Chapter 7

Formal Definition of the NSP Type System

This chapter formalizes the type system of Core NSP [63], which is the amalgamation of NSP concepts with a minimal imperative programming language similar to WHILE [71], 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 [115][116][132] style type system is given to specify type correctness. The aims of this formal NSP type system definition are manifold:

- Preciseness. 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.

- Justification. The implementations of the JSPick tool already convinces that NSP concepts like static client page description and type safety for server pages really work in practice. A formal type system definition makes it obvious.

- Linkage. A 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.

The results of this chapter are summarized in appendix B in order to provide a comprehensive easy reference. Furthermore example type derivation are given in this appendix in B.7 and B.8.

7.1 Core NSP Grammar

An abstract syntax of Core NSP programs is specified by a context free grammar[1] [2]. In appendix C.1 an alternative specification of Core NSP as an XML document is given by a document type definition (DTD) [19]. 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:

```
      system  ::=  page | system system
        page  ::=  <nsp name="id"> websig-core </nsp>
 websig-core  ::=  param websig-core | webcall | include
       param  ::=  <param name="id" type="parameter-type"/>
     webcall  ::=  <html> head body </html>
        head  ::=  <head><title> strings </title></head>
     strings  ::=  ε | string strings
        body  ::=  <body> dynamic </body>
     include  ::=  <include> dynamic </include>
```

There are some basic syntactic categories. The category id is a set of labels. The category string consists of character strings[3]. 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 7.3.

string	::=	$s \in \mathbf{String}$
id	::=	$l \in \mathbf{Label}$
parameter-type	::=	$t \in \mathbb{T} \cup \mathbb{P}$
supported-type	::=	$t \in \mathbb{B}_{supported}$

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.

[1] Nonterminals are underlined. Terminals are not emphasized - contrariwise to BNF standards like [97] or [46], however it fosters readability significantly.

[2] Every nonterminal corresponds to a syntactic category. In the grammar a syntactic category is depicted in bold face.

[3] 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 [19].

```
dynamic  ::=    dynamic dynamic
             |  ε | string
             |  ul | li
             |  table | tr | td
             |  call
             |  form | object | hidden | submit
             |  input | checkbox
             |  select | option
             |  expression
             |  code
```

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 7.2.

```
   ul  ::=  <ul> dynamic </ul>
   li  ::=  <li> dynamic </li>
table  ::=  <table> dynamic </table>
   tr  ::=  <tr> dynamic </tr>
   td  ::=  <td> dynamic </td>
```

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[4].

```
        call  ::=  <call callee="id"> actualparams </call>
actualparams  ::=  ε_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

[4]The call element may contain no element, too. As a matter of taste the special sign $\varepsilon_{\texttt{act}}$ for empty contents is used in the Core NSP grammar to avoid redundant production and typing rules for the call element.

tags in order to express the integration of active code parts with layout[5].

```
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 possesses only a set of most interesting features to model all the complexity of NSP technologies[6].

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.

7.2 Core NSP Type System Strength

The grammar given in 7.1 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 non-terminal for document fragments inside forms and duplicate and appropriately

[5]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.

[6]Core NSP is even not turing-complete. Build-in operations, i.e. a sufficient powerful expression language, must be added. But this would just result in a more complex type system without providing more insight into static NSP semantics.

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 [40] 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 [184], because an SGML expression that includes exceptions can be translated into an extended context free grammar [103][7]. The transformation algorithm given in [103] 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, the resulting mechanism formalizes the way the Amsterdam SGML parser [181] handles exceptions[8].

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 [98] and XHTML DTD [173] specifications are:

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

In Core NSP the document fragment in listing 7.1 is considered correct.

Listing 7.1

```
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>
```

[7] An extended context free grammar is a context free grammar with production rules that may have regular expressions as right hand sides.

[8] The Amsterdam SGML parser deals with exceptions by keeping track of excluded elements in a stack.

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.

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 [168][169]. In [169] 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[9]. 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 [173] exceptions only occur as comments[10] and XHTML has been created to overcome HTML. Unfortunately these comments become normative status in the corresponding XHMTL standard [172]; 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 fulfilling 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[11].

