HSS: a Compiler for Cascading Style Sheets

Manuel Serrano

Sophia Méditerranée
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex – France
Manuel.Serrano@inria.fr

Abstract

This article presents HSS[1], a compiler for CSS. It is first argued that generating CSS improves portability and maintainability of CSS files. This claim is supported by realistic examples. Then, the HSS compilation algorithm is presented. It is simple enough to be easily adapted to most web development kits.

HSS can be used as a stand-alone HSS-to-CSS compiler in the goal of enriching CSS with user defined variables, functions, and element types. It can also be used with the Hop web development kit in which case, working hand in hand with the Hop programming language, it can be used to implement *skinning* or *theming* of web applications.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Design languages

General Terms Design, Languages

Keywords Web programming, Functional languages, Scheme, CSS

1. Introduction

Cascading Style Sheets (henceforth CSS) is used to express formatting in HTML and XHTML. It is a declarative language that cannot express computation. The correctness of CSS files can be checked statically by CSS parsers (named *validators* in the web jargon). The W3C for example, provides one on-line (see `http://jigsaw.w3.org/css-validator/`).

CSS is essential for developing Web user interface (henceforth UI). It is used for fine graphical tunings as well as for managing UIs global layouts. As promoted by standard engineering methodologies, CSS enforces a strong separation between the implementation of the application and its UI.

CSS is defined by a W3C specification [4]. A revised version is in preparation (CSS level 3) that supports additional graphical tunings such as opacity, shaded boxes, or round borders. Few browsers already support CSS level 3 but most browsers support various extensions of CSS level 2 for providing similar fancy graphical features (for instance, Mozilla extensions are described at `https://developer.mozilla.org`).

Two restrictions of CSS have motivated the present work.

- The diversity of extensions web browsers support makes it difficult to write *portable* CSS files. Extensions are so common that they are even anticipated and acknowledged by the W3C's specification (see Section *Vendor-specific extensions* of CSS2.1). Extensions are tautologically not standardized and thus yield to portability issues. Frequently, browsers support similar graphical tunings but expressed by different syntaxes. For instance, both Firefox and Safari support round borders but the top left radius is called `-moz-border-radius-topLeft` by Firefox and `-webkit-border-top-left-radius` by Safari! These discrepancies have two negative effects. First, they clutter portable CSS files with redundant but vendor-specific declarations. Second, they impose to the programmers the burden of being aware and familiar with all browsers specificities.
- Users cannot declare new elements. Although consistent with static validation, this is inadequate with current web development trends where many JavaScript libraries abstract upon HTML to provide new sets of widgets and graphical facilities but that are forced to unveil private implementation details because CSS cannot be extended. For instance, the JQuery's documentation (see `http://docs.jquery.com`) teaches us that the *datepicker* widget of the version 1.3.2 is implemented with a HTML `DIV` of the class `ui-datepicker`.

Our contribution, HSS, a compiler for CSS, improves over these two aspects. It brings computing facilities to CSS and it allows users to extend CSS. A HSS file is composed of a list of declarations and rules. Declarations define new HTML elements, new property attributes, new property functions, or variables. Rules are CSS rules where property values can be dynamically computed. The HSS compiler can be used off-line to statically translate HSS source files into CSS files that can be used by any HTML documents, independently of any particular web programming environment. HSS files can also be used on-line by Hop, a development kit for the diffuse Web, that embeds HSS in its runtime system [1, 9]. HSS actually stands for Hop-CSS.

Section 2 presents the minimum CSS background needed to understand the paper. Section 3 presents the HSS language. It shows examples that illustrate the HSS extensions. Section 4 presents the compilation process that translates a HSS source file into a CSS file. Section 5 presents future work. Section 6 compares HSS with other systems that generate CSS files.

2. Background

In the mid 90s the web first started with plain HTML documents that were used to express both structure and visual aspects. In order

[1] This work is supported in part by the French ANR agency, grant `ANR-09--EMER-009-01`.

PPDP'10, July 26–28, 2010, Hagenberg, Austria.
Copyright © 2010 ACM 978-1-4503-0132-9/10/07...$10.00

to simplify authoring and site maintenance, the structuring and the presentation of the documents have then been separated: nowadays HTML is used for structuring and CSS (*Cascading Style Sheets*) is used to attach visual styles. In this section we present a brief CSS tutorial. Readers familiar with CSS may skip this section.

A web page is described by a HTML tree whose nodes have children and attributes such identifiers, classes, or actions to be executed when the mouse flies over them. CSS is a declarative language that let programmers specify the graphical rendering of HTML nodes. For instance, a CSS declaration may specify that a node is red, that it uses a large bold font, and that it is decorated with a dotted blue border.

