to (186) which is what we needed to prove. This concludes the proof of *minimal left-cut*. \square

4.8 An alternative algorithm for term difference

As mentioned in Section 2.3 it can be useful with an operator $\|\|^m$ which, instead of returning only the terms, and parts of terms which have changed, returns the new term with the changed parts highlighted.

I will not give a formal definition of the \mathcal{M} operator, I only present the corresponding algorithm tmd^m . The procedure mark, marks the outermost function symbol in a term.

Mark

$$\text{mark}(f(t_1, \dots, t_n)) = \overline{f}(t_1, \dots, t_n)$$

Using mark tmd^m can be defined quite similar to tmd . The difference between tmd and tmd^m is basically that whereas in the calculation of $\text{tmd}(s, t)$ we only keep the subterms of s which are “new”, (or have descendants which are new) and drop the rest. In $\text{tmd}^m(s, t)$ we keep *all* the subterms of s and thus the structure of s , but we mark the “new” subterms with the mark procedure.

Term difference mark tmd^m

Let $s = g(u_1, \dots, u_n), t = f(t_1, \dots, t_n)$ be terms such that:

$$s \not\leq t$$

$$s \sim t \tag{R1}$$

For any subterm $f(t_1, \dots, t_n)$ in s and t the following holds:

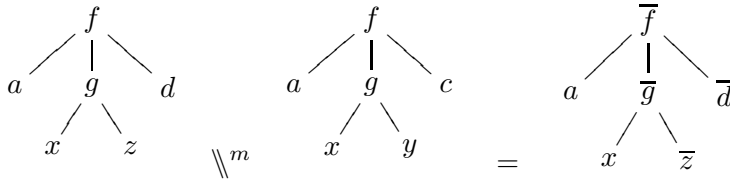
$$\forall i, j \ \mathcal{S}(t_i) = \mathcal{S}(t_j) \Rightarrow i = j \quad 1 \leq i, j \leq n \tag{R2}$$

$$\begin{aligned} \text{tmd}^m(g(u_1, \dots, u_n), f(t_1, \dots, t_n)) = \\ \text{mark}(g(\text{tmd}_l^m((u_1, \dots, u_m), (t_1, \dots, t_n)))) \quad (\text{tmdm}) \end{aligned}$$

The operation tmd_l^m has as its domain lists of terms and as its co-domain lists of terms. It is defined as follows:

Term list difference mark (tmd_l^m)		
$()$		(tlm0)
	if $m = 0$,	
$u_1 : \text{tmd}_l^m((u_2, \dots, u_m), (t_1, \dots, t_n))$		(tlm1)
	if $\exists i(u_1 = t_i)$ and (tlm0) does not apply,	
$\text{tmd}^m(u_1, t_i) : \text{tmd}_l^m((u_2, \dots, u_m), (t_1, \dots, t_n))$		(tlm2)
	if $\exists i(\mathcal{S}(u_1) = \mathcal{S}(t_i))$ and (tlm0,tlm1) does not apply,	
$\text{mark}(u_1) : \text{tmd}_l^m((u_2, \dots, u_m), (t_1, \dots, t_n))$		(tlm3)
	if (tlm0,tlm1,tlm2) does not apply.	

Example 4.29.



4 THE TERM DIFFERENCE ALGORITHM

5 Implementations

5.1 Implementation of utility functions

The Haskell implementation of `height` and `size` are straight forward:

```
height (SElem t []) = 1
height (SElem t elms) = 1 + maxlist(map height elms)
                        where maxlist = foldr max 0
size (SElem t elms) = 1 + sum(map size elms)
```

The Haskell implementation of the basic functions on terms are as follows. For *tmIso*:

```
tmIso (SElem t elms) (SElem t' elms') =
  ((length elms) == (length elms'))
  && (and (map (uncurry tmIso) (zip elms elms')))
```

Root tag equality is easy to implement:⁵⁷

```
tagEq (SElem t elms) (SElem t' elms') = t==t'
```

Function equality is also straight forward:

```
funEq (SElem t elms) (SElem t' elms') =
  t==t' && (length elms) == (length elms')
```

An explicit implementation of term equality (`=`) is not necessary. Haskell is capable of inferring equality for this type on basis of the definition of equality for the constituent types: lists and strings. We only have to make sure that the `SElem` is declared as `deriving(Eq)`.

```
type data SElem = SElem Name [SElem] deriving(Eq,Show)
```

Relative equality is implemented like this:

```
tmEqR n (SElem t elms) (SElem t' elms')
  |n==0 = t==t'
  |n/=0 = t==t' &&
  ((length elms) == (length elms'))
  && (and (map (uncurry (tmEqR (n-1))) (zip elms elms')))
```

Note that this definition is correct with regards to the arity of function symbols as long as we do not adorn the tags with arities as explained above. The alternative definition of term equality in terms of relative equality is:

```
tmEq' t1 t2 = tmEqR (max (height t1) (height t2)) t1 t2
```

⁵⁷We will not make any use of this `tagEq` function however.

5.2 Implementation of term difference

The implementation of `tmd` in Haskell follows below. Note that the equality function which is used to decide whether a term matches another term partially is a parameter to the function `tmDiffP`, thus it is easy to substitute (`tmEqR 0`) with some other function later.

```

tmDiff :: SElem -> SElem -> SElem
tmDiff = tmDiffP (tmEqR 0)

-- pmFun=partialMatchFunction

tmDiffP :: (SElem -> SElem -> Bool) -> SElem -> SElem -> SElem
tmDiffP pmFun t2@(SElem t elms) t1@(SElem t' elms')
  | t2==t1 = error ("The two terms are equal,"
                  ++"diffing them does not make sense.")
  | not (tmIso t2 t1) =
    error ("Conflict with Restriction 1. "
          ++"The two terms are not isomorphic")
  | (hasDup pmFun elms) = error (r2errmsg t2)
  | (hasDup pmFun elms') = error (r2errmsg t1)
  | otherwise = (SElem t (diffCh pmFun elms elms'))
  where r2errmsg tm = ("Conflict with Restriction 2 for"
                      ++nl++sElemToPlainStr(tm))

-- A check for R2: checking if any subterms "match each other partially"
-- (using function f)
hasDup f (x:xs) = topMatchIn f x xs || hasDup f xs
hasDup _ [] = False
topMatchIn f x xs = or (map (f x) xs)

-- Diff on children (the unstarred version)
-- diffCh pmFun listOfNewChildren listOfOldChildren

diffCh pmFun [] ys = []
diffCh pmFun (x:xs) ys
  | fullMatchIn x ys = diffCh pmFun xs ys
  | topMatchIn pmFun x ys =
    (buildDiffTerm pmFun x ys) : (diffCh pmFun xs ys)
  | otherwise = x : (diffCh pmFun xs ys)

-- Check if x is equal to any of the ys.
fullMatchIn x ys = or (map ((==) x) ys)

