The Definition of the NSP Type System

Technical Report B-02-11

Dirk Draheim, Elfriede Fehr, and Gerald Weber
Institute of Computer Science
Free University Berlin
email: {draheim,fehr,weber}@inf.fu-berlin.de

March 2002

Abstract

The static semantics of a new typed server pages approach is defined as an algorithmic, equi-recursive type system with respect to an amalgamation with a minimal imperative programming language and a collection of sufficiently complex programming language types.

Contents

1 Introduction

In [12] a strongly typed server pages technology NSP (Next Server Pages) has been proposed. In this technical report the results of NSP are formalized. Server pages technologies are widely used in the implementation of ultra-thin client applications. Unfortunately the low-level CGI programming model shines through in these technologies, especially user data is gathered in a completely untyped manner. In [12] a stronlgy typed server pages technology has designed from scratch. The contributions target stability and reusability of server pages based systems. The findings are programming language independent. Some of the most important contributions of NSP has been the following: parameterized server pages, support for complex types in writing forms, statically ensured client page type and description safety. A server page possesses a specified signature that consists of formal parameters, which are native typed with respect to a type system of a high-level programming language. New structured tags are offered for gathering arrays and objects of user defined types. The type correct interplay of dynamically generated forms and targeted server pages is checked at compile-time. It is checked at compile-time if generated page descriptions always are valid with respect to a defined client page description language.

Furthermore NSP is enabling technology for improved web-based application architecture and design. In NSP a server-side call to a server page is designed as a parameter-passing procedure call, too. This enables functional decomposition of server pages and therefore helps decoupling architectural issues and implementing design patterns. In this paper two further concepts are introduced implicitly as a result of the formal definition of NSP concepts: higher order server pages and the exchange of objects across the web user agent. Server pages may be actual form parameters. Server side programmed objects may be actual form parameters and therefore passed to client pages and back, either as messages or virtually as objects.

NSP provides both client page description checking and client page type checking. From the viewpoint of an NSP type system the generated client pages are the actual code, which has to be considered. The generated code is naturally not available at deployment time, therefore NSP defines guidelines and rules for writing the server pages which are non prohibitive and convenient. All reasonable applications of scripting are still allowed. At the same time the coding guidelines and rules target the NSP developer. They are natural and easy to understand. The coding guidelines and rules provide the informal definition of the NSP type system.

The NSP concepts are programming language independent results. They are reusable. They must be amalgamated with a concrete programming language. For every such amalgamation a concrete non-trivial language mapping must be provided. The NSP concepts are designed in a way that concrete amalgamations are conservative with respect to the programming language. That is the semantics of the programming language and especially its type system remain unchanged in the resulting technology. In [12] the NSP concepts are explained through a concrete amalgamation with the programming language Java. As a result of conservative amalgamation the NSP approach does not restrict the potentials of JSP in any way, for example its state handling facility, the Servlet API session concept, is available as a matter of course. Formal semantics of an NSP core type system is given with respect to an amalgamation with a minimal imperative programming language.

This technical report formalizes the type system of Core NSP, which is the amalgamation of NSP concepts with a minimal imperative programming language similar to WHILE [13], which encompasses assignments, command sequences, a conditional control structure and an unbounded loop. The tag set of Core NSP consists of the most important elements for writing forms as well as some nestable text layout tags. The programming language types of Core NSP comprise records, arrays, and recursive types for modeling all the complexity found in the type system of a modern high-level programming languages. The web signatures of Core NSP embrace server page type parameters, that is higher order server pages are modeled. Tags for server side calls to server pages belong to the language, that is functional server page decomposition is modeled. A Per Martin-Löf [27] style type system is given to specify type correctness.

2 Core NSP Grammar

An abstract syntax of Core NSP programs is specified by a context free grammar. Nonterminals are underlined. Terminals are not emphasized. This is contrariwise to BNF standards like [20] or [11], however it fosters readability significantly. Every nonterminal corresponds to a syntactic category. In the grammar a syntactic category is depicted in bold face. A Core NSP program is a whole closed system of several server pages. A page is a parameterized core document and may be a complete web server page or an include server page:

$$\begin{array}{rcl}
\underline{\texttt{system}} & \texttt{::=} & \underline{\texttt{page}} \mid \underline{\texttt{system}}\ \underline{\texttt{system}} \\
\underline{\texttt{page}} & \texttt{::=} & \texttt{<nsp name="}\underline{\texttt{id}}\texttt{">}\ \underline{\texttt{websig-core}}\ \texttt{</nsp>} \\
\underline{\texttt{websig-core}} & \texttt{::=} & \underline{\texttt{param}}\ \underline{\texttt{websig-core}} \mid \underline{\texttt{webcall}} \mid \underline{\texttt{include}} \\
\underline{\texttt{param}} & \texttt{::=} & \texttt{<param name="}\underline{\texttt{id}}\texttt{" type="}\underline{\texttt{parameter-type}}\texttt{"/>} \\
\underline{\texttt{webcall}} & \texttt{::=} & \texttt{<html>}\ \underline{\texttt{head}}\ \underline{\texttt{body}}\ \texttt{</html>} \\
\underline{\texttt{head}} & \texttt{::=} & \texttt{<head><title>}\ \underline{\texttt{strings}}\ \texttt{</title></head>} \\
\underline{\texttt{strings}} & \texttt{::=} & \varepsilon \mid \underline{\texttt{string}}\ \underline{\texttt{strings}} \\
\underline{\texttt{body}} & \texttt{::=} & \texttt{<body>}\ \underline{\texttt{dynamic}}\ \texttt{</body>} \\
\underline{\texttt{include}} & \texttt{::=} & \texttt{<include>}\ \underline{\texttt{dynamic}}\ \texttt{</include>}
\end{array}$$

There are some basic syntactic categories. The category id is a set of labels. The category string consists of character strings. A character string does not contain white spaces. We work with abstract syntax and therefore don't have to deal with white space handling problems [5]. The category parameter-type consists of the possible formal parameter types, i.e. programming language types plus page types. The category supported-type contains each type for which a direct manipulation input capability exists. The respective Core NSP types are specified in section 4.

$$\begin{array}{rcl}
\underline{\texttt{string}} & \texttt{::=} & \mathrm{s} \in \mathbf{String} \\
\underline{\texttt{id}} & \texttt{::=} & \mathrm{l} \in \mathbf{Label} \\
\underline{\texttt{parameter-type}} & \texttt{::=} & \mathrm{t} \in \mathbb{T} \cup \mathbb{P} \\
\underline{\texttt{supported-type}} & \texttt{::=} & \mathrm{t} \in \mathbb{B}_{supported}
\end{array}$$

Parameterized server pages are based on a dynamic markup language, which combines static client page description parts with active code parts. The static parts encompass lists, tables, server side calls, and forms with direct input capabilities, namely check boxes, select lists, and hidden parameters together with the object element for record construction.

