The Core NSP Type System

Dirk Draheim
Institute of Computer Science
Freie Universität Berlin
draheim@acm.org

Gerald Weber
Department of Computer Science
University of Auckland
g.weber@cs.auckland.ac.nz

ABSTRACT

A good deal of software development and maintenance costs for Web applications stem from the fact that the untyped, flat message concept of the CGI interface has its footprint in the commonly used Web programming models and Web development technologies. Still it is necessary to reengineer large legacy Web applications that have been developed without the help of an improved Web technology. Current web application frameworks offer support to deal with client page type errors dynamically, however, no static type checks are provided by these tools. Furthermore, they do not allow for detecting potential web page description errors at compile time. In this paper, 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.

1. INTRODUCTION

In [6] a strongly typed server pages technology NSP (Next Server Pages) has been proposed. The NSP type system has also been exploited in the design and implementation of the reverse engineering tool JSPick [5]. JSPick can recover the design and type structure of a web presentation layer that is based on server pages technology.

In this paper the type system of NSP is defined formally. 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. The NSP contributions target stability and reusability of server pages based systems. The findings are programming language independent.

Important contributions of NSP has been the following:

Parameterized server pages. 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. A server page signature is termed web signature in the sequel.

Statically ensured client page type and description safety. 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 are always valid with respect to a defined client page description language.

Support for complex types in writing forms. New structured tags are offered for gathering arrays and objects of user defined types.

Functional decomposition of server pages In NSP a server-side call to a server page is designed as a parameter-passing procedure call, too. This helps decoupling architectural issues and implementing design patterns.

Higher order server pages. Server pages may be actual form parameters. This enables improved web-based application architecture and design.

Exchange of objects across the web user agent. Server side programmed objects may be actual form parameters and therefore passed to client pages and back, either as messages or virtually as objects.

The NSP concepts are reusable, programming language independent results. They must be amalgamated with a concrete programming language. 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 [6] 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 the considered programming language in any way, for example in the case of Java the Servlet session facility for state handling is available as a matter of course.

The NSP coding rules [6] 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 am-

biguity. This paper formally defines the type system of Core NSP, which is the amalgamation of NSP concepts with a minimal imperative programming language. This enables precise reasoning about the NSP concepts.

2. CORE NSP GRAMMAR

In this section the abstract syntax of Core NSP programs is specified[1]. 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
param ::= <param name="id" type="param-type"/>
websig-core ::= webcall | include
webcall ::= <html> head body </html>
head ::= <head><title> strings </title></head>
strings ::= ε | string strings
body ::= <body> dynamic </body>
include ::= <include> dynamic </include>
param-type ::= t ∈ 𝕋 ∪ ℙ
supported-type ::= t ∈ 𝔹_supported
```

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

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.

```
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 system. We delve on this in section 3.

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

[1]Nonterminals are underlined. Terminals are not emphasized. Every nonterminal corresponds to a syntactic category. In the grammar a syntactic category is depicted in bold face.