```

5.3 An alternative implementation of term difference

```

-- Picks out the first x matching in the top with any of the ys.
-- (If it exists)
topMatchOf pmFun x [] =
    error "Function should only be used inside guarded expression"
topMatchOf pmFun x (y:ys)
    | pmFun x y = y
    | otherwise = topMatchOf pmFun x ys

-- Build a new subterm by taking the diff between x
-- and the one matching the top of x in the ys.
-- x is the new term and y:ys is the list of the old ones.
buildDiffTerm pmFun x (y:ys)
    | pmFun x y = tmDiffP pmFun x y
    | otherwise = buildDiffTerm pmFun x ys

```

5.3 An alternative implementation of term difference

The implementation of tmd^m is as follows. A marked term is modelled by an SElem where the tag starts with “_”.

```

tmDiffm :: SElem -> SElem -> SElem
tmDiffm = tmDiffPm (tmEqR 0)

tmDiffPm :: (SElem -> SElem -> Bool) -> SElem -> SElem -> SElem
tmDiffPm pmFun t2@(SElem t elms) t1@(SElem t' elms')
    | t2==t1 = error ("The two terms are equal,"
                    ++"diffing them does not make sense.")
    | not (tmIso t2 t1) =
        error ("Conflict with Restriction 1. "
              ++"The two terms are not isomorphic")
    | (hasDup pmFun elms) = error (r2errmsg t2)
    | (hasDup pmFun elms') = error (r2errmsg t1)
    | otherwise = mark (SElem t (diffChm pmFun elms elms'))
    where r2errmsg tm = ("Conflict with Restriction 2 for"
                        ++nl++sElemToPlainStr(tm))

mark (SElem t elms) = SElem ("_"++t) elms

diffChm pmFun [] ys = []
diffChm pmFun (x:xs) ys
    | fullMatchIn x ys = x : (diffChm pmFun xs ys)
    | topMatchIn pmFun x ys =
        (buildDiffTerm pmFun x ys) : (diffChm pmFun xs ys)
    | otherwise = (mark x) : (diffChm pmFun xs ys)

```

5 IMPLEMENTATIONS

```
buildDiffTermm pmFun x (y:ys)
  | pmFun x y = tmDiffPm pmFun x y
  | otherwise = buildDiffTermm pmFun x ys
```

6 Application to real-life web pages

After having gained some experience with simplified XML, by defining and studying a term model and functions defined for such terms, I will now move to non-simplified XML, (and HTML).⁵⁸ The purpose is to apply the experience from the simplified domain on “real-life” web-pages. The intention is to put the functions defined earlier to a test by applying them. The approach in this section is therefore rather more empirical and experimental than in the previous sections.

6.1 The Haskell type Element

In Section 2 I did two simplifications to the XML specification to be able to work with the basic properties in a formal manner, the simplifications were:

1. To say that a `content` is always an `element`.
2. To not allow attributes in elements.

These two restrictions will now be dropped.

The basic Haskell type used for representing the simplified XML was `SElem`:

```
data SElem = SElem Name [SElem]
type Name = String
```

I will use the `HaXML`-library created by Wallace and Runciman (1999). It contains Haskell types which correspond closely to the XML specification (W3C, 2000a). The basic type is `Element`:

```
data Element = Elem Name [Attribute] [Content]
type Attribute = (Name, AttValue)
data Content = CElem Element
              | CString Bool CharData -- bool is whether whitespace is significant
              | CRef Reference
              | CMisc Misc
```

where the types `Misc` and `Name` are:

```
data Misc = Comment Comment
          | PI ProcessingInstruction
type Name = String
```

⁵⁸Non-simplified XML is in the following simply referred to as XML. I do not treat HTML in particular but the results which applies for XML is easily transferable to HTML with minor modifications.

6 APPLICATION TO REAL-LIFE WEB PAGES

A model corresponding to the term model for simplified-XML could also be given for full XML. And this can be done in a similar manner as it was done for terms. However as the scope of this thesis is limited I will not do a thorough formal investigation of the properties and functions on XML documents, but only present implementations and a discussion related to this implementation. For this purpose a model is not needed. I use the symbol \ll_e to denote an operation for elements corresponding to \ll for terms. `elDiffis` the corresponding Haskell implementation.

There are two main differences between the `SElem` and the `Element` type. The first one is that whereas the children of an `SElem` can only be of the same type (`SElem`), for an `Element` the children is of the type `Content` which might have four different forms:

```
CElem e
CString b cd
CRef r
CMisc m
```

Of these four only the first, in which `e` is of the type `Element` (i.e. has the form: `Elem t as cs`), might have further children. For simplicity I will call such a Content-element for a “CElem content”. I might also say that a `Content` object is on `CElem`-form.

The second main difference is that an `Element` might have attributes. The attributes are represented by a list of key/value pairs, which does not complicate the structure of the data type, but which must be taken into consideration when matching elements against each other.

The fact that the `Element` type is more complex makes the implementations of the functions a bit more untidy, as more special cases must be taken into consideration.

6.2 Utility functions

We can implement utility functions on `Elements` corresponding to the ones for `SElems`. As before I say that an element with no children has height 1, and that the height of an `Element` with children is 1+ the highest of its children. It must now also be decided what the height of the other `Content` types should be. I chose to say that their height is 0, this can also be taken to reflect the fact that these types cannot have children.