$$\begin{array}{rcll}
\underline{\texttt{dynamic}} & \texttt{::=} & & \underline{\texttt{dynamic}}\ \underline{\texttt{dynamic}} \\
 & & \mid & \varepsilon \mid \underline{\texttt{string}} \\
 & & \mid & \underline{\texttt{ul}} \mid \underline{\texttt{li}} \\
 & & \mid & \underline{\texttt{table}} \mid \underline{\texttt{tr}} \mid \underline{\texttt{td}} \\
 & & \mid & \underline{\texttt{call}} \\
 & & \mid & \underline{\texttt{form}} \mid \underline{\texttt{object}} \mid \underline{\texttt{hidden}} \mid \underline{\texttt{submit}} \\
 & & \mid & \underline{\texttt{input}} \mid \underline{\texttt{checkbox}} \\
 & & \mid & \underline{\texttt{select}} \mid \underline{\texttt{option}} \\
 & & \mid & \underline{\texttt{expression}} \\
 & & \mid & \underline{\texttt{code}}
\end{array}$$

Core NSP comprises list and table structures for document layout. All the XML elements of the dynamic markup language are direct subcategories of the category dynamic, which means that the grammar does not constrain arbitrary nesting of these elements. Instead of that the manner of use of a document fragment is maintained by the type systems. We delve on this in section 3.

$$\begin{array}{rcl}
\underline{\texttt{ul}} & \texttt{::=} & \texttt{<ul>}\ \underline{\texttt{dynamic}}\ \texttt{</ul>} \\
\underline{\texttt{li}} & \texttt{::=} & \texttt{<li>}\ \underline{\texttt{dynamic}}\ \texttt{</li>} \\
\underline{\texttt{table}} & \texttt{::=} & \texttt{<table>}\ \underline{\texttt{dynamic}}\ \texttt{</table>} \\
\underline{\texttt{tr}} & \texttt{::=} & \texttt{<tr>}\ \underline{\texttt{dynamic}}\ \texttt{</tr>} \\
\underline{\texttt{td}} & \texttt{::=} & \texttt{<td>}\ \underline{\texttt{dynamic}}\ \texttt{</td>}
\end{array}$$

The rest of the static language parts address server side page calls, client side page calls and user interaction. A call may contain actual parameters only. The call element may contain no element, too. As a matter of taste the special sign $\varepsilon_{\mathtt{act}}$ for empty contents is used in the Core NSP grammar to avoid redundant production and typing rules for the call element.

```
        call  ::=  <call callee="id"> actualparams </call>
actualparams  ::=  $\varepsilon_{\mathtt{act}}$ | actualparam actualparams
 actualparam  ::=  <actualparam param="id"> expr </actualparam>
        form  ::=  <form callee="id"> dynamic </form>
      object  ::=  <object param="id"> dynamic </object>
      hidden  ::=  <hidden param="id"> expr </hidden>
      submit  ::=  <submit/>
       input  ::=  <input type="supported-type" param="id"/>
    checkbox  ::=  <checkbox param="id"/>
      select  ::=  <select param="id"> dynamic </select>
      option  ::=  <option>
                       <value> expr </value>
                       <label> expr </label>
                   </option>
```

Core NSP comprises expression tags for direct writing to the output and code tags in order to express the integration of active code parts with layout. The possibility to integrate layout code into active parts is needed. It is given by reversing the code tags. This way all Core NSP programs can be easily related to a convenient concrete syntax.

```
expression  ::=  <expression> expr </expression>
      code  ::=  <code> com </code>
       com  ::=  </code> dynamic <code>
```

The imperative sublanguage of Core NSP comprises statements, command sequences, an if-then-else construct and a while loop.

```
com  ::=      stat
          |   com ; com
          |   if expr then com else com
          |   while expr do com
```

The only statement is assignment. Expressions are just variable values or deconstructions of complex variable values, i.e. arrays or user defined typed objects.

```
stat  ::=  id := expr
expr  ::=  id  | expr.id | expr[expr]
```

Core NSP is not a working programming language. It posseses only a set of most interesting features to model all the complexity of NSP technologies. Instead Core NSP aims to specify the typed interplay of server pages, the interplay of static and active server page parts and the non-trivial interplay of the several complex types, i.e. user defined types and arrays, which arise during dynamically generating user interface descriptions.

3 Core NSP Type System Strength

The grammar given in 2 does not prevent arbitrary nestings of the several Core NSP dynamic tag elements. Instead necessary constraints on nesting are guaranteed by the type system. Therefore the type of a server page fragment comprises information about the manner of use of itself as part of an encompassing document.

As a result some context free properties are dealt with in static semantics. There are pragmatic reasons for this. Consider an obvious examples first. In HTML forms must not contain other forms. Furthermore some elements like the ones for input capabilities may only occur inside a form. If one wants to take such constraints into account in a context free grammar, one must create a nonterminal for document fragments inside forms and duplicate and appropriately modify all the relevant production rules found so far. If there exist several such constraints the resulting grammar would quickly become unmaintainable. For that reason the Standard Generalized Markup Language supports the notions of exclusion and inclusion exception. The declaration of the HTML form element in the HTML 2.0 SGML DTD [7] is the following:

```
<!ELEMENT FORM - - %body.content -(FORM) +(INPUT|SELECT|TEXTAREA)>
```

The expression `-(FORM)` uses exclusion exception notation and the expression `+(INPUT|SELECT|TEXTAREA)` uses inclusion exception notation exactly for establishing the mentioned constraints. Indeed the SGML exception notation does not add to the expressive power of SGML [40], because an SGML expression that includes exceptions can be translated into an extended context free grammar [21]. The transformation algorithm given in [21] produces $2^{2^{|N|}}$ nonterminals in the worst case. This shows: if one does not have the exception notation at hand then one needs another way to manage complexity. The Core NSP way is to integrate necessary information into types.

Furthermore in NSP the syntax of the static parts is orthogonal to the syntax of the active parts, nevertheless both syntactic structures must regard each other. For example HTML or XHTML lists must not contain other elements than list items. The corresponding SGML DTD [19] and XHTML DTD [38] specifications are:

```
<!ELEMENT (OL|UL) - - (LI)+> resp.  <!ELEMENT ul (li)+>
```

In Core NSP the document fragment in listing 1 is considered correct.

Listing 1

```
01 <ul>
02   <code> x:=3; </code>
03   <li>First list item</li>
04   <code>
05     if condition then </code>
07       <li>Second list item</li> <code>
08     else </code>
09       <li>Second list item</li> <code>
11   </code>
12 </ul>
```

Line 2 must be ignored with respect to the correct list structure, furthermore it must be recognized that the code in lines 4 to 11 corretly provides a list item. Again excluding wrong documents already by abstract syntax amounts to duplicate production rules for the static parts that may be contained in dynamic parts.

A Core NSP type checker has to verify uniquely naming of server pages in a complete system, which is a context dependent property. It has to check whether include pages provide correct elements. The way Core NSP treats dynamic fragment types fits seamlessly to these tasks.

4 Core NSP Types

In this section the types of Core NSP and the subtype relation between types are introduced simultanously. There are types for modeling programming language types, and special types for server pages and server page fragments in order to formalize the NSP coding guidelines and rules. The Core NSP types are given by a family of recursively defined type sets. Some of the Z mathematical toolkit notation [34] is used. Every type represents an infinite labeled regular tree.

The subtype relation formalizes the relationship of actual client page parameters and formal server page parameters by strictly applying the Barbara Liskov principle [22]. A type A is subtype of another type B if every actual parameter of type A may be used in server page contexts requiring elements of type B. The subtype relation is defined as the greatest fix point of a generating function. The generating function is presented by a set of convenient judgment rules for deriving judgments of the form $\vdash\ S < T$.