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, denoted by $\varepsilon_{\mathtt{act}}$.

```
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 tags in order to express the integration of active code parts with layout parts. 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 ::= <expr> expr </expr>
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. Core NSP, in contrast, 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 section 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 example. 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 (SGML) supports the notions of exclusion and inclusion exception. Indeed the SGML exception notation does not add to the expressive power of SGML [26], because an SGML expression that includes exceptions can be translated into an extended context free grammar [10]. The transformation algorithm given in [10] produces $2^{2^{|\mathbb{N}|}}$ 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. Again excluding wrong documents already by abstract syntax amounts to duplicate production rules for the static parts that may be contained in dynamic parts.

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. The Core NSP types are given by a family of recursively defined type sets. 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 adopting the Barbara Liskov principle [11]. 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 judgements 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. 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. The types introduced so far and the type variables $\mathbb{V}$ together form the set of programming language types $\mathbb{T}$.

$$\mathbb{T} = \mathbb{B} \cup \mathbb{V} \cup \mathbb{A} \cup \mathbb{R} \cup \mathbb{Y}$$
$$\mathbb{B} = \mathbb{B}_{primitive} \cup \mathbb{B}_{supported}$$
$$\mathbb{B}_{primitive} = \{\texttt{int}, \texttt{float}, \texttt{boolean}\}$$
$$\mathbb{B}_{supported} = \{\texttt{int}, \texttt{Integer}, \texttt{String}\}$$
$$\mathbb{V} = \{X, Y, Z, \ldots\} \cup \{\texttt{Person}, \texttt{Customer}, \texttt{Article}, \ldots\}$$

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. Judgment rule 2 is the preserving subtyping[2] rule for array types.

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

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

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

The usage of some Z Notation [20] for record types will ease writing type operator definitions and typing rules later on: a record type is a finite partial function from a set of labels to the set of programming language types.

$$\mathbb{R} = \mathbf{Label} \twoheadrightarrow \mathbb{T}$$

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

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

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 [3] or objects [1]. But there is no contradiction, because these rules describe hierarchies of feature support and we just specify another phenomenon: rule 3

[2]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 judgements that deal with defining the effects of the various type constructors on the subtype relation.

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. 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.

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

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

We have chosen to handle recursive types in an equi-recursive way [7]. Core NSP types represent finite trees or possibly infinite regular trees [4]. Type equivalence is not explicitly defined, it is given implicitly by the subtype relation. The subtype relation is defined as the greatest fixpoint of a monotone generating function on the universe of type trees [7]. The Core NSP subtyping rules provide an intuitive description of this generating function. Thereby the subtyping rules for left folding and right folding (5) provide the desired recursive subtyping. Beyond this only one further subtyping rule is needed, namely the rule 6 for introducing reflexivity.

$$\frac{}{\vdash T < T} \tag{6}$$

4.2 Server Page Types

In order to formalize the NSP coding rules the type system of Core NSP comprises server page types $\mathbb{P}$, web signatures $\mathbb{W}$, a single complete web page type $\square \in \mathbb{C}$, document 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.

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

$$\mathbb{W} = \mathbf{Label} \twoheadrightarrow (\mathbb{T} \cup \mathbb{P}) \qquad \mathbb{C} = \{\square\}$$