A CSS file is composed of CSS declarations which have two parts:

1. a pattern, also known as a *selector*, that distinguishes the nodes on which the declaration applies.
2. a list of *properties* that express the graphical configurations.

The syntax of CSS declarations mimics those of C structures. In the following declaration:

```
div.example {
  color: blue;
}
```

"`div.example`" is the selector and "`color`" is the specified property. This declaration reads as follows: all the HTML `div` elements (a `div` is a plain box) that belong to the `example` class are painted in blue. The following HTML snippet shows how such an element can be declared:

```
<DIV class="example" id="ex1">
  This is an example
 </DIV>
```

Using the Hop programming language, one would write:

```
(<DIV> :class "example" :id "ex1"
   "This is an example")
```

Various selectors allow CSS declarations to select nodes. In addition to the *class* selector presented above, three other selectors are frequently used all along this paper:

- `#`: the *identifier* selector that filters elements according to their identifier. For instance, the selector

  ```
  div#ex1 { ... }
  ```

  matches exactly the unique element whose identifier is "`ex1`" (HTML demands uniqueness of identifiers).

- `[`*attr*`=`*val*`]`: the *attribute* selector that filters elements according to their attribute values. For instance, the selector

  ```
  div[hssclass=myhssclass] { ... }
  ```

  matches all the elements whose attribute `hssclass` is the string of characters "`myhssclass`". Such an element can be created with:

  ```
  (<SPAN> :hssclass "myhssclass" ....)
  ```

- `:hover`: a *virtual* selector that matches only when the mouse flies over the elements matching the rest of the selector. For instance the declaration:

  ```
  div.example:hover {
    color: red;
  }
  ```

  can be used to turn to red `div.example` elements when the mouse flies over them. Using the two declarations produces a graphical animation. The `div` elements of the class "`example`" are all blue but the one under the mouse which is red.

The selectors, may range from simple element such as above to rich contextual patterns. If all conditions in the pattern are true for a certain element, the selector matches the element. Contextual patterns express properties of part of documents. For instance, the selector of the following declaration:

```
div.example span.important { ... }
```

matches `span` elements of the class `important` whose ancestor (direct or not in the HTML tree) is a `div` of the class `example`. In the following example:

```
(<DIV> :class "tutorial"
   (<DIV> :class "example"
      (<SPAN> :class "important" :id "span1"
         (<SPAN> :class "important" :id "span2")))
   (<SPAN> :class "important" :id "span3"))
```

only the elements `span1` and `span2` are selected by the pattern. The element `span3` does not match because its only ancestor is a `div` of the class "`tutorial`", as opposed to "`example`".

An element may match several selectors. For instance, if we add the following declaration to our previous example:

```
span.important {
  color: green;
}
```

all the `span` elements of the class "`important`" that are children of `div` of the class "`example`" matches the two selectors. To resolve this ambiguity, which means here choosing between green and blue for painting the `span`, CSS resorts on a normalized algorithm that selects amongst all the matching rules the *most specific* one.

Readers keen to understand the full technical details and subtleties of CSS may refer to the W3C specifications [4].

3. The HSS language

HSS is both a declarative language built on top of CSS and a compiler that translates the HSS language into CSS. Any correct CSS file is a correct HSS source file. Additionally, HSS supports four syntactic extensions:

- `define-hss-property`, which defines new CSS properties;
- `define-hss-function`, which defines new CSS functions;
- `define-hss-type`, which defines new CSS elements;
- `$-form`, which allows arbitrary dynamic computations.

The formal syntax of HSS is described by the H syntax presented in Figures 1, 2, 3, 4, 5 and 6.

H	::=	X	*HSS definition*
	|	`$expr`	*Hop expression*
	|	$S_1, \ldots, S_n \{D_1; \ldots D_m;\}$	*rule set*
X	::=	H_{prop}	*property definition*
	|	H_{fun}	*function definition*
	|	H_{type}	*type definition*

Figure 1. Grammar for HSS.

The definitions of H_{prop} H_{fun}, and H_{type} will be presented in the rest of this section along with examples.

The actual concrete syntax of child of operator is only the empty string. For the sake of readability of the paper, we have added here the terminal <.

S	::=	S'_1 *op* ... *op* S'_k	*compound selector*
S'	::=	IDENT	*element type*
	|	S' . STRING	*class*
	|	S' : STRING	*pseudo-element*
op	::=	' ' | <	*child of*
	|	>	*descendant of*
	|	+	*sibling*
	|	[IDENT *opa* STRING]	*attribute*
opa	::=	=	*match exact*
	|	~=	*match member*
	|	|=	*match start*

Figure 2. Grammar for HSS selectors.