4.1 Programming Language Types

In order to model the complexity of current high-level programming language type systems, the Core NSP types comprise basic types $\mathbb{B}_{primitive}$ and $\mathbb{B}_{supported}$, array types $\mathbb{A}$, record types $\mathbb{R}$, and recursive types $\mathbb{Y}$. $\mathbb{B}_{primitive}$ models types, for which no null object is provided automatically on submit. $\mathbb{B}_{supported}$ models types, for which a direct manipulation input capability exists. Note that $\mathbb{B}_{primitive}$ and $\mathbb{B}_{supported}$ overlap because of the int type. The set of all basic types $\mathbb{B}$ is made of the union of $\mathbb{B}_{primitive}$ and $\mathbb{B}_{supported}$. Record types and recursive types play the role of user defined form message types. The recursive types allow for modeling cyclic user defined data types. Thereby Core NSP works solely with structural type equivalence [8], i.e. there is no concept of introducing named user defined types, which would not contribute to the understanding of NSP concepts. The types introduced so far and the type variables $\mathbb{V}$ together form the set of programming language types $\mathbb{T}$.

$$\mathbb{T} = \mathbb{B} \cup \mathbb{V} \cup \mathbb{A} \cup \mathbb{R} \cup \mathbb{Y}$$

$$\begin{array}{lcl}
\mathbb{B} & = & \mathbb{B}_{primitive} \cup \mathbb{B}_{supported} \\
\mathbb{B}_{primitive} & = & \{\mathtt{int}, \mathtt{float}, \mathtt{boolean}\} \\
\mathbb{B}_{supported} & = & \{\mathtt{int}, \mathtt{Integer}, \mathtt{String}\} \\
\mathbb{V} & = & \{X, Y, Z, \ldots\} \\
 & \cup & \{\mathtt{Person}, \mathtt{Customer}, \mathtt{Article}, \ldots\}
\end{array}$$

Type variables may be bound by the recursive type constructor μ. Overall free type variables, that is type variables free in an entire Core NSP system resp. complete Core NSP program, represent opaque object reference types. Similarly in Core NSP example programs free term variables are used to model basic constant data values.

For every programming language type, there is an array type. According to subtyping rule 1 every type is subtype of its immediate array type. In commonly typed programming languages it is not possible to use a value as an array of the value's type. But the Core NSP subtype relation formalizes the relationship between actual client page and formal server page parameters. It is used in the NSP typing rules very targeted to constrain data submission. A single value may be used as an array if it is submitted to a server page.

In due course we informally distinguish between establishing subtyping rules and preserving subtyping rules. The establishing subtyping rules introduce initial NSP specific subtypings. The preserving subtyping rules are just the common judgments that deal with defining the effects of the various type constructors on the subtype relation. Judgment rule 2 is the preserving subtyping rule for array types.

$$\mathbb{A} = \{\ \mathtt{array\ of}\ T \mid T \in \mathbb{T} \setminus \mathbb{A}\}$$

$$\frac{}{\vdash T < \texttt{array of } T} \tag{1}$$

$$\frac{\vdash S < T}{\vdash \texttt{array of } S < \texttt{array of } T} \tag{2}$$

A record is a finite collection of uniquely labeled data items, its fields. A record type is a finite collection of uniquely labeled types. In [30] record types are deliberately introduced as purely syntactical and therefore ordered entities. Then permutation rules are introduced that allow record types to be equal up to ordering. In other texts, like e.g. [2] or [31], record types are considered unordered from the beginning. We take the latter approach: a record type is a function from a finite set of labels to the set of programming language types. The usage of some Z Notation [41][18] will ease writing type operator definitions and typing rules later on.

$$\mathbb{R} = (\mathbb{F}\mathbf{Label}) \longrightarrow \mathbb{T}$$

$$\frac{T_j \notin \mathbb{B}_{primitive} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{3}$$

$$\frac{\vdash S_1 < T_1 \ldots \vdash S_n < T_n}{\vdash \{l_i \mapsto S_i\}^{i \in 1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{4}$$

Rule 4 is just the necessary preserving subtyping rule for records.

The establishing subtyping rule 3 states that a shorter record type is subtype of a longer record type, provided the types are equal with respect to labeled type variables. At a first site this contradicts the well-known rules for subtyping records [6] or objects [1]. But there is no contradiction, because these rules describe hierarchies of feature support

and we just specify another phenomenon: rule 3 models that an actual record parameter is automatically filled with null objects for the fields of non-primitive types that are not provided by the actual parameter, but expected by the formal parameter.

The Core NSP type system encompasses recursive types for modeling the complexity of cyclic user defined data types.

$$\mathbb{Y} = \{ \, \mu X \, . \, R \mid X \in \mathbf{V} \, , \, R \in \mathbb{R} \, \}$$

$$\frac{\vdash S[{}^{\mu X.S}/_{X}] < T}{\vdash \mu X.S < T} \tag{5}$$

$$\frac{\vdash S < T[{}^{\mu X.T}/_{X}]}{\vdash S < \mu X.T} \tag{6}$$

Recursive types may be handled in an iso-recursive or an equi-recursive way. The terms iso-recursive or an equi-recursive stem from [10]. In an iso-recursive type system, a recursive type is considered isomorphic to its one-step unfolding and a family of unfold and fold operations on the term level is provided in order to represent the type isomorphisms. A prominent example of this purely syntactical approach is [2]. In an equi-recursive type system like the one given in [3], two recursive types are considered equal if they have the same infinite unfolding. We have chosen to follow the equi-recursive approach along the lines of [14] for two reasons. First it keeps the Core NSP language natural, no explicit fold and unfolding is needed. More importantly, though the theory of an equi-recursive treatment is challenging, it is well-understood and some crucial results concerning proof-techniques and type checking of recursive typing and recursive subtyping are elaborated in an equi-recursive setting.

The subtype relation adequately formalizes all the advanced NSP notions like form message types and higher order server pages.

In the Core NSP type system types represent finite trees or possibly infinite regular trees. A regular tree [9] is a possibly infinite tree that has a finite set of distinct subtrees only. More precisely these type trees are unordered, labeled, and finitely branching. Type equivalence is not explicitly defined, it is given implicitly by the subtype relation: the subtype relation is not a partial order but a pre-order and two types are equal if they are mutual subtypes. The subtype relation is defined in this section as the greatest fixpoint of a monotone generating function on the universe of type trees [14]. The Core NSP subtyping rules provide an intuitive description of this generating function. Thereby the subtyping rules for left folding 5 and right folding 6 provide the desired recursive subtyping.

Beyond this only one further subtyping rule is needed, namely the rule 7 for introducing reflexivity. No explicit introduction of transitivity must be provided as in iso-recursive type systems, because this property already follows from the definition of the subtype relation as greatest fixed point of a generating function [14].

$$\frac{}{\vdash T < T} \tag{7}$$

4.2 Server Page Types

In order to formalize the NSP coding guidelines and rules the type system of Core NSP comprises server page types $\mathbb{P}$, web signatures $\mathbb{W}$, a single complete web page type $\square \in \mathbb{C}$, dynamic fragment types $\mathbb{D}$, layout types $\mathbb{L}$, tag element types $\mathbb{E}$, form occurence types $\mathbb{F}$ and system types $\mathbb{S}$.