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 7, because formal parameters of server page type must always be provided, too.

Subtyping rule 8 is standard and states, that server page types are contravariant in the argument type and covariant in the result type.

$$\frac{T_j \notin \mathbb{B}_{primitive} \cup \mathbb{P} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{7}$$

$$\frac{\vdash w' < w \qquad \vdash R < R'}{\vdash w \to R < w' \to R'} \tag{8}$$

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.

$$\mathbb{D} = \mathbb{L} \times \mathbb{W} \qquad \mathbb{L} = \mathbb{E} \times \mathbb{F}$$

$$\frac{\vdash S_1 < T_1 \quad \vdash S_2 < T_2}{\vdash (S_1, S_2) < (T_1, T_2)} \tag{9}$$

Subtyping rule 9 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 10.

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

$$\frac{T \in \mathbb{E}}{\vdash \circ\ <\ T} \tag{10}$$

The form occurrence types further constrain 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 11 specifies, that such fragments can play the role of both fragments of outside form and fragments of inside form type.

$$\mathbb{F} = \{\ \Downarrow, \Uparrow, \Updownarrow\ \} \qquad \mathbb{S} = \{\ \diamond, \surd\ \}$$

$$\frac{T \in \mathbb{F}}{\vdash \Updownarrow < T} \tag{11}$$

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. If applied to an array the operater lets the type unchanged, otherwise it yields the respective array type.

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

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.

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

$$\begin{array}{lll} \bot & \boldsymbol{,if} & \exists(l_1 \mapsto T_1) \in w_1 \bullet\ \exists(l_2 \mapsto T_2) \in w_2 \bullet \\ & & l_1 = l_2\ \wedge\ P_1 \in \mathbb{P}\ \wedge\ P_2 \in \mathbb{P} \\ \bot & \boldsymbol{,if} & \exists(l_1 \mapsto T_1) \in w_1 \bullet\ \exists(l_2 \mapsto T_2) \in w_2 \bullet \\ & & l_1 = l_2\ \wedge\ T_1 \sqcup T_2\ undefined \end{array}$$

$$\begin{array}{lll} & (\mathtt{dom}\ w_2) \lhd w_1\ \cup\ (\mathtt{dom}\ w_1) \lhd w_2 & \\ \cup & l \mapsto (T_1 \sqcup T_2)*\ \mid & \\ & (l \mapsto T_1) \in w_1\ \wedge\ (l \mapsto T_2) \in w_2 & \boldsymbol{,else} \end{array}$$

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 [20] this is specified succinctly in 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 server page: 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 server page 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 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.

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.

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 23 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 45.

$$\begin{array}{lll} \Gamma : \mathbf{Label} \twoheadrightarrow (\mathbb{T} \cup \mathbb{P}) & = \mathbb{W} \\ \Delta : \mathbf{Label} \twoheadrightarrow \mathbb{P} & \subset \mathbb{W} \end{array}$$

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 en-

tity is allowed; e.g. in the statement typing rule 15 it is prevented that page identifiers may become program parts. The Core NSP type system relies on several typing judgements:

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

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. 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 judgements 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 judgements have 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. Compared to a declarative version extra premises are needed in some of the typing rules, in some premises slightly bit more complex type patterns have to be used. However in the Core NSP type system these extra complexity fosters understandability. The typing rule 12 allows for extraction of an identifier typing assumption from the typing environment. Rules 13 and 14 give the types of selected record fields respectively indexed array elements.

$$\frac{(v \mapsto T) \in \Gamma}{\Gamma \vdash v : T} \tag{12}$$

$$\frac{\Gamma \vdash e : \{l_i \mapsto T_i\}^{i \in 1 \ldots n} \qquad j \in 1 \ldots n}{\Gamma \vdash e.l_j : T_j} \tag{13}$$

$$\frac{\Gamma \vdash e : \texttt{array of}\ T \qquad \Gamma \vdash i : \texttt{int}}{\Gamma \vdash e[i] : T} \tag{14}$$

Typing rule 15 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 16.

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T \qquad T \in \mathbb{T}}{\Gamma \vdash x := e \ : \ ((\circ, \Updownarrow), \emptyset)} \tag{15}$$