---

[9] There are a couple of other projects for dynamic XML generation, that guarantee some level of user interface description language safety, e.g. [78][86][120]. We delve on some further representative examples. In [180] 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 guarantees full XML validity. Haskell Server Pages [122] guarantee well-formedness of XML documents. The small functional programming language XMλ [162] is based on XML documents as basic datatypes and is designed to ensure full XML validity [121].

[10] In XML DTDs no exception mechanism is available.

[11] In [157][17] it is shown that the normative element prohibitions of the XHMTL standard [172] can be statically checked by employing flow analysis [131][140][139].

7.3 Core NSP Types

In this section the types of Core NSP and the subtype relation between types are introduced simultaneously.

- Core NSP types. 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 [163] is used. Every type represents an infinite labeled regular tree.

- Core NSP subtyping. The subtype relation formalizes the relationship of actual client page parameters and formal server page parameters by strictly applying the Barbara Liskov principle [111][12]. 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 $\vdash\ S < T$.

7.3.1 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 [26], i.e. there is no concept of introducing named user defined types, which would not contribute to the understanding of NSP concepts[13]. The types introduced so far and the type variables $\mathbb{V}$ together form the set of programming language types $\mathbb{T}$.

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

[12] Liskov Substitution Principle: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.

[13] Enumeration types $\mathbb{E}$, i.e subtype polymorphism in the narrow sense of [26], could be introduced in order to specify the behavior of radio button tag structures.

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

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[14].

For every programming language type, there is an array type. According to subtyping rule 7.1 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 7.2 is the preserving subtyping rule for array types.

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

$$\frac{}{\vdash T < \texttt{array of}\ T} \tag{7.1}$$

$$\frac{\vdash S < T}{\vdash \texttt{array of}\ S < \texttt{array of}\ T} \tag{7.2}$$

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 [145] 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 [158], 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[15] [185][91] will ease writing type operator definitions and

[14] It is a usual economy not to introduce ground types in the presence of type variables [27]. Similarly in Core NSP example programs free term variables are used to model basic constant data values.

[15] $A \twoheadrightarrow\!\!\!\!\!+ B$ is the set of all finite partial functions from A to B.

typing rules later on.

$$\mathbb{R} = \textbf{Label} \nrightarrow \mathbb{T}$$

$$\frac{T_j \notin \mathbb{B}_{primitive} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{7.3}$$

$$\frac{\vdash S_1 < T_1 \ldots \vdash S_n < T_n}{\vdash \{l_i \mapsto S_i\}^{i \in 1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{7.4}$$

Rule 7.4 is just the necessary preserving subtyping rule for records.

The establishing subtyping rule 7.3 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 [27] or objects [1]. But there is no contradiction, because these rules describe hierarchies of feature support[16] and we just specify another phenomenon: rule 7.3 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.

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

$$\frac{\vdash S[{}^{\mu X.S}/_{X}] < T}{\vdash \mu X.S < T} \tag{7.5}$$

$$\frac{\vdash S < T[{}^{\mu X.T}/_{X}]}{\vdash S < \mu X.T} \tag{7.6}$$

Recursive types may be handled in an iso-recursive or an equi-recursive way[17]. 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 [10], 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 [76] 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[18], it is well-understood and some crucial results concerning proof-techniques and type checking of recursive typing

[16] In [55] object subtyping is explained strictly along the notion of feature support.

[17] The terms iso-recursive or an equi-recursive stem from [45].

[18] Compared to iso-recursive typing, an equi-recursive typing is a Curry-style [10], i.e implicit typing discipline, and therefore its theory is more challenging, or in more practical terms, constructing a type checker is more challenging.

and recursive subtyping like [28][3][160][101] 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 as may be checked against the examples given in chapter 3 and the sample type derivations in appendices B.7 and B.8.

In the Core NSP type system types represent finite trees or possibly infinite regular trees[19][42]. More precisely these type trees are unordered[20], 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 [76]. The Core NSP subtyping rules provide an intuitive description of this generating function. Thereby the subtyping rules for left folding 7.5 and right folding 7.6 provide the desired recursive subtyping.