A server page type is a functional type, that has a web signature as argument type. An include server page has a dynamic document fragment type as result type, and a web server page the unique complete web page type.

$$\mathbb{P} = \{\ w \to r \mid w \in \mathbb{W}\ ,\ r \in \mathbb{C} \cup \mathbb{D}\ \}$$

$$\mathbb{W} = (\mathbb{F}\mathbf{Label}) \ \longrightarrow\ (\mathbb{T} \cup \mathbb{P})$$

$$\mathbb{C} = \{\square\}$$

A web signature is a record. This time a labeled component of a record type is either a programming language type or a server page type, that is the type system supports higher order server pages. Noteworthy a clean separation between the programming language types and the additional NSP specific types is kept. Server page types may be formal parameter types, but these formal parameters can be used only by specific NSP tags. Server pages deliberately become no first class citizens, because this way the Core NSP models conservative amalgamation of NSP concepts with a high-level programming language.

The preserving subtyping rule 4 for records equally applies to web signatures. The establishing subtyping rule 3 must be slightly modified resulting in rule 8, because formal parameters of server page type must always be provided, too.

Subtyping rule 9 is standard and states, that server page types are contravariant in the argument type and covariant in the result type.

$$\frac{T_j \notin \mathbb{B}_{primitive} \cup \mathbb{P} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{8}$$

$$\frac{\vdash w' < w \qquad \vdash R < R'}{\vdash w \to R < w' \to R'} \tag{9}$$

A part of a core document has a document fragment type. Such a type consists of a layout type and a web signature. The web signature is the type of the data, which is eventually provided by the document fragment as part of an actual form parameter. If a web signature plays part of a document fragment type it is also called form type. The layout type constrains the usability of the document fragment as part of an encompassing document. It consists of an element type and a form occurence type.

$$\mathbb{D} = \mathbb{L} \times \mathbb{W}$$

$$\mathbb{L} = \mathbb{E} \times \mathbb{F}$$

$$\frac{\vdash S_1 < T_1 \quad \vdash S_2 < T_2}{\vdash (S_1, S_2) < (T_1, T_2)} \tag{10}$$

Subtyping rule 10 is standard for products and applies both to layout and tag element types. An element type partly describes where a document fragment may be used. Document fragment that are sure to produce no output have the neutral document type $\circ$. Examples for such neutral document parts are hidden parameters and pure java code. Document fragments that may produce visible data like String data or controls have the output type $\bullet$. Document fragments that may produce list elements, table data, table rows or select list options have type **LI**,**TD**, **TR** and **OP**. They may be used in contexts where the respective element is demanded. Neutral code can be used everywhere. This is expressed by rule 11.

$$\mathbb{E} = \{\ \circ, \bullet, \mathbf{TR}, \mathbf{TD}, \mathbf{LI}, \mathbf{OP}\}$$

$$\frac{T \in \mathbb{E}}{\vdash \circ\ <\ T} \tag{11}$$

The form occurrence types further constrains the usability of document fragments. Fragments that must be used inside a form, because they generate client page parts containing controls, have the inside form type $\Downarrow$. Fragments that must be used outside a form, because they generate client page fragments that already contain forms, have the outside form type $\Uparrow$. Fragments that may be used inside or outside forms have the neutral form type $\Updownarrow$. Rule 12 specifies, that such fragments can play the role of both fragments of outside form and fragments of inside form type.

$$\mathbb{F} = \{\ \Downarrow, \Uparrow, \Updownarrow\ \}$$

$$\frac{T \in \mathbb{F}}{\vdash \Updownarrow <\ T} \tag{12}$$

$$\mathbb{S} = \{\ \diamond, \surd\ \}$$

An NSP system is a collection of NSP server pages. NSP systems that are type correct receive the well type $\diamond$. The complete type $\surd$ is used for complete systems. A complete system is a well typed system where all used server page names are defined, i.e. are assigned to a server page of the system, and no server page names are used as variables.

5 Type Operators

In the NSP typing rules in section 7 a central type operation, termed form type composition $\odot$ in the sequel, is used that describes the composition of form content fragments with respect to the provided actual superparameter type. First an auxiliary operator $*$ is defined, which provides the dual effect of the array item type extractor $\Downarrow$ in section **??**. If applied to an array the operater lets the type unchanged, otherwise it yields the respective array type.

$$T* \equiv_{\mathtt{DEF}} \begin{cases} \mathtt{array\ of}\ T & , T \notin \mathbb{A} \\ T & , else \end{cases}$$

The form type composition $\odot$ is the corner stone of the NSP type system. Form content provides direct input capabilities, data selection capabilities and hidden parameters. On submit an actual superparameter is transmitted. The type of this superparameter can be determined statically in NSP, it is called the form type (section 4.2) of the form content. Equally document fragments, which dynamically may generate form content, have a form type. Form type composition is applied to form parameter types and describes the effect of sequencing document parts. Consequently form type composition is used to specify typing with respect to programming language sequencing, loops and document composition.

$w_1 \odot w_2 \equiv_{\mathtt{DEF}}$

$$\begin{cases} \bot & , \boldsymbol{if}\ \exists(l_1 \mapsto T_1) \in w_1 \bullet \exists(l_2 \mapsto T_2) \in w_2 \bullet l_1 = l_2 \wedge P_1 \in \mathbb{P} \wedge P_2 \in \mathbb{P} \\ \bot & , \boldsymbol{if}\ \exists(l_1 \mapsto T_1) \in w_1 \bullet \exists(l_2 \mapsto T_2) \in w_2 \bullet l_1 = l_2 \wedge T_1 \sqcup T_2\ undefined \\ \begin{array}{ll} & (\mathtt{dom}\ w_2) \lhd w_1 \cup (\mathtt{dom}\ w_1) \lhd w_2 \\ \cup & \{ (l \mapsto (T_1 \sqcup T_2)*) \mid (l \mapsto T_1) \in w_1 \wedge (l \mapsto T_2) \in w_2 \} \end{array} & , \boldsymbol{else} \end{cases}$$

If a document fragment targets a formal parameter of a certain type and another document fragment does not target this formal parameter, then and only then the document resulting from sequencing the document parts targets the given formal parameter with unchanged type. That is, with respect to non-overlapping parts of form types, form type composition is just union. With antidomain restriction notation [34] this is specified succinctly in line 3 of the $\odot$ operator definition.

Two document fragments that target the same formal parameters may be sequenced, if the targeted formal parameter types are compatible for each formal parameter. NSP types are compatible if they have a supertype in common. The NSP subtype relation formalizes when an actual parameter may be submitted to a dialogue method: if its type is a subtype of the targeted formal parameter. So if two documents have targeted parameters with compatible types in common only, the joined document may target every dialogue method that fulfills the following: formal parameters that are targeted by both document parts have an array type, because of sequencing a single data transmission cannot be ensured in neither case, thereby the array items' type must be a common supertype of the targeting actual parameters. This is formalized in line 4 of the the $\odot$ operator definition: for every shared formal parameter a formal array parameter of the least common supertype belongs to the result form type. The least common supertype of two types is given as least upper bound of the two types, which is unique up to the equality induced by recursive subtyping itself. Consider the following example application of the $\odot$ operator:

$$\begin{array}{lll} & \{l \mapsto \mathtt{int}, n \mapsto \{o \mapsto \mathtt{int}, p \mapsto \mathtt{String}\}\} & (T_1) \\ \odot & \{m \mapsto \mathtt{int}, n \mapsto \{o \mapsto \mathtt{int}, q \mapsto \mathtt{String}\}\} & (T_2) \\ & & \\ = & \begin{array}{l} \{\ \ l \mapsto \mathtt{int}, \\ \quad m \mapsto \mathtt{int}, \\ \quad n \mapsto \mathtt{array\ of} \{o \mapsto \mathtt{int}, p \mapsto \mathtt{String}, q \mapsto \mathtt{String}\} \\ \} \end{array} & (T_3) \end{array}$$

In the example two form fragments are concatenated, the first one having type T_1, the second one having type T_2. The compound form content will provide int values for the formal parameters l and m. It will provide to actual parameters for the formal parameter n. Thereby the record stemming from the first form fragment can be automatically filled with a null object for a formal q parameter of type String, because String is a non-primitive type. Analogously, the record stemming from the second form fragment can be automatically filled with a null object for a formal p parameter. The compound form document therefore can target a dialogue method with web signature T_3.