D	::=	IDENT : E	*declaration*
E	::=	`$expr`	*Hop expression*
	|	IDENT(E_1, ... ,E_p)	*function call*
	|	STRING	*literal*

Figure 3. Grammar for HSS declarations.

In this section we present successively the HSS extensions, each accompanied with a realistic motivating example.

3.1 HSS user-defined properties

A regular CSS declaration assigns a value to a property. Its syntax consists of a property name followed by colon followed by a value. Examples:

```
border: 1px solid red;
font-size: 120%;
background: rgb(100,50,200);
```

The set of property names that can be used in an assignment is closed, *i.e.,* users cannot define their own properties.

3.1.1 HSS properties

HSS extends CSS by allowing developers to declare their own new CSS properties. HSS programmers declare *property compilers* that rewrite a declaration into a list of new HSS declarations. These new declarations are themselves compiled. This process iterates until the declarations produced by compilation only concern primitive CSS-2 properties.

HSS views a property as consisting of a list of strings (the parameters) plus a distinguished string (the priority); the user may define a function, a *property compiler*, that takes these and returns a string that replaces the property call. In the example:

```
border: 1px solid red ! important;
```

The property compiler is the user-defined function associated with `border`, the *value* parameter is bound to the list `"1px" "solid" "red"` and the *priority* is the string `"important"`. The result of a property compiler is a string that is inserted in the generated file, in replacement of the property call.

H_{prop} ::= `(define-hss-property (`IDENT IDENT IDENT`) expr)`

Figure 4. Grammar for HSS property compilers.

Property compilers are defined by the `define-hss-property` form whose syntax is given Figure 4. Their signature is as follows:

property compiler: `string list` $\times$ `string` $\rightarrow$ `string`

The body of the property compiler is given by the expression `expr` that belongs to the programming language used with HSS. The current actual HSS implementation uses Hop so the rest of the paper.

The following example defines a new `black-and-white` property that controls the background and foreground colors of HTML elements. It accepts two possible values `regular` and `invert`. It might be defined as:

```
(define-hss-property (black-and-white val prio)
   (if (string=? (car val) "regular")
       "color: black; background: white;"
       "color: white; background: black;"))
```

and might be used in rules such as:

```
div.box {
  black-and-white: invert;
}
```

Note: it might be noted that a property compiler returns a new HSS fragment that is inserted in the style sheet. Because it is deemed simpler to generate actual HSS external syntax than any internal format, it has been decided that property compilers return strings instead of abstract syntax trees. As presented in Section 4 these strings are parsed back by the HSS compiler in order to complete the compilation process.

3.1.2 Motivating example: browser compatibility

The actual CSS properties set is specific to each browser. All mainstream browsers support a superset of CSS-2.1 properties. Vendor-specific properties have name prefixed with a distinguishing mark, *e.g.,* `-moz-`, `-webkit-`, or `-o-`. Implementing CSS files compatible with all the major browsers is then tedious because cluttered with many vendor-specific redundant declarations. For instance, specifying round borders for boxes could yield to:

```
pre.example {
  -moz-border-radius: 1em 2em 1em 2em;
  -webkit-border-top-left-radius: 1em;
  -webkit-border-top-right-radius: 2em;
  -webkit-border-bottom-right-radius: 1em;
  -webkit-border-bottom-left-radius: 2em;
  border-top-left-radius: 1em;
  border-top-right-radius: 2em;
  border-bottom-right-radius: 1em;
  border-bottom-left-radius: 2em;
}
```

HSS properties can be used to ensure browser compatibility automatically. For instance, the compatibility rule for round borders could be defined as:

```
(define-hss-property (border-radius v p)
   (match-case v
     ((?tl ?tr ?br ?bl)
      (format "-moz-border-radius: ~a ~a ~a ~a;
          -webkit-border-top-left-radius: ~a;
          -webkit-border-top-right-radius: ~a;
          -webkit-border-bottom-right-radius: ~a;
          -webkit-border-bottom-left-radius: ~a;
          border-top-left-radius: ~a;
          border-top-right-radius: ~a;
          border-bottom-right-radius: ~a;
          border-bottom-left-radius: ~a;"
          tl tr br bl tl tr br bl tl tr br bl))
     ((?r)
      (format "-moz-border-radius: ~a;
          -webkit-border-radius: ~a;
          border-radius: ~a;"
          r r r))
     (else
      (error 'border-radius "wrong number of args" v))))
```

The HSS property `border-radius` can be given either four values (one for each of the four corners) or only one value (that is used for all the four corners). Provided with this definition, the previous `pre.example` example can be re-written using the following compact declaration:

```
pre.example {
  border-radius: 1em 2em 1em 2em;
}
```

This example illustrates two advantages of HSS over CSS declarations:

1. HSS source files are more *compact* than their corresponding CSS files;
2. HSS files are *easier to maintain* because if a new browser shows up with its own syntax for supporting a popular extension, only the HSS rule declaring that extension has to be modified, not the user HSS sources. HSS helps applications developers to write portable CSS files.