Beyond this only one further subtyping rule is needed, namely the rule 7.7 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 [76].

$$\frac{}{\vdash T < T} \tag{7.7}$$

We will continue to explain server page types in section 7.3.2. But first, a digression on Core NSP recursive record subtyping.

A Digression on Core NSP Recursive Record Subtyping

Normally sum or variant types are used in order to provide some finite data to be inhabited in recursive types. There is no sum or variant type in Core NSP. Instead the establishing record subtyping rule 7.3 makes it possible that finite data may be submitted to a formal parameter of recursive type. For example the finite data gathered in listing 7.2 has the recursive type given by subtype derivation 7.8.

$$\begin{array}{ll} & e \equiv_{\texttt{DEF}} element \\ & n \equiv_{\texttt{DEF}} next \\ & \\ \vdash & \{e \mapsto int, n \mapsto \{e \mapsto int, n \mapsto \{e \mapsto int\}\}\} \\ < & \{e \mapsto int, n \mapsto \{e \mapsto int, n \mapsto \{e \mapsto int, n \mapsto \mu X.\{e \mapsto int, n \mapsto X\}\}\}\} \\ = & \mu X.\{e \mapsto int, n \mapsto X\} \end{array} \tag{7.8}$$

[19] A regular tree is a possibly infinite tree that has a finite set of distinct subtrees only.

[20] We treat records as unordered tuples from the outset.

Listing 7.2

```
<object param="list">
  <input param="element" type="int"/>
  <object param="next">
    <input param="element" type="int"/>
    <object param="next">
      <input param="next" type="int"/>
    </object>
  </object>
</object>
```

Interestingly the record subtyping rule 7.3 is more generally appropriate for a direct formalization of cyclic data type definitions found in usual object-oriented languages. Consider the definition of possibly infinite lists in the Haskell [144] code fragment 7.9. The null data constructor for the left summand is needed in order to explicitly enable finite lists.

$$\texttt{data}\quad \texttt{List} = \texttt{NULL} \;\mid\; \texttt{NODE}\,\{\texttt{element}::\texttt{Int},\, \texttt{next}::\texttt{List}\} \tag{7.9}$$

The direct counterpart of this data type definition in an object-oriented language is depicted as an UML diagram[21] in Figure 7.1(a). A more appropriate way of modeling lists in an object-oriented programming language is given in Figure 7.1(b). Basically the 1 multiplicity of the navigated association in 7.1(a) is replaced by an `0..1` multiplicity in 7.1(b). It is possible to define lists this way, because in an object-oriented language every object field of complex type that is not constructed is implicitly provided as a null object. This mechanism is exactly the one formalized by subtyping rule 7.3. To see this, note that the lists defined in equation 7.9 and 7.1(a) both have type $\texttt{List}_{\texttt{expl}}$ given in 7.10[22], whereas 7.1(b) has type $\texttt{List}_{\texttt{impl}}$ given in equation 7.11 provided that subtyping rule 7.3 is present.

$$\texttt{List}_{\texttt{expl}} \equiv_{\texttt{DEF}} \mu List.[\texttt{NULL} : \{\}, \texttt{NODE} : \{element \mapsto \texttt{int}, next \mapsto List\}] \tag{7.10}$$
$$\texttt{List}_{\texttt{impl}} \equiv_{\texttt{DEF}} \mu List.\{element \mapsto \texttt{int}, next \mapsto List\} \tag{7.11}$$

7.3.2 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 occurrence types $\mathbb{F}$ and system types $\mathbb{S}$.

[21] The diagram is drawn within the an implementation perspective in the sense of [74][41].
[22] The variant notation is taken from [83].