The error cases in the $\odot$ operator definition are equally important. The $\odot$ operator is a partial function. If two document fragments target a same formal parameter with non-compatible types, they simply cannot be sequenced. The $\odot$ operator is undefined for the respective form types. More interestingly, two document fragments that should be composed must not target a formal server page parameter. This would result in an actual server page parameter array which would contradict the overall principle of conservative language amalgamation introduced in chapter **??**.

Form type composition can be characterized algebraically. The web signatures form a monoid $(\ \mathbb{W}\ ,\ \odot\ ,\ \emptyset\ )$ with the $\odot$ operator as monoid operation and the empty web signature as neutral element. The operation $(\lambda v.v \odot w)_w$ is idempotent for every arbitrary fixed web signature w, which explains why the typing rule 24 for loop-structures is adequate.

6 Environments and Judgements

In the NSP type system two environments are used. The first environment Γ is the usual type environment. The second environment Δ is used for binding names to server pages, i.e. as a definition environment. It follows from their declaration that environments are web signatures. All definitions coined for web signatures immediately apply to the environments. This is exploited for example in the system parts typing rule 46.

$$\begin{aligned} \Gamma : (\mathbb{F}\mathbf{Label}) &\longrightarrow (\mathbb{T} \cup \mathbb{P}) && = \mathbb{W} \\ \Delta : (\mathbb{F}\mathbf{Label}) &\longrightarrow \mathbb{P} && \subset \mathbb{W} \end{aligned}$$

The Core NSP identifiers are used for basic programming language expressions, namely variables and constants, and for page identifiers, namely formal page parameters and server pages names belonging to the complete system. In some contexts, e.g. in hidden parameters or in select menu option values, both page identifiers and arbitrary programming language expressions are allowed. Therefore initially page identifiers are treated syntactically as programming language expressions. However a clean cut between page identifiers and the programming language is maintained, because the modeling of conservative amalgamation is an objective. The cut is provided by the premises of typing rules concerning such elements where only a certain kind of entity is allowed; e.g. in the statement typing rule 16 it is prevented that page identifiers may become program parts.

The Core NSP type system relies on several typing judgments:

$$\begin{array}{ll} \Gamma \vdash e : \mathbb{T} \cup \mathbb{P} & e \in \mathbf{expr} \\ \Gamma \vdash n : \mathbb{D} & n \in \mathbf{com} \cup \mathbf{dynamic} \\ \Gamma \vdash c : \mathbb{P} & c \in \mathbf{websig\text{-}core} \\ \Gamma \vdash a : \mathbb{W} & a \in \mathbf{actualparams} \\ \Gamma, \Delta \vdash s : \mathbb{S} & s \in \mathbf{system} \end{array}$$

Eventually the judgment that a system has complete type is targeted. In order to achieve this, different kinds of types must be derived for entities of different syntactic categories. Expressions have programming language types or page types, consequently along the lines just discussed. Both programming language code and user interface descriptions have document fragment types, because they can be interlaced arbitrarily and therefore belong conceptually to the same kind of document. Parameterized core documents have page types. The actual parameters of a call element together provide an actual superparameter, the type of this is a web signature and is termed a call type. All the kinds of judgments so far work with respect to a given type environment. If documents are considered as parts of a system they must mutually respect defined server page names. Therefore subsystem judgments has to be given additionally with respect to the defintion environment.

7 Typing Rules

The notion of Core NSP type correctness is specified as an algorithmic type system. In the presence of subtyping there are two alternatives for specifying type correctness with a type system. The first one is by means of a declarative type system. In such a type system a subsumption rule is present. Whenever necessary it can be derived that an entity has always each of its supertypes. Instead in an algorithmic type system reasoning about an entities' supertypes happens in a controlled way by fulfilling typing rule premises. Both approaches have their advantages and drawbacks. The declarative approach usually leads to more succinct typing rules whereas reasoning about type system properties may become complicated - cut elimination techniques may have to be employed. In the algorithmic approach the single typing rules may quickly become complex, however an algorithmic type system is easier to handle in proofs.

For Core NSP an algorithmic type system is the correct choice. Extra premises are needed in some of the typing rules, e.g. in the typing rule for form submission. In some rules slightly bit more complex type patterns have to be used in the premises, e.g. in the typing rules concerning layout structuring document elements. However in the Core NSP type system these extra complexity fosters understandability. The typing rules are presented by starting from basic building blocks to more complex building blocks.

The typing rule 13 allows for extraction of an identifier typing assumption from the typing environment. Rules 14 and 15 give the types of selected record fields respectively indexed array elements.

$$\frac{(v \mapsto T) \in \Gamma}{\Gamma \vdash v : T} \tag{13}$$

$$\frac{\Gamma \vdash e : \{l_i \mapsto T_i\}^{i \in 1 \ldots n} \qquad j \in 1 \ldots n}{\Gamma \vdash e.l_j : T_j} \tag{14}$$

$$\frac{\Gamma \vdash e : \texttt{array of } T \qquad \Gamma \vdash i : \texttt{int}}{\Gamma \vdash e[i] : T} \tag{15}$$

Typing rule 16 introduces programming language statements, namely assignments. Only programming language variables and expression may be used, i.e. expressions must not contain page identifiers. The resulting statement is sure not to produce any output. It is possible to write an assignment inside forms and outside forms. If it is used inside a form it will not contribute to the submitted superparameter. Therefore a statement has a document fragment type which is composed out of the neutral document type, the neutral form type and the empty web signature. The empty string, which is explicitly allowed as content in NSP, obtains the same type by rule 17.