3.2 HSS user-defined functions

A CSS value may be a literal (a string, a number, a color, etc.), a URI, or a function call. Examples:

```
color: red;
background: rgb(20%,40%,15%);
content: attr(bg);
```

As properties, functions cannot be defined by users and mainstream browsers extend the set defined by CSS-2. As properties, these extensions raise portability issues.

3.2.1 HSS functions

HSS supports user-defined functions. They are introduced by the `define-hss-function` form whose syntax is given Figure 5.

H_{fun} ::= `(define-hss-function (`IDENT IDENT ...`) expr)`

Figure 5. Function definition.

The signature of a HSS function is:

value function: string × ... → string

The formal parameters of HSS functions are bound to strings representing the actual values specified at the call sites. For instance, in the property assignment:

```
background: rgb(20%,40%,15%);
```

The three formal parameters of the `rgb` function are bound to "20%", "40%", and "15%".

3.2.2 Motivating example: specification compatibility

CSS-3 extends CSS-2 in many directions. In particular, the color module has been significantly augmented. CSS-3 colors may be specified using *RGB* components as well as *HSL* or *HSLA* components. For instance

```
div.highlight {
  background: rgb(255,0,0);
  opacity: 0.9;
}
```

may also be written using CSS-3 specification as:

```
div.highlight {
  background: hsla(0,100%,50%,90%);
}
```

The CSS-3 `hsla` function can be defined as a HSS value function and calls to `hsla` converted into CSS-2 attribute values during the HSS compilation. With HSS, `hsla` can be defined as:

```
(define-hss-function (hsla h s l a)
   (multiple-value-bind (r g b)
      (hsl->rgb (string->number h)
                (string->number s)
                (string->number l))
      (format "rgb(~a,~a,~a); opacity:~a;"
              r g b
              (percentage->real a))))
```

This definition uses the Hop library function `hsl->rgb` that returns the *RGB* components of a *HSL* color specification. This function returns three values that are bound to local variables (`r`, `g`, and `b`) by the `multiple-value-bind` form. The `a` component is compiled into an CSS2 `opacity` attribute. Provided with this definition of `hsla`, the example above using the *HSLA* components will be compiled into an equivalent CSS-2 compatible *RGB* form.

This example shows that HSS can be used to ensure backward compatibility of CSS files. Using HSS functions, a CSS-3 source file can be compiled into an equivalent CSS-2 file. This compilation can be automatically triggered by clients-side programs for which an auto-configuration test shows a lack of CSS-3 support.

3.3 HSS user-defined types

HTML types (*e.g.*, `DIV`, `TABLE`, ...) are central to the pattern matching rules that determine which style rules apply to each element of a document tree. HTML is not extensible, *i.e.*, users cannot define new types, as is CSS. HSS allows programmers to define new types that are used in the selector rules as any primitive type. This is one of the main HSS contributions.

3.3.1 HSS types

Types are defined by the `define-hss-type` form whose syntax is:

H_{type} ::= `(define-hss-type` IDENT S [`:body` S] $H_{prop}{}^{*}$ `)`

Figure 6. HSS types.

The rest of this section illustrates type definitions with various examples. The first one shows that `define-hss-type` can be used to assign names to selectors. For instance, provided with the following declaration,

```
(define-hss-type warning "span.warning")
```

the new type `warning` can be used as a substitute of `span.warning`. Hence, one may write:

```
/* global warning's borders */
warning {
  border: 4px dotted red;
}
/* "important" warning's color */
div.important warning {
  color: red;
}
/* buttons embedded inside warnings */
div.important warning button {
  background: yellow;
}
```

The first rule (global warning) specifies the general graphical aspect of a *warning* span (a span with a wide red dotted border). The second rule (import warning) overrides *warning* spans by specifying that *important warning* spans, *e.g.,* amongst the *warning* spans those that are included inside *important* divs, in addition to be red-bordered, are written in red. The third rule specifies that buttons that might be embedded inside *import warning* spans enjoy a yellow background.