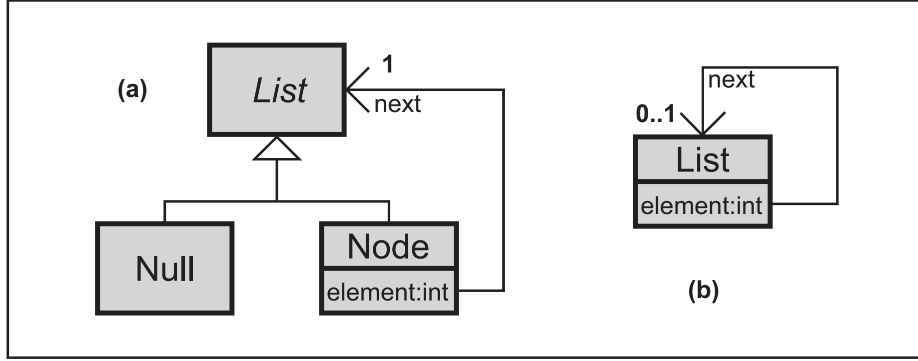

Figure 7.1: UML list definitions. The figure shows to alternative cyclic data models of the list data type.

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.

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

$$\mathbb{W} = \textbf{Label} \nrightarrow\!\!\!\!\!\!\!\!\!\rightarrow (\mathbb{T} \cup \mathbb{P})$$

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

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 7.4 for records equally applies to web signatures. The establishing subtyping rule 7.3 must be slightly modified resulting in rule 7.12, because formal parameters of server page type must always be provided, too.

Subtyping rule 7.13 is standard and states, that server page types are contravariant in the argument type and covariant in the result type.

$$\frac{T_j \notin \mathbb{B}_{primitive} \cup \mathbb{P} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{7.12}$$

$$\frac{\vdash w' < w \qquad \vdash R < R'}{\vdash w \rightarrow R < w' \rightarrow R'} \tag{7.13}$$

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.

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

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

$$\frac{\vdash S_1 < T_1 \quad \vdash S_2 < T_2}{\vdash (S_1, S_2) < (T_1, T_2)} \tag{7.14}$$

Subtyping rule 7.14 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 7.15.

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

$$\frac{T \in \mathbb{E}}{\vdash \circ \; < \; T} \tag{7.15}$$

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 7.16 specifies, that such fragments can play the role of both fragments of outside form and fragments of inside form type.

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

$$\frac{T \in \mathbb{F}}{\vdash \Updownarrow < \; T} \tag{7.16}$$

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

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.

7.4 Type Operators

In the NSP typing rules in chapter 7.6 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 6.2. 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[23] 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.

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

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

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[24] this is specified succinctly in line 3 of the $\odot$ operator definition.

[23] section 7.3.2
[24] Appendix D

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[25]. Consider the following example application of the $\odot$ operator:

$$
\begin{array}{ll}
 & \{l \mapsto \texttt{int}, n \mapsto \{o \mapsto \texttt{int}, p \mapsto \texttt{String}\}\} \qquad (T_1) \\
\odot & \{m \mapsto \texttt{int}, n \mapsto \{o \mapsto \texttt{int}, q \mapsto \texttt{String}\}\} \qquad (T_2)
\end{array}
$$

$$
= \quad \begin{array}{l}
\{ \quad l \mapsto \texttt{int}, \\
\quad\; m \mapsto \texttt{int}, \\
\quad\; n \mapsto \texttt{array of}\{o \mapsto \texttt{int}, p \mapsto \texttt{String}, q \mapsto \texttt{String}\} \\
\}
\end{array} \qquad (T_3)
$$

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. A complex example for the application of the $\odot$ operator in the interplay of subtyping and recursive typing can be found in appendix B.8.

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[26] introduced in chapter 1.

[25] 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.

[26] If desired page array types must be introduced by tag support.

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 7.28 for loop-structures is adequate.

7.5 Environments and Judgments

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 7.50.

$$\Gamma : \textbf{Label} \mathrel{+\!\!\!\!\!\!\mapsto} (\mathbb{T} \cup \mathbb{P}) \quad = \mathbb{W}$$

$$\Delta : \textbf{Label} \mathrel{+\!\!\!\!\!\!\mapsto} \mathbb{P} \quad \subset \mathbb{W}$$

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 7.20 it is prevented that page identifiers may become program parts.