The Haskell implementation of `elHeight` (Element height) becomes:

```
elHeight (Elem t as []) = 1
elHeight (Elem t as cs) = 1 + maxlist(map csHeight cs)
      where maxlist = foldl1(max)
```

```

csHeight c =
  case c of
    CElem e -> elHeight e
    CString b cd -> 0
    CRef r -> 0
    CMisc m -> 0

```

The implementation of structural isomorphism for `Elements` corresponds to the implementation for `SElem`:

```

elIso e1@(Elem t as cs) e2@(Elem t' as' cs') =
  ((csLength e1) == (csLength e2))
  && (and (map (uncurry csIso) (zip cs cs'))))

csIso c c' =
  case (c,c') of
    ((CElem e),(CElem e')) -> elIso e e'
    ((CElem e),_) -> False
    (_,(CElem e)) -> False
    (_,_) -> True

```

Earlier I introduced the restriction that two terms had to be isomorphic or else the operator `\|` (and the corresponding algorithm `tmd`) would be undefined. This restriction is too strict for the domain of non-simplified XML. The point of introducing the restriction was to make sure there could be a clear intuition about what it meant to compute the difference between two terms. The point being only that the terms should be, in some sensible way, *comparable*. I originally had the idea therefore to define a similar concept of *comparability* which were supposed to play the same role for `Element` as isomorphy plays for `SElem`.

However it turns out that if this restriction—that the arguments to `\|e` must be comparable in this sense—is enforced, only quite similar documents may be compared. So I chose to drop this restriction in the implementation. Despite that, I will present the concept of comparability below to illustrate the idea.

It is obvious that two `Elements` are comparable if they are isomorphic. However, `Elements` might very well be comparable even if they are not isomorphic. The idea is to treat so-called “leaf”-nodes in a special way. There are two kinds of leaf nodes: `leaf-Element` and `leaf-Content`. An object of type `Element` is a `leaf-Element` if it has no children, or if it has only one child and that child is a `leaf-Content`. An object of type `Content` is a `leaf-Content` if it is on either of the forms `CString b cd`, `CRef r` or `CMisc m` (i.e. it cannot have children) or it is on the form `CElem (Elem t as cs)`,

6 APPLICATION TO REAL-LIFE WEB PAGES

where the list `cs` is empty, or have only one element which is not on the form `(CElem e)` (i.e. it can have at most one descendant.)

The children are at least comparable if they are isomorphic but we shall also say that two `Elements` are comparable if the following holds: Either they are both `leaf-Elements`, or (i) they have the same number of children (i.e. the two lists of `Content` is of equal length) and (ii) the children are pairwise comparable.

Any children of an element are always of type `Content`. If two objects, `c` and `c'` are on the form `CElem e` and `CElem e'` then they are comparable iff `e` and `e'` are comparable. If only one of the two is on the form `CElem e` it is comparable to the other iff `e` is a `leaf-Element`. If neither of `c, c'` are on the form `CElem e` they are comparable. This is a strict version of *comparability* which corresponds quite closely to isomorphism defined earlier. In this sense l_0 - l_2 below are all comparable. A more lax version could allow also l_3 to be comparable to l_0 - l_2 . I.e. we could allow a “leaf” node to be compared to any node which is known not to branch out further down. The rationale for this is that as long as we know that a node does not branch out further down, we know that it might be subtracted from a leaf node or another non-branching node and vice versa. I will use “singular node” as a common denominator for “leaf” nodes and other non-branching nodes.

Example 6.1 (Comparability). *Consider the HTML fragments:*

```

l0: <p>                l1: <p>                l2: <p>
      <ol>              <ol>              <ol>
      <li></li>         <li>Frege</li>         Frege
      <li></li>         <li>Turing</li>       <li>Turing</li>
      <li></li>         <li>Church</li>      <li>Church</li>
      </ol>            </ol>            </ol>
</p>                  </p>                  </p>

l3: <p>                l4: <p>
      <ol>              <ol>
      <li>              <li>Frege</li>
      <p>Frege</p>      <li>Church</li>
      </li>            </ol>
      <li>Turing</li> </p>
      <li>Church</li>
      </ol>
</p>

```

Of these, l_0 - l_2 are comparable in the strict sense, l_0 - l_3 are comparable in

6.2 Utility functions

the slack sense, whereas l_4 is not comparable to any of the others in any of the senses of comparability defined above.

The two notions of comparability may be implemented as follows:

```
elCompStrict e1@(Elem t as cs) e2@(Elem t' as' cs') =
  -- Prevent singular nodes which are not leaves from being comparable
  if (
    ((length cs == 1) && (length cs' == 1)) &&
    (csHeight (head cs) /= csHeight (head cs'))) then False
  else elComp e1 e2

elComp e1@(Elem t as cs) e2@(Elem t' as' cs') =
  if isLeafEl e1 && isLeafEl e2 then True else
    ((length cs) == (length cs'))
    &&
    (and (map (uncurry csComp) (zip cs cs')))
  )

csComp c c' = case (c,c') of
  ((CElem e),(CElem e')) -> elComp e e'
  ((CElem e),x) -> isLeafEl e
  (y,(CElem e)) -> isLeafEl e
  (x,y) -> True

-- Check if a Content is a leaf node.
isLeafCon :: Content -> Bool
isLeafCon (CElem (Elem _ _ [])) = True
isLeafCon (CElem (Elem _ _ [c])) = not (isCElem c)
isLeafCon c = not (isCElem c)

-- Check if an Element is a leaf.
-- An Element is a leaf iff it is a leaf in itself, (i.e. has no children),
-- or if its only child is a Content leaf.
isLeafEl :: Element -> Bool
isLeafEl (Elem _ _ []) = True
isLeafEl (Elem _ _ [c]) = isLeafCon c
isLeafEl (Elem _ _ (c:cs)) = False

isCElem :: Content -> Bool
isCElem (CElem _) = True
isCElem _ = False
```

6.3 A difference operator for Element

When making an implementation of a difference function for “real” XML I start out with the same basic intuitions about the difference operator as before. When calculating $e_2 \parallel_e e_1$ we only want to keep the parts in e_2 which are “new” or which have descendants which are “new”.