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T \qquad T \in \mathbb{T}}{\Gamma \vdash x := e \ : \ ((\circ, \Updownarrow), \emptyset)} \tag{16}$$

$$\frac{}{\Gamma \vdash \varepsilon : ((\circ, \Updownarrow), \emptyset)} \tag{17}$$

Actually in Core NSP programming language and user interface description language are interlaced tightly by the abstract syntax. The code tags are just a means to relate the syntax to common concrete server pages syntax. The code tags are used to switch explicitly between programming language and user interface description and back. For the latter the tags may be read in reverse order. However this switching does not affect the document fragment type and therefore the rules 18 and 19 do not, too.

$$\frac{\Gamma \vdash c : D}{\Gamma \vdash <\texttt{code}> c </\texttt{code}> \ : \ D} \tag{18}$$

$$\frac{\Gamma \vdash d : D}{\Gamma \vdash </\texttt{code}> d <\texttt{code}> \ : \ D} \tag{19}$$

Equally basic as rule 16, rule 20 introduces character strings as well typed user interface descriptions. A string's type consists of the output type, the neutral form type and the empty web signature. Another way to produce output is by means of expression elements, which support all basic types and get by rule 21 the same type as character strings.

$$\frac{d \in \textbf{string}}{\Gamma \vdash d : ((\bullet, \Updownarrow), \emptyset)} \tag{20}$$

$$\frac{\Gamma \vdash e : T \quad T \in \mathbb{B}}{\Gamma \vdash <\texttt{expression}> e </\texttt{expression}> \ : \ ((\bullet, \Updownarrow), \emptyset)} \tag{21}$$

Composing user descriptions parts and sequencing programming language parts must follow essentially the same typing rule. In both rule 22 and rule 23 premises ensure that the document fragment types of both document parts are compatible. If the parts have a common layout supertype, they may be used together in server pages contexts of that type. If in addition to that the composition of the parts' form types is defined, the composition becomes the resulting form type. Form composition has been explained in section 5.

$$\frac{\begin{array}{c} d_1, d_2 \in \textbf{dynamic} \\ \Gamma \vdash d_1 : (L_1, w_1) \quad \Gamma \vdash d_2 : (L_2, w_2) \quad L_1 \sqcup L_2 \downarrow \quad w_1 \odot w_2 \downarrow \end{array}}{\Gamma \vdash d_1 \ d_2 : (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{22}$$

$$\frac{\Gamma \vdash c_1 : (L_1, w_1) \qquad \Gamma \vdash c_2 : (L_2, w_2) \qquad L_1 \sqcup L_2 \downarrow \qquad w_1 \odot w_2 \downarrow}{\Gamma \vdash c_1 ; c_2 \ : \ (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{23}$$

The loop is a means of dynamically sequencing. From the type system's point of view it suffices to regard it as a sequence of twice the loop body as expressed by typing rule 24. For an if-then-else-structure the types of both branches must be compatible in order to yield a well-typed structure. Either one or the other branch is executed, so the least upper bound of the layout types and least upper bound of the form types establish the adequate new document fragment type.

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash c : (L, w)}{\Gamma \vdash \texttt{while}\, e \,\texttt{do}\, c \ : \ (L, w \odot w)} \tag{24}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash c_1 : D_1 \quad \Gamma \vdash c_2 : D_2 \qquad D_1 \sqcup D_2 \downarrow}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1 \texttt{else}\ c_2 \ : \ D_1 \sqcup D_2} \tag{25}$$

Next the typing rules for controls are considered. The submit button is a visible control and must not occur outside a form, in Core NSP it is an empty element. It obtains the output type, the inside form type, and the empty web signature as document fragment type. Similarly an input control obtains the output type and the inside form type. But an input control introduces a form type. The type of the input control is syntactically fixed to be a widget supported type. The param-attribute of the control is mapped to the control's type. This pair becomes the form type in the control's document fragment type. Check boxes are similar. In Core NSP check boxes are only used to gather boolean data.

$$\frac{}{\Gamma \vdash <\texttt{submit}/> \ : \ ((\bullet, \Downarrow), \emptyset)} \tag{26}$$

$$\frac{T \in \mathbb{B}_{supported}}{\Gamma \vdash \ <\texttt{input}\ \texttt{type} = "T"\ \texttt{param} = "l"/>: ((\bullet, \Downarrow), \{(l \mapsto T)\})} \tag{27}$$

$$\frac{}{\Gamma \vdash <\texttt{checkbox} \quad \texttt{param} = "l"/> \ : \ ((\bullet, \Updownarrow), \{(l \mapsto \mathit{boolean})\})} \tag{28}$$

Hidden parameters are not visible. They get the neutral form type as part of their fragment type. The value of the hidden parameter may be a programming language expression of arbitrary type or an identifier of page type.

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash <\texttt{hidden}\ \texttt{param} = "l" > e </\texttt{hidden}> \ : \ ((\circ, \Downarrow), \{(l \mapsto T)\})} \tag{29}$$

The select element may only contain code that generates option elements. Therefore an option element obtains the option type **OP** by rule 31 and the select element typing rule 30 requires this option type from its content. An option element has not an own param-element. The interesting type information concerning the option value is wrapped as an array type that is assigned to an arbitrary label. The type information is used by rule 30 to construct the correct form type. This way no new kind of judgement has to be introduced for select menu options.

$$\frac{\Gamma \vdash d : \big((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of}\ T)\}\big)}{\Gamma \vdash \begin{array}{l} <\texttt{select} \quad \texttt{param} = "l" > \\ \qquad d \\ </\texttt{select}> \ : \ ((\bullet, \Downarrow), \{(l \mapsto \texttt{array of}\ T)\}) \end{array}} \tag{30}$$

$$\frac{\Gamma \vdash v \ : \ T \qquad \Gamma \vdash e : S \quad S \in \mathbb{B} \qquad l \in \mathbf{Label}}{\Gamma \vdash \begin{array}{l} <\texttt{option}> \\ \qquad <\texttt{value}> v </\texttt{value}> \\ \qquad <\texttt{label}> e </\texttt{label}> \\ </\texttt{option}> : ((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of}\ T)\}) \end{array}} \tag{31}$$

The object element is a record construction facility. The enclosed document fragment's layout type lasts after application of typing rule 32, whereas the fragment's form type is assigned to the object element's param-attribute. This way the superparameter provided by the enclosed document becomes a named object attribute.

$$\frac{\Gamma \vdash d : (L, w)}{\Gamma \vdash <\texttt{object}\ \texttt{param} = "l" > d </\texttt{object}> \ : \ (L, \{(l \mapsto w)\})} \tag{32}$$

The form typing rule 33 requires that a form may target only a server page that yields a complete web page if it is called. Furthermore the form type of the form content must be a subtype of the targeted web signature, because the Core NSP subtype relations specifies when a form parameter may be submitted to a dialogue method of given signature. Furthermore the form content's must be allowed to occur inside a form. Then the rule 33 specifies that the form is a vizible element that must not contain inside another form.

$$\frac{\Gamma \vdash l : w \to \Box \qquad \Gamma \vdash d : ((e, \Downarrow), v) \qquad \vdash v < w}{\Gamma \vdash <\texttt{form}\quad \texttt{callee} = "l" > d < /\texttt{form} > : ((e, \Uparrow), \emptyset)} \tag{33}$$

Now the layout structuring elements, i.e. lists and tables, are investigated. The corresponding typing rules 34 to 38 do not affect the form types and form occurrence types of contained elements. Only document parts that have no specific layout type, i.e. are either neutral or merely vizible, are allowed to become list items by rule 34. Only documents with list layout type may become part of a list. A well-typed list is a vizible element. The rules 36 to 38 work analogously for tables.