The Core NSP type system relies on several typing judgments:

$$\Gamma \vdash e : \mathbb{T} \cup \mathbb{P} \qquad e \in \textbf{expr}$$

$$\Gamma \vdash n : \mathbb{D} \qquad n \in \textbf{com} \cup \textbf{dynamic}$$

$$\Gamma \vdash c : \mathbb{P} \qquad c \in \textbf{websig-core}$$

$$\Gamma \vdash a : \mathbb{W} \qquad a \in \textbf{actualparams}$$

$$\Gamma, \Delta \vdash s : \mathbb{S} \qquad s \in \textbf{system}$$

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 definition environment.

7.6 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.

In this section the typing rules are presented by starting from basic building blocks to more complex building blocks. In appendix B.6 the rules are ordered with respect to the production rules in appendix B.1 for easy reference.

The typing rule 7.17 allows for extraction of an identifier typing assumption from the typing environment. Rules 7.18 and 7.19 give the types of selected record fields respectively indexed array elements.

$$\frac{(v \mapsto T) \in \Gamma}{\Gamma \vdash v : T} \tag{7.17}$$

$$\frac{\Gamma \vdash e : \{l_i \mapsto T_i\}^{i \in 1 \ldots n} \qquad j \in 1 \ldots n}{\Gamma \vdash e.l_j : T_j} \tag{7.18}$$

$$\frac{\Gamma \vdash e : \texttt{array of } T \qquad \Gamma \vdash i : \texttt{int}}{\Gamma \vdash e[i] : T} \tag{7.19}$$

Typing rule 7.20 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 7.21.

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T \qquad T \in \mathbb{T}}{\Gamma \vdash x := e \ : \ ((\circ, \Updownarrow), \emptyset)} \tag{7.20}$$

$$\frac{}{\Gamma \vdash \boldsymbol{\varepsilon} : ((\circ, \Updownarrow), \emptyset)} \tag{7.21}$$

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 7.22 and 7.23 do not, too.

$$\frac{\Gamma \vdash c : D}{\Gamma \vdash < \texttt{code} > c < /\texttt{code} > \ : \ D} \tag{7.22}$$

$$\frac{\Gamma \vdash d : D}{\Gamma \vdash < /\texttt{code} > d < \texttt{code} > \ : \ D} \tag{7.23}$$

Equally basic as rule 7.20, rule 7.24 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 7.25 the same type as character strings.

$$\frac{d \in \textbf{string}}{\Gamma \vdash d : ((\bullet, \Updownarrow), \emptyset)} \tag{7.24}$$

$$\frac{\Gamma \vdash e : T \quad T \in \mathbb{B}}{\Gamma \vdash < \texttt{expression} > e < /\texttt{expression} > \ : \ ((\bullet, \Updownarrow), \emptyset)} \tag{7.25}$$

Composing user descriptions parts and sequencing programming language parts must follow essentially the same typing rule. In both rule 7.26 and rule 7.27 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 7.4.

$$\frac{\begin{array}{c} d_1, d_2 \in \mathbf{dynamic} \\ \Gamma \vdash d_1 : (L_1, w_1) \quad \Gamma \vdash d_2 : (L_2, w_2) \quad L_1 \sqcup L_2 \downarrow \quad w_1 \odot w_2 \downarrow \end{array}}{\Gamma \vdash d_1 \ d_2 : (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{7.26}$$

$$\frac{\Gamma \vdash c_1 : (L_1, w_1) \qquad \Gamma \vdash c_2 : (L_2, w_2) \qquad L_1 \sqcup L_2 \downarrow \qquad w_1 \odot w_2 \downarrow}{\Gamma \vdash c_1 ; c_2 \ : \ (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{7.27}$$

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 7.28. 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.

$$\frac{\Gamma \vdash e : \mathtt{boolean} \qquad \Gamma \vdash c : (L, w)}{\Gamma \vdash \mathtt{while}\ e\ \mathtt{do}\ c \ : \ (L, w \odot w)} \tag{7.28}$$

$$\frac{\Gamma \vdash e : \mathtt{boolean} \quad \Gamma \vdash c_1 : D_1 \quad \Gamma \vdash c_2 : D_2 \qquad D_1 \sqcup D_2 \downarrow}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ c_1 \mathtt{else}\ c_2 \ : \ D_1 \sqcup D_2} \tag{7.29}$$