So, quite analogous to what was the case with the \parallel operator. The new operator \parallel_e should do the following when comparing two elements e_2 and e_1 : if the new element e_2 differs from the old one e_1 , we build the result ($e_2 \parallel_e e_1$) from e_2 . I.e. we use the tag and attributes of e_2 , and we build a new list of content for this element. As **Content** in $e_2 \parallel_e e_1$ we include

1. those **Content** elements from e_2 which are not in the content list in e_1 .
2. if a **Content** element in e_2 *partially matches* a **Content** element in e_1 we include in the content list in $e_2 \parallel_e e_1$ the result of applying a difference operation on the two **Content** elements.

To be able to do this we must consider several questions.

First: What does it mean that two **Content** elements match each other partially? If two **Content** objects are on a non-**CElem**-form they are either equal or not. There are no grades of equality and the concept of partial match as I use it makes no sense for such objects. For two **CElem**-form **Content** objects there might be different degrees of equality, as these might have a tree structure. And this is quite similar to what was the situation for the simple model. We shall stick to the idea that two elements match each other partially if their outermost tags are equal.⁵⁹

Earlier we had the domain restriction that there could be no repetition of tags in a list of subterms in a term. This restriction was introduced first to simplify the problem matter at hand to be able to get started on the analysis. It later turned out that this restriction was indispensable for some of the proofs of the formal properties of the \parallel operator. However this restriction is too strict if we want to compare XML and HTML documents. We must allow repetition of tags in the **Content**-list in an element (as e.g. in the fragment l_0 above) if the \parallel_e operator is to be interesting.

So I drop this restriction altogether. This of course implies that some of the results from the analysis of \parallel is not applicable to \parallel_e . But we have through the analysis of \parallel become aware of some of the challenges which

⁵⁹For a more complex concept of equality we might also take attributes into consideration.

6.3 A difference operator for *Element*

occur. There are two problems which must be addressed: First what I earlier described as “the problem of choice” (p. 53), i.e. the problem of deciding which of several partial matching elements to choose for comparison. In this initial implementation of \ll_e I adopt the strategy that we pick the first one that matches.

For a smarter implementation this problem of choice needs to be addressed more thoroughly. To do this we would have to introduce criteria for selecting one child before another. These criteria would then become crucial for the behavior of the resulting difference function. One could for example on the one hand select the candidate which is “most equal” or on the other hand the one that is “least equal” and this could lead to quite different behaviour. It might be a context sensitive issue to decide on a criteria.⁶⁰ Which criteria to select might depend on the information we have about the two documents or on the situation on a whole. If it is known that there is only one small difference between two documents one would probably be better off using a criteria of equality.⁶¹

In a more developed implementation one could first employ a concept of “partial match” to select a set of candidates for comparison. And then again choose from this set by applying some further criteria. So the “partial match” operator is a necessary but not sufficient criteria for selection. The \ll_e operator could be modified to take a selection function as a parameter. This selection-function could on its side again of course be a quite complex function. And an advanced implementation might even calculate this function on basis of what results one wants to achieve and what is known about the documents which are to be compared.

The second problem, which I termed “the question of keeping”, is the question of whether in the calculation of $s\ll t$ a subterm in the list of subterms of the “old” term t should be removed after it has been “used” once in the comparison, or whether it should be kept and may be used again. I earlier defined two variants of the algorithm *tmd* (a starred and an unstarred version) (cf. p. 63). In the initial implementation of \ll_e I used the starred version as a point of departure. I.e. adopted the strategy that when a **Content** element from e_1 had been used once in a comparison, it was removed from the list of children in the “old” **Element**. However this did not give the intended result.

So for the final implementation this strategy was somewhat modified.

⁶⁰It could for example depend on what kind of tag the enclosing tag of a given element is

⁶¹The assumption that the documents to compare are quite similar is the basic horizon for the simple implementation in the following.

6 APPLICATION TO REAL-LIFE WEB PAGES

The solution is that an element is dropped after comparison only if it matches the other element completely. If it matches the other element only partially it is kept. This crude strategy would also have to be refined in a developed implementation and how this second problem should be addressed is closely related to how one chooses to deal with the first problem.

6.4 Implementation of Element difference

This is an implementation of a simple variant of \setminus_e which is based on the strategies for addressing the two problems of choice and keeping discussed in the previous section.

```
-- The toplevel elDiff function
elDiff :: Element -> Element -> Element
elDiff = elDiffP (conTagEq)

-- Check for partial match for Content
conTagEq :: Content -> Content -> Bool
conTagEq c c' = case (c,c') of
    ((CElem e),(CElem e')) -> elTagEq e e'
    (_,_) -> False

-- Check for partial match for Element.
elTagEq :: Element -> Element -> Bool
elTagEq (Elem t as cs) (Elem t' as' cs') = t==t'

-- The parametrised elDiff function.
-- pmFun=partialMatchFunction
elDiffP :: (Content -> Content -> Bool) -> Element -> Element -> Element
elDiffP pmFun e2@(Elem t as cs) e1@(Elem t' as' cs')
    | e2==e1 = error ("The two terms are equal,"
                    ++"diffing them does not make sense.")
    | otherwise = (Elem t as (conDiffCh pmFun cs cs'))

conDiffCh pmFun [] cs' = []
-- List of new children is empty , nothing more to compare, so we return.
conDiffCh pmFun (x:xs) ys
    | fullMatchIn x ys = conDiffCh pmFun xs (delete x ys)
    -- A new child x is equal to an old one so we drop it.
    | topMatchIn pmFun x ys =
        (buildDiffElem pmFun x ys) : (conDiffCh pmFun xs ys)
    -- A new child x partially matches an old one
```