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{< \texttt{li} > d < /\texttt{li} > \ : \ ((\mathbf{LI}, F), w)} \tag{34}$$

$$\frac{\Gamma \vdash d : ((\mathbf{LI} \vee \circ, F), w)}{< \texttt{ul} > d < /\texttt{ul} > \ : \ ((\bullet, F), w)} \tag{35}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{< \texttt{td} > d < /\texttt{td} > \ : \ ((\mathbf{TD}, F), w)} \tag{36}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TD} \vee \circ, F), w)}{< \texttt{tr} > d < /\texttt{tr} > \ : \ ((\mathbf{TR}, F), w)} \tag{37}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TR} \vee \circ, F), w)}{< \texttt{table} > d < /\texttt{table} > \ : \ ((\bullet, F), w)} \tag{38}$$

As the last core document element the server side call is treated. A call element may only contain actual parameter elements. This is ensured syntactically. The special sign $\varepsilon_{\texttt{act}}$ acts as an empty parameter list if necessary. It has the empty web signature as call type. Typing rule 41 makes it possible that several actual parameter elements uniquely provide the parameters for a server side call. Rule 39 specifies, that a server call can target an include server page only. The call element inherits the targeted include server page's document fragment type, because this page will replace the call element if it is called.

$$\frac{\Gamma \vdash l : w \to D \qquad \Gamma \vdash as : v \qquad \vdash v < w}{\Gamma \vdash < \texttt{call}\quad \texttt{callee} = "l" > as < /\texttt{call} > \ : \ D} \tag{39}$$

$$\frac{}{\Gamma \vdash \varepsilon_{\texttt{act}} : \emptyset} \tag{40}$$

$$\frac{\Gamma \vdash as : w \qquad \Gamma \vdash e : T \qquad l \notin (dom\ w)}{\Gamma \vdash \begin{array}{l} < \texttt{actualparam}\quad \texttt{param} = "l" > \\ \qquad e \\ < /\texttt{actualparam} > as \ : \ w \cup \{(l \mapsto T)\} \end{array}} \tag{41}$$

With the typing rule 42 and 45 arbitrary document fragment may become an include server page, thereby the document fragment's type becomes the server page's result type. A document fragment may become a complete web page by typing rules 43 and 45 if it has no specific layout type, i.e. is neutral or merely visible, and furthermore is not intended to be used inside forms. The resulting server page obtains the complete type as result type. Both include server page cores and web server page cores start with no formal parameters initially. With rule 44 parameters can be added to server page cores. The rule's premises ensure that a new formal parameter must have another name than all the other parameters and that the formal parameter is used in the core document type-correctly. A binding of a type to a new formal parameter's name is erased from the type environment.

$$\frac{\Gamma \vdash d : D \qquad d \in \textbf{dynamic}}{\Gamma \vdash <\texttt{include}> d </\texttt{include}> : \emptyset \to D} \tag{42}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, \Updownarrow \vee \Uparrow), \emptyset) \qquad t \in \textbf{strings} \qquad d \in \textbf{dynamic}}{\Gamma \vdash \begin{array}{l} <\texttt{html}> \\ \qquad <\texttt{head}><\texttt{title}> t </\texttt{head}></\texttt{title}> \\ \qquad <\texttt{body}> d </\texttt{body}> \\ </\texttt{html}> : \emptyset \to \Box \end{array}} \tag{43}$$

$$\frac{\Gamma \vdash l : T \qquad \Gamma \vdash c : w \to D \qquad l \notin (\mathit{dom}\ w)}{\Gamma \backslash (l \mapsto T) \vdash \begin{array}{l} <\texttt{param}\quad \texttt{name} = "l" \quad \texttt{type} = "T" /> \\ c : (w \cup \{(l \mapsto T)\}) \to D \end{array}} \tag{44}$$

$$\frac{\Gamma \vdash l : P \qquad \Gamma \vdash c : P \qquad c \in \textbf{websig-core}}{\Gamma \backslash (l \mapsto P), \{(l \mapsto P)\} \vdash <\texttt{nsp}\quad \texttt{name} = "l" > c </\texttt{nsp}> \ : \ \diamond} \tag{45}$$

A server page core can become a well-typed server page by rule 45. The new server page name and the type bound to it are taken from the type environment and become the definition environment. An NSP system is a collection of NSP server pages. A single well-typed server page is already a system. Rule 46 specifies system compatibility. Rule 47 specifies system completeness. Two systems are compatible if they have no overlapping server page definitions. Furthermore the server pages that are defined in one system and used in the other must be able to process the data they receive from the other system, therefore the types of the server pages defined in the one system must be subtypes of the ones bound to their names in the other's system type environment.

$$\frac{\begin{array}{c} s_1, s_2 \in \textbf{system} \quad (\mathit{dom}\ \Delta_1) \cap (\mathit{dom}\ \Delta_2) = \emptyset \\ ((\mathit{dom}\ \Gamma_2) \lhd\ \Delta_1) < ((\mathit{dom}\ \Delta_1) \lhd\ \Gamma_2) \\ ((\mathit{dom}\ \Gamma_1) \lhd\ \Delta_2) < ((\mathit{dom}\ \Delta_2) \lhd\ \Gamma_1) \\ \Gamma_1, \Delta_1 \vdash s_1 : \diamond \qquad \Gamma_2, \Delta_2 \vdash s_2 : \diamond \end{array}}{((\mathit{dom}\ \Delta_2) \lessdot \Gamma_1) \cup ((\mathit{dom}\ \Delta_1) \lessdot \Gamma_2)\ ,\ \Delta_1 \cup \Delta_2\ \vdash s_1\ s_2\ :\ \diamond} \tag{46}$$

$$\frac{\begin{array}{c} \Gamma \in \mathbb{R} \\ (\mathit{dom}\ \Delta) \cap \mathit{bound}(s) = \emptyset \\ \Gamma, \Delta \vdash s : \diamond \end{array}}{\Gamma, \Delta \vdash s : \surd} \tag{47}$$

Typing rule 47 specifies when a well-typed system is complete. First, all of the used server pages must be defined, that is the type environment is a pure record type. Second server page definitions may not occur as bound variables somewhere in the system.

Theorem 7.1 *Core NSP type checking is decidable.*

Proof(7.1): Core NSP is explicitly typed. The Core NSP type system is algorithmic. Recursive subtyping is decidable. The least upper bound can be considered as a union operation during type checking - as a result a form content is considered to have a finite collection of types, which are checked each against a targeted server page if rule 33 is applied.□

8 Related Work

WASH/HTML is a mature embedded domain specific language for dynamic XML coding in the functional programming language Haskell, which is given by combinator libraries [35][36]. In [36] four levels of XML validity are defined. Well-formedness is the property of correct block structure, i.e. correct matching of opening and closing tags. Weak validity and elementary validity are both certain limited conformances to a given document type definition (DTD). Full validity is full conformance to a given DTD. The WASH/HTML approach can guarantee full validity of generated XML. It only guarantees weak validity with respect to the HTML SGML DTD under an immediate understanding of the defined XML validity levels for SGML documents. As a reason for this the exclusion and inclusion exceptions in the HTML SGML DTD are given. The problem is considered less severe for the reason that in the XHTML DTD [38] exceptions only occur as comments - in XML DTDs no exception mechanism is available - and XHTML has been created to overcome HTML. Unfortunately these comments become normative status in the corresponding XHMTL standard [37]; they are called element prohibitions. Therefore the problem of weak versus full validity remains an issue for XHTML, too.

The Core NSP type system shows that it is possible to statically ensure normative element prohibitions of the XHMTL standard. Anyway, despite questions concerning concrete technologies like fullfilling HTML/XHMTL are very interesting, the NSP approach targets to understand user interface description safety on a more conceptual level: the obvious, nonetheless important, statement is that it is possible to check arbitrary context free constraints on tag element nestings. In [32][4] it is shown that the normative element prohibitions of the XHMTL standard [37] can be statically checked by employing flow analysis [26][29][28].