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.

$$\frac{}{\Gamma \vdash < \mathtt{submit/} > \ : \ ((\bullet, \Downarrow), \emptyset)} \tag{7.30}$$

$$\frac{T \in \mathbb{B}_{supported}}{\Gamma \vdash \ < \mathtt{input}\ \mathtt{type} = "T" \ \mathtt{param} = "l" / > : ((\bullet, \Downarrow), \{(l \mapsto T)\})} \tag{7.31}$$

$$\frac{}{\Gamma \vdash <\texttt{checkbox}\quad\texttt{param} = "l"/> \ :\ ((\bullet, \Updownarrow), \{(l \mapsto \mathit{boolean})\})} \tag{7.32}$$

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.

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash <\texttt{hidden}\,\texttt{param} = "l" > e </\texttt{hidden}> \ :\ ((\circ, \Downarrow), \{(l \mapsto T)\})} \tag{7.33}$$

The select element may only contain code that generates option elements. Therefore an option element obtains the option type **OP** by rule 7.35 and the select element typing rule 7.34 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 7.34 to construct the correct form type[27].

$$\frac{\Gamma \vdash d : \big((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of } T)\}\big)}{\Gamma \vdash \begin{array}{l} <\texttt{select}\quad\texttt{param} = "l" > \\ \qquad d \\ </\texttt{select}> \ :\ ((\bullet, \Downarrow), \{(l \mapsto \texttt{array of } T)\}) \end{array}} \tag{7.34}$$

$$\frac{\Gamma \vdash v\ :\ T \qquad \Gamma \vdash e : S \quad S \in \mathbb{B} \qquad l \in \mathbf{Label}}{\Gamma \vdash \begin{array}{l} <\texttt{option}> \\ \qquad <\texttt{value}> v </\texttt{value}> \\ \qquad <\texttt{label}> e </\texttt{label}> \\ </\texttt{option}> : ((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of } T)\}) \end{array}} \tag{7.35}$$

The object element is a record construction facility. The enclosed document fragment's layout type lasts after application of typing rule 7.36, 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.

$$\frac{\Gamma \vdash d : (L, w)}{\Gamma \vdash <\texttt{object}\,\texttt{param} = "l" > d </\texttt{object}> \ :\ (L, \{(l \mapsto w)\})} \tag{7.36}$$

The form typing rule 7.37 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

[27] This way no new kind of judgment has to be introduced for select menu options.

be allowed to occur inside a form. Then the rule 7.37 specifies that the form is a vizible element that must not contain inside another form.

$$\frac{\Gamma \vdash l : w \rightarrow \Box \qquad \Gamma \vdash d : ((e, \Downarrow), v) \qquad \vdash v < w}{\Gamma \vdash < \texttt{form} \quad \texttt{callee} = \text{"}l\text{"} > d < /\texttt{form} > : ((e, \Uparrow), \emptyset)} \tag{7.37}$$

Now the layout structuring elements, i.e. lists and tables, are investigated. The corresponding typing rules 7.38 to 7.42 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 7.38. Only documents with list layout type may become part of a list. A well-typed list is a vizible element. The rules 7.40 to 7.42 work analogously for tables.

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{< \texttt{li} > d < /\texttt{li} > \ : \ ((\mathbf{LI}, F), w)} \tag{7.38}$$

$$\frac{\Gamma \vdash d : ((\mathbf{LI} \vee \circ, F), w)}{< \texttt{ul} > d < /\texttt{ul} > \ : \ ((\bullet, F), w)} \tag{7.39}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{< \texttt{td} > d < /\texttt{td} > \ : \ ((\mathbf{TD}, F), w)} \tag{7.40}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TD} \vee \circ, F), w)}{< \texttt{tr} > d < /\texttt{tr} > \ : \ ((\mathbf{TR}, F), w)} \tag{7.41}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TR} \vee \circ, F), w)}{< \texttt{table} > d < /\texttt{table} > \ : \ ((\bullet, F), w)} \tag{7.42}$$