$$\frac{}{\Gamma \vdash \varepsilon : ((\circ, \Updownarrow), \emptyset)} \tag{16}$$

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 17 and 18 do not, too.

$$\frac{\Gamma \vdash c : D}{\Gamma \vdash <\texttt{code}> c <\texttt{/code}> \ : \ D} \tag{17}$$

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

Rule 19 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 20 the same type as character strings.

$$\frac{d \in \textbf{string}}{\Gamma \vdash d : ((\bullet, \Updownarrow), \emptyset)} \tag{19}$$

$$\frac{\Gamma \vdash e : T \quad T \in \mathbb{B}}{\Gamma \vdash <\texttt{expr}> e <\texttt{/expr}> \ : \ ((\bullet, \Updownarrow), \emptyset)} \tag{20}$$

Composing user descriptions parts and sequencing programming language parts must follow essentially the same typing rule. In both rule 21 and rule 22 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.

$$\frac{\begin{array}{c} d_1, d_2 \in \textbf{dynamic} \\ \Gamma \vdash d_1 : (L_1, w_1) \quad \Gamma \vdash d_2 : (L_2, w_2) \\ L_1 \sqcup L_2 \downarrow \quad w_1 \odot w_2 \downarrow \end{array}}{\Gamma \vdash d_1 \ d_2 : (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{21}$$

$$\frac{\begin{array}{cc} \Gamma \vdash c_1 : (L_1, w_1) & \Gamma \vdash c_2 : (L_2, w_2) \\ L_1 \sqcup L_2 \downarrow & w_1 \odot w_2 \downarrow \end{array}}{\Gamma \vdash c_1 ; c_2 \ : \ (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{22}$$

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 23. 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.

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash c : (L, w)}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \ : \ (L, w \odot w)} \tag{23}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash c_1 : D_1 \quad \Gamma \vdash c_2 : D_2 \quad D_1 \sqcup D_2 \downarrow}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1 \texttt{else}\ c_2\ :\ D_1 \sqcup D_2} \tag{24}$$

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.

$$\frac{}{\Gamma \vdash <\texttt{submit/}> : ((\bullet, \Downarrow), \emptyset)} \tag{25}$$

$$\frac{T \in \mathbb{B}_{supported}}{\Gamma \vdash \begin{array}{l} <\texttt{input type} = "T"\ \texttt{param} = "l"/> : \\ ((\bullet, \Downarrow), \{(l \mapsto T)\}) \end{array}} \tag{26}$$

$$\frac{}{\Gamma \vdash \begin{array}{l} <\texttt{checkbox}\quad \texttt{param} = "l"/> : \\ ((\bullet, \Updownarrow), \{(l \mapsto \mathit{boolean})\}) \end{array}} \tag{27}$$

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.

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \begin{array}{l} <\texttt{hidden param} = "l" > e < /\texttt{hidden} > : \\ ((\circ, \Downarrow), \{(l \mapsto T)\}) \end{array}} \tag{28}$$

The select element may only contain code that generates option elements. Therefore an option element obtains the option type **OP** by rule 30 and the select element typing rule 29 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 29 to construct the correct form type.

$$\frac{\Gamma \vdash d : \ (\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of}\ T)\}}{\Gamma \vdash \begin{array}{l} <\texttt{select}\quad \texttt{param} = "l" > d \\ </\texttt{select}> \ :\ ((\bullet, \Downarrow), \{(l \mapsto \texttt{array of}\ T)\}) \end{array}} \tag{29}$$

$$\frac{\Gamma \vdash v\ :\ T \qquad \Gamma \vdash e : S \quad S \in \mathbb{B} \qquad l \in \mathbf{Label}}{\Gamma \vdash \begin{array}{l} <\texttt{option}> \\ \quad <\texttt{value}> v </\texttt{value}> \\ \quad <\texttt{label}> e </\texttt{label}> \\ </\texttt{option}> : ((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of}\ T)\}) \end{array}} \tag{30}$$

The object element is a record construction facility. The enclosed document fragment's layout type lasts after application of typing rule 31, 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.

$$\frac{\Gamma \vdash d : (L, w)}{\Gamma \vdash \begin{array}{l} <\texttt{object param} = "l" > d </\texttt{object}> : \\ (L, \{(l \mapsto w)\}) \end{array}} \tag{31}$$

The form typing rule 32 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 server page of given signature. Furthermore the form content's must be allowed to occur inside a form. Then the rule 32 specifies that the form is a vizible element that must not contain inside another form.

$$\frac{\Gamma \vdash l : w \to \square \qquad \Gamma \vdash d : ((e, \Downarrow), v) \qquad \vdash v < w}{\Gamma \vdash <\texttt{form}\quad \texttt{callee} = "l" > d </\texttt{form}> : ((e, \Uparrow), \emptyset)} \tag{32}$$