6.5 Alternative implementation of Element difference

```
-- so we build a new term recursively.
| otherwise = x : (conDiffCh pmFun xs ys)
-- Otherwise x is neither fully nor partially equal to an old CElem.
-- It must be "new" and we include it in the result.

-- Check if x is equal to any of the ys.
fullMatchIn :: Eq a => a -> [a] -> Bool
fullMatchIn x ys = or (map ((==) x) ys)

-- Check if x partially matches any of the ys with regard to f.
topMatchIn :: (a -> b -> Bool) -> a -> [b] -> Bool
topMatchIn f x xs = or (map (f x) xs)

-- Build a new subterm by calculating the difference between x and
-- the first term y partially matching x in the list of ys.
-- At this point we know that there is such a y.
-- (x is the new term and y:ys is the list of the old ones).
buildDiffElem :: (Content -> Content -> Bool) -> Content -> [Content] -> Content
buildDiffElem pmFun x (y:ys)
  | pmFun x y = conDiffP pmFun x y
  -- We found the y matching. So we build the new subterm.
  | otherwise = buildDiffElem pmFun x ys
  -- We have not found the y yet so keep looking.
  -- We know that pmFun x y holds and that we will find a y eventually.

conDiffP :: (Content -> Content -> Bool) -> Content -> Content -> Content
conDiffP pmFun x y =
  -- We know that pmFun x y holds.
  -- And since pmFun=conTagEq
  -- we know that x and y both are on the form ((CElem e),(CElem e'))
  case (x,y) of
    ((CElem e),(CElem e')) -> CElem (elDiffP pmFun e e')
    (_,_) -> error "This should not happen!!"
```

6.5 Alternative implementation of Element difference

I also implement a version of the algorithm corresponding to tmd^m . In this implementation, instead of only returning the difference between two elements, the structure of the first operand is kept and the differences are highlighted. Highlighting is done by adding an attribute to the changed parts. If the changed element is plain text, I use the attribute `color="red"`. Elements which are not plain text is wrapped in an underline (`<u></u>`)tag. These

6 APPLICATION TO REAL-LIFE WEB PAGES

are just examples of how one could mark changes, they were chosen because they give a decent visual result for HTML documents.

```
-- Mark function (Adds the color="red" attribute to changed tags.
mark c =
  case c of
    (CElem (e)) -> CElem (Elem "U" [] [c])
    (CString b str) -> CElem (Elem "FONT" [mkAtt("color","red")] [c])
    _ -> c

-- The toplevel elDiff mark function
elDiffM :: Element -> Element -> Element
elDiffM = elDiffPm (conTagEq)

elDiffPm :: (Content -> Content -> Bool) -> Element -> Element -> Element
elDiffPm pmFun e2@(Elem t as cs) e1@(Elem t' as' cs')
  | e2==e1 = error ("The two terms are equal,"
    ++"diffing them does not make sense.")
  | otherwise = (Elem t as (conDiffChm pmFun cs cs'))

conDiffChm :: (Content -> Content -> Bool) -> [Content] -> [Content] -> [Content]
conDiffChm pmFun [] cs' = []
conDiffChm pmFun (x:xs) ys
  | fullMatchIn x ys = x : (conDiffChm pmFun xs (delete x ys) )
  | topMatchIn pmFun x ys =
    (buildDiffElemm pmFun x ys) : (conDiffChm pmFun xs ys)
  | otherwise = (mark x) : (conDiffChm pmFun xs ys)

buildDiffElemm :: (Content -> Content -> Bool) -> Content -> [Content] -> Content
buildDiffElemm pmFun x (y:ys)
  | pmFun x y = conDiffPm pmFun x y
  | otherwise = buildDiffElemm pmFun x ys

conDiffPm :: (Content -> Content -> Bool) -> Content -> Content -> Content
conDiffPm pmFun x y =
  case (x,y) of
    ((CElem e),(CElem e')) -> CElem (elDiffPm pmFun e e')
    (_,_) -> error "This should not happen!!"
```

6.6 Examples

I tried the implementation on three instances of the same web page. The web page was the w3.org homepage. It was chosen as an example because it is valid XHTML and therefore quite unproblematic to parse and process. I made a copy of the web page on three different days, 15, 19, and 20 September 2002.⁶² The figures 1–3 (in the appendix) depict the web page at the three different dates. The page is laid out in a table. There is a top, a middle and a bottom section, the middle section of the page contains the bulk of the content, and contains three cells, the first cell with a list of links, the second with a list of news items, and the third contains a search field and some more links.

The change in the page from the 19th to the 20th is that a sentence in the first news item (“Call for Papers: Workshop on Usability and the Web”) has been changed from “Position papers are due 20 September.” to “The extended deadline for position papers is 30 September.”

Evaluation of `e1Diff(w320,w319)` gives the result which can be seen in figure 4. The result is as intended, we get the difference between the two pages.⁶³

The result of evaluating `e1Diff(w320,w315)` is presented in figure 5. As in the previous figure the changed part of the first news item is returned, but also the three new news items which has been added to the page since the 15th. This is also as intended. Note that if we keep the restriction that the operands should be “comparable” in the sense discussed above, `e1Diff(w320,w319)` would be well-defined, but `e1Diff(w320,w315)` would not be, as the two terms `w320` and `w315` are not comparable. However they are fairly similar and it ought to be possible to compare the two documents. For me this indicated that the restriction of comparability was too strong and made me drop it in this implementation.

As expected the structure of `e1Diff(w320,w315)` is quite different from the structure of `w320`. If we use the alternative “mark” variant of the algorithm (`e1DiffM`) the structure of the original term is kept and the changes highlighted. This is shown in figure 6.⁶⁴

⁶²Referred to as `w315,w319` and `w320`.

⁶³In fact we get a little more than the exact difference because the result is the whole element which contains the changed text, and not only the text itself. But this is a reasonable result.

⁶⁴The red color shows up as a lighter shade of grey.

6 APPLICATION TO REAL-LIFE WEB PAGES

7 Conclusion

7.1 Results and assessments

The final result in this thesis was the implementation of an algorithm which can be used to compare XML documents and web pages to identify updates and changes. The intermediate results can be summed up by the following description of the path we have followed towards this final end.

I started out with a simplified model for XML, where XML documents are modelled as terms (or multi-branching trees). The reason that I chose to start with a simplified model was to be able to focus on the principal aspects of comparing tree structured documents. The first result I sought was therefore to define an operator for comparing terms in the simplified model. For that, a proper formal apparatus was necessary so a foundation of basic definitions and functions for terms was established. On this basis the two relations *cut* (\preceq) and *list cut* (\preceq_l), for terms and lists respectively, could be defined. It was proved that *cut* is a partial order relation for terms.

The term difference operator \setminus was defined by identifying four properties it must satisfy. With these properties the result of the operator is a unique term. The properties can be expressed by \preceq and by the auxiliary concept of *term intersection*. An operator \cap was introduced for this concept and used in the definition of \setminus . The term intersection operator was defined and briefly discussed but its properties has not been covered in detail.