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 7.45 makes it possible that several actual parameter elements uniquely provide the parameters for a server side call. Rule 7.43 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.

$$\frac{\Gamma \vdash l : w \rightarrow D \qquad \Gamma \vdash as : v \qquad \vdash v < w}{\Gamma \vdash < \texttt{call} \quad \texttt{callee} = \text{"}l\text{"} > as < /\texttt{call} > \ : \ D} \tag{7.43}$$

$$\frac{}{\Gamma \vdash \varepsilon_{\texttt{act}} : \emptyset} \tag{7.44}$$

$$\frac{\Gamma \vdash as : w \qquad \Gamma \vdash e : T \qquad l \notin (dom\ w)}{\Gamma \vdash \begin{array}{l} <\texttt{actualparam}\ \ \texttt{param} = "l" > \\ \qquad e \\ </\texttt{actualparam}> as\ :\ w \cup \{(l \mapsto T)\} \end{array}} \tag{7.45}$$

With the typing rule 7.46 and 7.49 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 7.47 and 7.49 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 7.48 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.

$$\frac{\Gamma \vdash d : D \qquad d \in \textbf{dynamic}}{\Gamma \vdash <\texttt{include}> d </\texttt{include}> : \emptyset \rightarrow D} \tag{7.46}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, \Updownarrow \vee \Uparrow), \emptyset) \qquad t \in \textbf{strings} \qquad d \in \textbf{dynamic}}{\Gamma \vdash \begin{array}{l} <\texttt{html}> \\ \qquad <\texttt{head}><\texttt{title}> t </\texttt{head}></\texttt{title}> \\ \qquad <\texttt{body}> d </\texttt{body}> \\ </\texttt{html}> : \emptyset \rightarrow \Box \end{array}} \tag{7.47}$$

$$\frac{\Gamma \vdash l : T \qquad \Gamma \vdash c : w \rightarrow D \qquad l \notin (dom\ w)}{\Gamma \backslash (l \mapsto T) \vdash \begin{array}{l} <\texttt{param}\ \ \texttt{name} = "l"\ \ \texttt{type} = "T" /> \\ c : (w \cup \{(l \mapsto T)\}) \rightarrow D \end{array}} \tag{7.48}$$

$$\frac{\Gamma \vdash l : P \qquad \Gamma \vdash c : P \qquad c \in \textbf{websig-core}}{\Gamma \backslash (l \mapsto P), \{(l \mapsto P)\} \vdash <\texttt{nsp}\ \ \texttt{name} = "l" > c </\texttt{nsp}>\ :\ \Diamond} \tag{7.49}$$

A server page core can become a well-typed server page by rule 7.49. 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 7.50 specifies system compatibility. Rule 7.51 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.

$$\frac{\begin{array}{c} s_1, s_2 \in \textbf{system} \quad (dom\ \Delta_1) \cap (dom\ \Delta_2) = \emptyset \\ ((dom\ \Gamma_2) \lhd\ \Delta_1) < ((dom\ \Delta_1) \lhd\ \Gamma_2) \\ ((dom\ \Gamma_1) \lhd\ \Delta_2) < ((dom\ \Delta_2) \lhd\ \Gamma_1) \\ \Gamma_1, \Delta_1 \vdash s_1 : \diamond \qquad \Gamma_2, \Delta_2 \vdash s_2 : \diamond \end{array}}{((dom\ \Delta_2) \lhd \Gamma_1) \cup ((dom\ \Delta_1) \lhd \Gamma_2)\ ,\ \Delta_1 \cup \Delta_2\ \vdash s_1\ s_2\ :\ \diamond} \tag{7.50}$$

$$\frac{\begin{array}{c} \Gamma \in \mathbb{R} \\ (dom\ \Delta) \cap bound(s) = \emptyset \\ \Gamma, \Delta \vdash s : \diamond \end{array}}{\Gamma, \Delta \vdash s : \surd} \tag{7.51}$$

Typing rule 7.51 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 7.6.1 *Core NSP type checking is decidable.*

Proof(7.6.1): 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 7.37 is applied.□