Now the layout structuring elements, i.e. lists and tables, are investigated. The corresponding typing rules 33 to 37 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 33. Only documents with list layout type may become part of a list. A well-typed list is a vizible element. The rules 35 to 37 work analogously for tables.

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{\Gamma \vdash <\texttt{li}> d </\texttt{li}> \ :\ ((\mathbf{LI}, F), w)} \tag{33}$$

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

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

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

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

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 40 makes it possible that several actual parameter elements uniquely provide the parameters for a server side call. Rule 38 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.

$$\frac{\Gamma \vdash l : w \to D \qquad \Gamma \vdash as : v \qquad \vdash v < w}{\Gamma \vdash <\texttt{call}\quad \texttt{callee} = "l" > as </\texttt{call}> \ :\ D} \tag{38}$$

$$\frac{}{\Gamma \vdash \varepsilon_{\texttt{act}} : \emptyset} \tag{39}$$

$$\frac{\Gamma \vdash as : w \qquad \Gamma \vdash e : T \qquad l \notin (\mathit{dom}\ w)}{\Gamma \vdash \begin{array}{l} <\texttt{actualparam}\quad \texttt{param} = "l" > e \\ </\texttt{actualparam}> as \ :\ w \cup \{(l \mapsto T)\} \end{array}} \tag{40}$$

With the typing rule 41 and 44 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 42 and 44 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 43 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.

$$\frac{\Gamma \vdash d : D \qquad d \in \textbf{dynamic}}{\Gamma \vdash < \texttt{include} > d < /\texttt{include} > : \emptyset \rightarrow D} \tag{41}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, \Updownarrow \vee \Uparrow), \emptyset)\ t \in \textbf{strings}\ d \in \textbf{dynamic}}{\Gamma \vdash \begin{array}{l} < \texttt{html} > \\ \quad < \texttt{head} > \\ \quad\quad < \texttt{title} > t < /\texttt{title} > \\ \quad < /\texttt{head} > \\ \quad < \texttt{body} > d < /\texttt{body} > \\ < /\texttt{html} > : \emptyset \rightarrow \square \end{array}} \tag{42}$$

$$\frac{\Gamma \vdash l : T \qquad \Gamma \vdash c : w \rightarrow D \qquad l \notin (dom\ w)}{\begin{array}{l} \Gamma \backslash (l \mapsto T) \vdash \\ < \texttt{param}\ \ \texttt{name} = "l"\ \ \texttt{type} = "T" / > \\ c : (w \cup \{(l \mapsto T)\}) \rightarrow D \end{array}} \tag{43}$$

$$\frac{\Gamma \vdash l : P \qquad \Gamma \vdash c : P \qquad c \in \textbf{websig-core}}{\begin{array}{l} \Gamma \backslash (l \mapsto P), \{(l \mapsto P)\} \vdash \\ < \texttt{nsp}\ \ \texttt{name} = "l" > c < /\texttt{nsp} > \ :\ \diamond \end{array}} \tag{44}$$

A server page core can become a well-typed server page by rule 44. 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 45 specifies system compatibility. Rule 46 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.

$$\frac{\begin{array}{c} s_1, s_2 \in \textbf{system} \quad (dom\ \Delta_1) \cap (dom\ \Delta_2) = \emptyset \\ ((dom\ \Gamma_2) \lhd\ \Delta_1) < ((dom\ \Delta_1) \lhd\ \Gamma_2) \\ ((dom\ \Gamma_1) \lhd\ \Delta_2) < ((dom\ \Delta_2) \lhd\ \Gamma_1) \\ \Gamma_1, \Delta_1 \vdash s_1 : \diamond \qquad \Gamma_2, \Delta_2 \vdash s_2 : \diamond \end{array}}{\begin{array}{l} ((dom\ \Delta_2) \lhd \Gamma_1) \cup ((dom\ \Delta_1) \lhd \Gamma_2)\ ,\ \Delta_1 \cup \Delta_2\ \vdash \\ s_1 s_2\ :\ \diamond \end{array}} \tag{45}$$

$$\frac{\begin{array}{c} (dom\ \Delta) \cap bound(s) = \emptyset \\ \Gamma, \Delta \vdash s : \diamond \qquad \Gamma \in \mathbb{R} \end{array}}{\Gamma, \Delta \vdash s : \surd} \tag{46}$$

Typing rule 46 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 32 is applied.□

8. RELATED WORK

WASH/HTML is a embedded domain specific language for dynamic XML coding in the functional programming language Haskell, which is given by combinator libraries [23][24]. In [24] 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. In the XHTML DTD [21] exceptions only occur as comments - in XML DTDs no exception mechanism is available - however these comments become normative status in the corresponding XHMTL standard [22]; they are called element prohibitions. In [18][2] it is shown that the normative element prohibitions of the XHMTL standard [22] can be statically checked by employing flow analysis [15][17][16].