The term difference algorithm *tmd* is the next result. The algorithm was formally defined and it was proved that the four above-mentioned properties for \setminus holds for the algorithm.⁶⁵

To alleviate the definition of the term difference operator there were introduced two restrictions on their domains. The first was the restriction that the two terms to compare should be isomorphic. Since this restriction is rather strict and therefore narrows the domain for the operator considerably, I had hoped to be able to eventually get rid of it by modifying the definitions and proofs. I have not managed to do so however⁶⁶ but I believe it is feasible with minor modifications of the definitions and algorithm.

The second restriction was that there should be no repetition of function symbols in the list of children for any subterm.⁶⁷ This is also a rather strict restriction and I discuss at some length the reasons for including it. The

⁶⁵More precisely it is shown that if u is the result of the calculation $\text{tmd}(s, t)$, then u has the four defining properties of \setminus .

⁶⁶We have to assume isomorphy to prove one of the four properties mentioned above.

⁶⁷For example this term $f(g(x), g(y))$ would not be allowed.

7 CONCLUSION

reason for this second restriction hints at some interesting issues which must be addressed in any application intended to be used for comparison of terms.

To illustrate how the results for the simplified model may be applied to proper XML a Haskell function ranging over a type representing XML without simplifications was implemented. It is shown by examples how this function can be applied to some web-pages. There is also a discussion about how the domain restrictions for the simplified model might be handled in the general case. For the implementation for proper XML both restrictions are dropped and *ad hoc* strategies are adopted to face the challenges which then occur. These strategies seem to be successful since the function yields the intended result when applied to some sample web pages.

7.2 Improvements and further work

As is inevitable when the time-frame is limited there will certainly be technical details of proofs and definitions which could be improved. Especially it can be mentioned that we ought to have proved that the algorithms tmd and tmd_l terminate. This can be done by induction on the height of terms and length of lists respectively. It would also have been nice to have an implementation of an algorithm for \mathfrak{m} .

Other more specific issues which could be addressed involve the model and the domain restrictions. As mentioned, the domain restrictions narrow the domain for the operators considerably. In the implementation of a function for proper XML they are dropped and *ad hoc* strategies are employed to face the challenges which was met by the restrictions. It would be interesting to study in general and in more detail what these challenges are and how they may be addressed in the definition and implementation of comparison operators for XML documents. To be able to do that one would probably need a proper formal model for XML without the simplifications which was done for the model in this thesis.

I am confident that such a model can be developed along the same lines as in this work. And I also believe that some of the present results may be translated into such a model. It would however involve quite a few more details in the definitions and proofs.

As mentioned in the preface the original ambitions for this Cand. Scient. project was rather high and included four steps. First to identify criteria for comparing structured documents. Secondly to study and implement basic operations on structured documents. Thirdly to study how the basic operations can be combined into high-level operations. Finally to find out how and whether implementation of high-level operations can be automated.

7.2 *Improvements and further work*

In the present work I have studied different equality criteria for structured documents and I have suggested definitions for two basic operations, (term intersection and term difference). A natural direction for further work could be to define more basic operations which can be implemented as Haskell functions. Hopefully this can be done on the basis of the framework already established here, or in an extended model as discussed above. These basic functions can then be used to define higher order functions or complex filters, by utilising the powerful mechanism of higher-order functions in Haskell. The goal is to accomplish additional tasks à la the ones listed in the preface (p. 9). The final step of automated deduction of such filters or complex functions from examples presupposes that a range of basic and higher order functions are available and also a good and proper understanding of their workings and formal properties. I hope that this thesis has contributed towards such understanding.

7 CONCLUSION

APPENDIX

A Figures

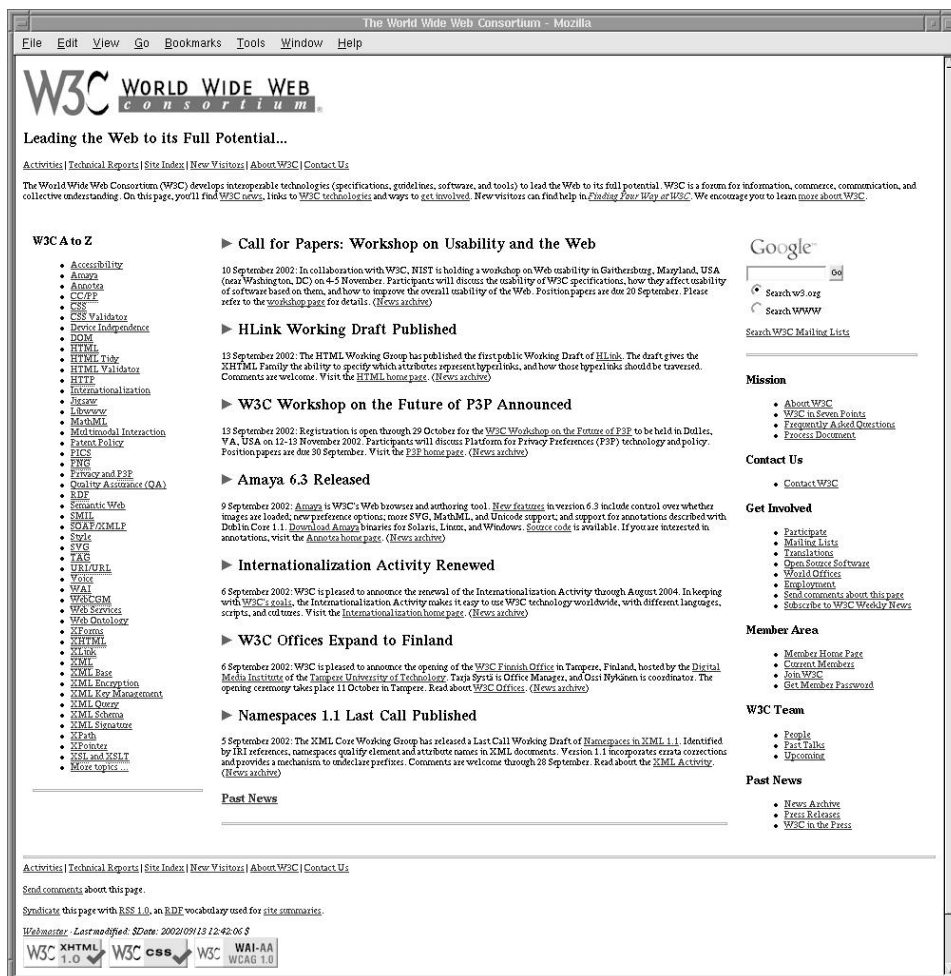


Figure 1: www.w3.org at 15.09.2002 (w315)