There are a couple of other projects for dynamic XML generation, that garuantee some level of user interface description language safety, e.g. [15][17][23]. We delve on some further representative examples. In [39] two approaches are investigated. The first provides a library for XML processing arbitrary documents, thereby ensuring well-formedness. The second is a type-based translation framework for XML documents with respect to a given DTD, which garuantees full XML validity. Haskell Server Pages [25] garuantee well-formedness of XML documents. The small functional programming language XMλ [33] is based on XML documents as basic datatypes and is designed to ensure full XML validity [24].

9 Conclusion

The core type system of NSP has been given as a convenient Per Martin-Löf style type system. This enables precise reasoning about the NSP concepts. The NSP Coding guidelines and rules give an informal explanation of NSP type correctness. They are easy to learn and will help in everyday programming tasks, but may give rise to ambiguity. A precise description of the static semantics of NSP languages is desired. The formal Core NSP type system provides a succinct precise definition at the right level of communication. The formal type system definition makes it easier to adapt results from the vast amount of literature on type systems to the NSP approach, especially concerning type inference resp. type checking algorithms.

References

[1] Martin Abadi, Luca Cardelli. A Theory of Primitive Objects - Untyped and First-Order Systems. Information and Computation, 125(2), pp.78-102, 1996. Earlier version appeared in TACS '94 proceedings, LNCS 789, 1994.

[2] Martin Abadi, Luca Cardelli. A Theory Of Objects. Springer, 1996.

[3] H.P. Barendregt: Lambda Calculi with Types. In: S.Abramsky, D.V. Gabbay, T.S.E. Maibaum (eds.): Handbook of Logic in Computer Science, vol.2., pp.118-331. Clarendon Press, 1992.

[4] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In: Proceedings of Workshop on Program Analysis for Software Tools and Engineering. ACM, 2001.

[5] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). World Wide Web Consortium, 2000.

[6] Luca Cardelli. Type systems. In: Handbook of Computer Science and Engineering. CRC Press, 1997

[7] Daniel W. Connolly Document Type Definition for the HyperText Markup Language, level 2. World Wide Web Consortium, 1995.

[8] Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. In: Computing Surveys, Vol. 17, No. 4, ACM 1985.

[9] Bruno Courcelle. Fundamental Properties of Infinite Trees. In: Theoretical Computer Science 25, pp.95-169. Norh-Holland Publishing Company, 1983.

[10] Crary, K., R. Harper and S. Puri. What is a recursive module? In: Proc. ACM Conference on Programming Language Design and Implementation, pages 50–63, May 1999.

[11] D. Crocker P. Overell (editors). Augmented BNF for Syntax Specifications: ABNF. Dequest for Comments: 2234. Network Working Group, November 1997.

[12] Draheim, D., Weber, G.: Strongly Typed Server Pages. In: Proceedings of The Fifth Workshop on Next Generation Information Technologies and Systems, LNCS. Springer-Verlag, June 2002.

[13] Elfriede Fehr. Semantik von Programmiersprachen. Springer-Verlag, 1989.

[14] Vladimir Gapayev, Michael Y. Levin, Benjamin C. Pierce. Recursive Subtyping Revealed. In: International Conference on Functional Programming, 2000. To appear in Journal of Functional Programming

[15] Andy Gill. HTML combinators, version 2.0.
http://www.cse.ogi.edu/ andy/html/intro.htm, 2002.

[16] Gunter, C.A.: Semantics of Programming Languages - Structures and Techniques. The MIT Press, 1992.

[17] Michael Hanus. Server side Web scripting in Curry. In Workshop on (Constraint) Logic Programming and Software Engineering (LPSE2000), London, July 2000.

[18] Ian Hayes. Specification Case Studies. Prentice Hall, 1993.

[19] International Standardization Organisation. ISO 8879. Information Processing – Text and Office Systems - Standard Generalized Markup Language (SGML). ISO,1986.

[20] International Standardization Organisation. International Standard ISO/ICE 14977. Syntactic metalanguage - Extended BNF. ISO,1996.

[21] Pekka Kilpeläinen, Derick Wood. SGML and Exceptions, Department of Computer Science, University of Helsinki, Technical Report HKUST-CS96-03, 1996.

[22] Barbara Liskov. Data Abstraction and Hierarchy. SIGPLAN Notices. 23(5), May 1988.

[23] Erik Meijer. Server-side Scripting in Haskell. Journal of Functional Programming, 2000.

[24] Erik Meijer and Mark Shields. XMλ - A Functional Language for Constructing and Manipulating XML Documents.
http://www.cse.ogi.edu/~mbs, Draft, 2000.

[25] Erik Meijer, Danny van Velzen. Haskell Server Pages - Functional Programming and the Battle for the Middle Tier. Electronic Notes in Theoretical Computer Science 41, No.1, Elsevier Science, 2001.

[26] Flemming Nielson, Hanne Nielson, Chris Hankin: Principles of Program Analysis. Springer-Verlag, 1999.

[27] Bengt Nordström, Kent Peterson, Jan M. Smith: Programming in Martin-Löfs Type Theory. The International Series of Monographs on Computer Science. Clarendon Press, 1990.

[28] Palsberg, J., and O'Keefe, P. A type system equivalent to flow analysis. In Proceedings of the ACM SIGPLAN '95 Conference on Principles of Programming Languages, pp. 367-378, 1995.

[29] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. Information and Computation, 118(1), pp.128-141, 1995.

[30] Benjamin C. Pierce: Types and Programming Languages. MIT Press, 2002.

[31] Davide Sangiorgi, David Walker. The π-calculus - A Theory of Mobile Processes. Cambridge University Press, 2001.

[32] Sandholm, A., Schwartzbach, M.I.: A type system for dynamic web documents . In Reps, T., ed.: Proc. 27th Annual ACM Symposium on Principles of Programming Languages, pp. 290-301, ACM Press, 2000.

[33] Shields, M., and Meijer, E. Type-indexed rows . In Proceedings of the 28th Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL'01), ACM Press, pp. 261-275, 2001.

[34] J.M. Spivey. The Z Notation. Prentice Hall, 1992.

[35] Peter Thiemann. "Modeling HTML in Haskell". Practical Applications of Declarative Programming, PADL '00. volume 1753 of Lecture Notes in Computer Science. January 2000.

[36] Peter Thiemann. "A typed representation for HTML and XML documents in Haskell". February 2001.

[37] The W3C HTML working group. XHTML 1.0 The Extensible HyperText Markup Language.
http://www.w3.org/TR/xhtml1/. W3C, 2000.

[38] The W3C HTML working group. Extensible HTML version 1.0 Strict DTD.
http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd W3C, 2000.

[39] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or typebased translation? ACM SIGPLAN Notices, 34(9):148-159, Sept. 1999. Proceedings of ICFP'99.

[40] D. Wood. Standard generalized markup language: Mathematical and philosophical issues . In J. van Leeuwen, editor, Computer Science Today. Recent Trends and Developments, volume 1000 of Lecture Notes in Computer Science, pages 344-365. Springer-Verlag, 1995. 60

[41] J.B. Wordsworth. Software Developement with Z - A Practical Approach to Formal Methods in Software Engineering. Addison Wesley, 1992.