There are a couple of other projects for dynamic XML generation, that garuantee some level of user interface description language safety, e.g. [8][9][12]. We delve on some further representative examples. In [25] 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 [14] garuantee well-formedness of XML documents. The small functional programming language XMλ [19] is based on XML documents as basic datatypes and is designed to ensure full XML validity [13].

9. CONCLUSION

The best practice of the proven 3GL programming languages – to define a programming system as the interplay of statically typed components – has not yet been adopted to the

development of Web interfaces. With respect to Software design, this problem is tackled by the introduction of proprietary concepts in several commercial Web technologies, like the concept of object wrappers for the form data in the SAP technology BSP (Business Server Pages). Dealing with type errors is supported by web applications frameworks like Struts or IBM Websphere, too, however, only dynamic concepts are offered.

There are several initiatives that propose a statically typed approach to web application development. With NSP, web development with server pages is addressed. A precise description of the type system of NSP is desired, because it (i) can be used as the specification for implementations of NSP concepts, (ii) allows for precise reasoning about web interaction and therefore (iii) deepens the understanding of the interplay between web pages, forms and Web scripts. Therefore, in this paper the core type system of NSP has been given as a Per Martin-Löf style type system.

10. REFERENCES

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

[2] C. Brabrand, A. Møller, and M. I. Schwartzbach. Static validation of dynamically generated HTML. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering*. ACM, 2001.

[3] L. Cardelli. Type systems. In *Handbook of Computer Science and Engineering*. CRC Press, 1997.

[4] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[5] D. Draheim, E. Fehr, and G. Weber. JSPick - A Server Pages Design Recovery Tool. In *Proceedings of CSMR 2003 - 7th European Conference on Software Maintenance and Reengineering*. IEEE Press, 2003.

[6] D. Draheim and G. Weber. Strongly Typed Server Pages. In *Proceedings of The Fifth Workshop on Next Generation Information Technologies and Systems*, LNCS, pages 29–44. Springer, June 2002.

[7] V. Gapayev, M. Y. Levin, and B. C. Pierce. Recursive Subtyping Revealed. In *International Conference on Functional Programming*, 2000. To appear in Journal of Functional Programming.

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

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

[10] P. Kilpeläinen and D. Wood. SGML and Exceptions. Technical Report HKUST-CS96-03, Department of Computer Science, University of Helsinki, 1996.

[11] B. Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23(5), May 1988.

[12] E. Meijer. Server-side Scripting in Haskell. *Journal of Functional Programming*, 2000.

[13] E. Meijer and M. Shields. XMλ - A Functional Language for Constructing and Manipulating XML Documents. 2000. http://www.cse.ogi.edu/∼mbs, Draft.

[14] E. Meijer and D. van Velzen. Haskell Server Pages - Functional Programming and the Battle for the Middle Tier. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001.

[15] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

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

[17] J. Palsberg and M. I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[18] A. Sandholm and M. Schwartzbach. A type system for dynamic web documents. In T. Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 290–301. ACM Press, 2000.

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

[20] J. Spivey. *The Z Notation*. Prentice Hall, 1992.

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

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

[23] P. Thiemann. Modeling HTML in Haskell. In *Practical Applications of Declarative Programming (PADL '00)*, LNCS, January 2000.

[24] P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4):435–468, July 2002.

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

[26] D. Wood. Standard generalized markup language: Mathematical and philosophical issues. In J. van Leeuwen, editor, *Computer Science Today. Recent Trends and Developments*, LNCS, pages 344–365. Springer, 1995.