A FIGURES

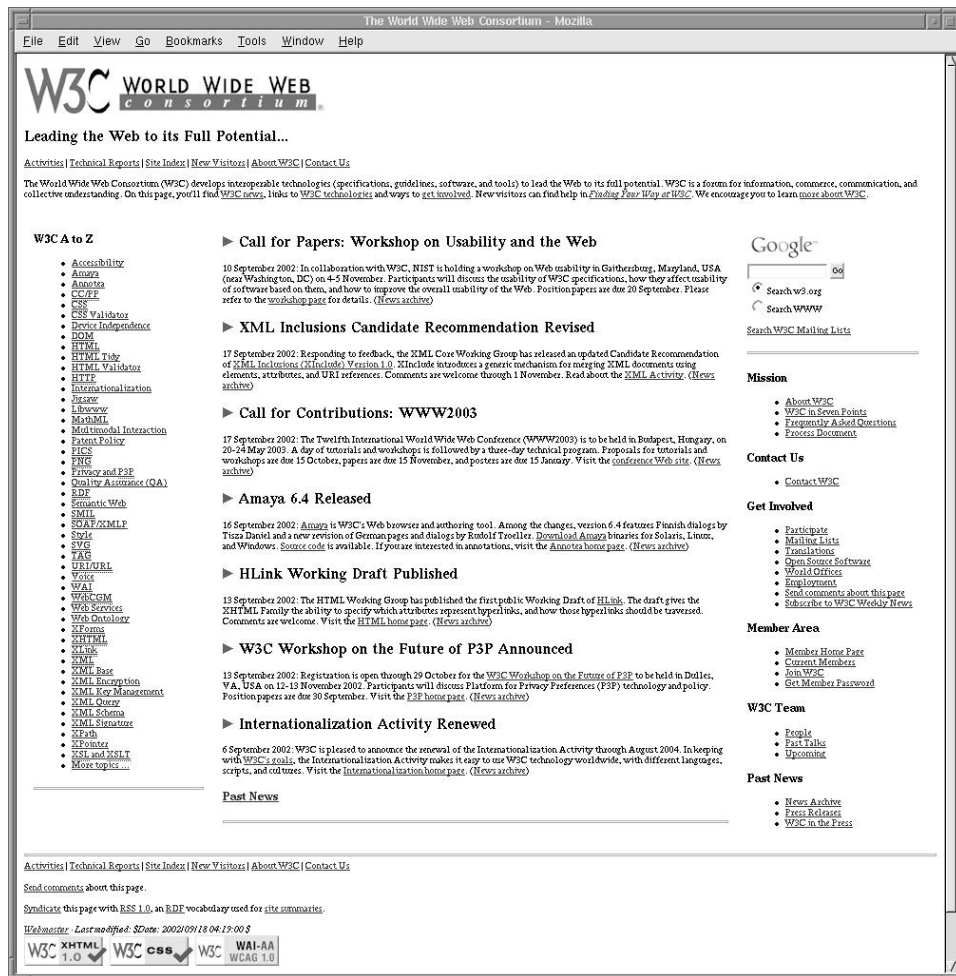


Figure 2: www.w3.org at 19.09.2002 (w319)

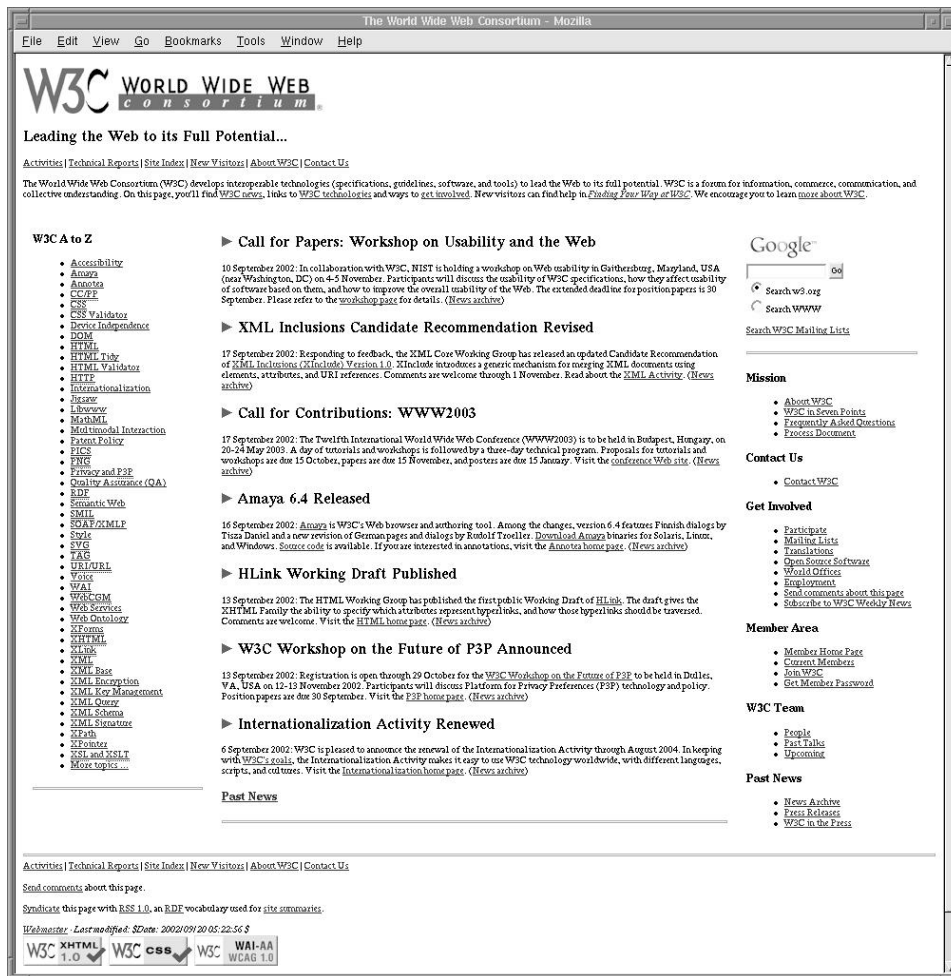


Figure 3: www.w3.org at 20.09.2002 (w320)

A FIGURES

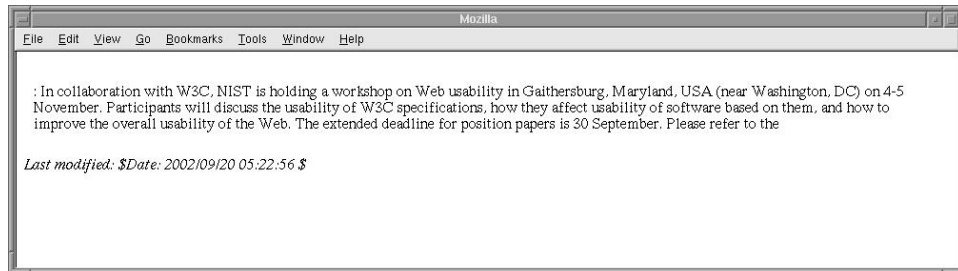


Figure 4: Result of $\text{elDiff}(w320, w319)$

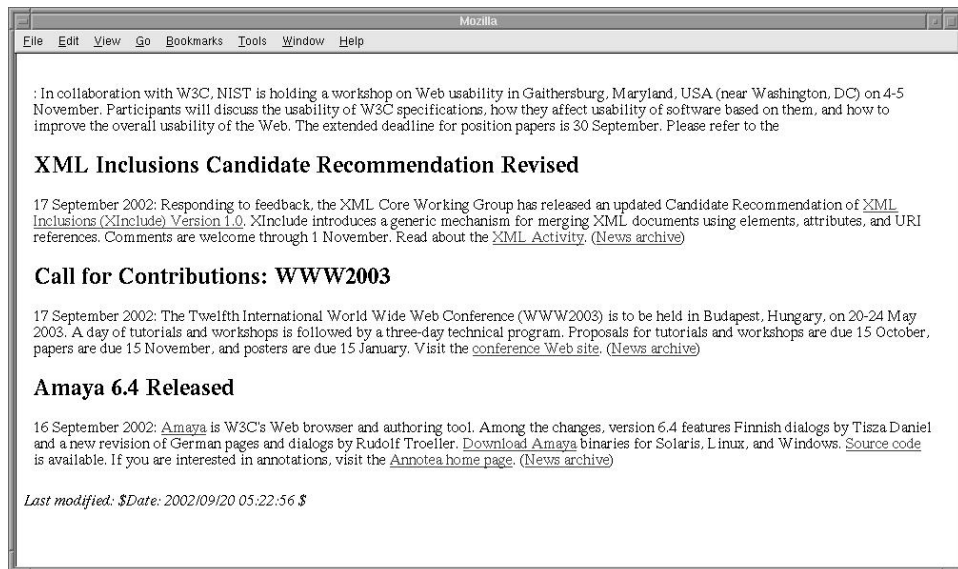


Figure 5: Result of $\text{elDiff}(w320, w315)$



Figure 6: Result of $e1DiffM(w320, w315)$

REFERENCES

References

- Richard Bird. *Introduction to functional programming using Haskell*. Prentice Hall, 2 edition, 1998.
- Alonso Church. *The calculi of Lambda-Conversion*, volume 6 of *Annals of mathematical studies*. Princeton University Press, Princeton NJ, 1941.
- H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1998. URL citeseer.nj.nec.com/comon98tree.html.
- Anthony J. T. Davie. *An Introduction to functional programming systems using Haskell*. Cambridge University Press, 1992.
- Ferenc Gécseg and Magnus Steinby. Tree languages. In *Handbook of formal languages, vol. 3: beyond words*, pages 1–68. Springer-Verlag New York, Inc., 1997. ISBN 3-540-60649-1.
- C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.
- John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.
- Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1998. URL citeseer.nj.nec.com/article/hutton93monadic.html.
- Simon Peyton Jones, John Hughes, et al. Haskell 98—a non-strict, purely functional language. (the haskell 98 report), February 1999a. <http://www.haskell.org/onlinereport/>.
- Simon Peyton Jones, John Hughes, et al. Standard libraries for the haskell 98 programming language. (the haskell 98 library report), February 1999b. <http://www.haskell.org/onlinelibrary/>.
- Peter Lee, editor. *1999 ACM SIGPLAN International Conference on Functional Programming*, 1999. ACM Press.
- Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, New-Jersey, 2 edition, 1998.

REFERENCES

- Peter Csaba Ølveczky. Formal modeling and analysis of distributed systems in Maude, 2003. University of Oslo, Dept of Informatics.
- W.V.O. Quine. *Word and Object*. MIT Press, Cambridge, Massachusetts, 1960.
- Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, Boston, 1997.
- Peter Thiemann. Modeling HTML in Haskell. In *Practical Aspects of Declarative Languages*, pages 263–277, 2000. URL citeseer.nj.nec.com/273347.html.
- Peter Thiemann. A typed representation for HTML and XML documents in Haskell. February 2001.
- W3C. *HTML 4.01 Specification*. W3C (World Wide Web Consortium), 1999. W3C Recommendation 24 December 1999. <http://www.w3.org/TR/1999/REC-html401-19991224/>.
- W3C. *Extensible Markup Language (XML) 1.0*. W3C (World Wide Web Consortium), 2 edition, 2000a. W3C Recommendation 6 October 2000. <http://www.w3.org/TR/2000/REC-xml-20001006>.
- W3C. *XHTML[tm] 1.0: The Extensible HyperText Markup Language//A Reformulation of HTML 4 in XML 1.0*. W3C (World Wide Web Consortium), 2000b. W3C Recommendation 26 January 2000. <http://www.w3.org/TR/2000/REC-xhtml1-20000126>.
- Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In Lee (1999), pages 148–159.