HSS types may be accompanied with type-specific properties. For instance it can be found convenient to associate a *level* to the warning elements defined above. The higher the level, the more visible the graphical rendering. The *level* property of warning elements can be implemented as:

```
(define-hss-type warning "span.warning"
   (define-hss-property (level l)
      (cond
         ((string=? l "benign")
          "border: 2px solid red;")
         ((string=? l "important")
          "border: 4px dotted red;")
         ((string=? l "critical")
          "border: 4px dotted red;
           color: red;")
         (else
          "border: 4px dotted red;
           color: red;
           background: yellow"))))
```

rovided with these declarations, `warning` can be used in expressions such as:

```
warning#disk-failure {
  level: critical;
}
```

As seen in 2 the `#` pattern operator matches elements according to their identifier (here `disk-failure`). Then, the selector `warning#disk-failure` matches any element that belongs to the `warning` class and whose identifier is `disk-failure`. Such an element can be build with a Hop expression such as:

```
(<SPAN> :class "warning" :id "disk-failure"
   "Unable to read disk /dev/sda1")
```

3.3.2 Motivating example: abstraction

Many modern web frameworks abstract over HTML by proposing additional sets of widgets. JQuery or the Dojo toolkit are two popular representatives of this kind. These frameworks consist of JavaScript APIs that amongst other things implement widget constructors.

The Hop [9] development kit provides similar facilities with an important difference :

- Hop is a multi-tier language so its new widgets are supported on the *client-side* as well as on the *server-side*;
- pre-defined HTML markups *and* user-defined markups use the same syntax.

For instance, in Hop, one might create *labelled frames* with two following declarations:

```
(define-markup <LFRAME> (body)
   (<DIV> :hssclass "hop-lframe"
      (<DIV> :hssclass "hop-lfborder"
         (<DIV> :hssclass "hop-lfbody" body))))

(define-markup <LFLABEL> (body)
   (<DIV> :hssclass "hop-lflabel"
      (<SPAN> body)))
```

These two user-defined markups can be used as any HTML elements such as:

```
(<LFRAME> :class "lframe-left"
   (<LFLABEL> "A Label")
   (<DIV> :id "a-lfbody" "The lframe body."))
```

Hop compiles this expression into:

```
<DIV> class="lframe-left" hssclass="hop-lframe">
 <DIV hssclass="hop-lfborder">
  <DIV hssclass="hop-lfbody">
   <DIV hssclass="hop-lflabel">
    <SPAN>A Label</SPAN>
   </DIV>
   <DIV id='a-lfbody'>The lframe body.</DIV>
  </DIV>
 </DIV>
</DIV>
```

Being able to define new widgets is essential to web development kits. It allows developers to create *abstractions* on top of HTML in order to help creating rich user interfaces on the web. Web developers then no longer directly use HTML `DIV`, `SPAN`, or `TABLE` but `NOTEPAD`, `TREE`, or `DATEPICKER` instead.

Unfortunately, CSS does not support type abstraction. Thus, graphical tunings can only be applied to primitive HTML elements even when they are only used to implement high-level abstractions. For instance, the JQuery's API documentation acknowledges that a `DATEPICKER` is implemented as a HTML `DIV` of the class `ui-datepicker`. Unveiling these implementation details is opposed to elaborating safe and sound abstractions on top of a minimal core kernel.

HSS user-defined types can prevent CSS configurations to break abstractions of high-level APIs. This is illustrated with the HSS specification of `LFRAME` presented Figure 7. It shows the definition of the type element and its specific properties.

```
(define-hss-type lframe "div[hssclass=hop-lframe]"
   :body "div[hssclass=hop-lfbody]"
   (define-hss-property (-hop-label-margin v)
      (format "padding: ~l;" v))
   (define-hss-property (-hop-label-border v)
      (format "div[hssclass=hop-lfborder] {
                 border: ~l; }" v))
   (define-hss-property (padding v)
      (format "div[hssclass=hop-lfbody] {
                 padding: ~l; }" v))
   (define-hss-property (-hop-label-border-radius v)
      (format "div[hssclass=hop-lfborder] {
                 border-radius: ~l; }" v))
   (define-hss-property (-hop-label-align v)
      (format "div[hssclass=hop-lflabel] {
                 text-align: ~l; }" v))
   (define-hss-property (-hop-label-offset v)
      (format "div[hssclass=hop-lflabel] {
                 top: -~l; }" v))
   (define-hss-property (background v)
      (list (format "background: ~a;" (car v))
            (format "div[hssclass=hop-lflabel] > span {
                       background: ~a;
                    }"
                   (car v)))))

   (define-hss-type lflabel
      "div[hssclass=hop-lframe]
         div[hssclass=hop-lflabel]
           span")
```

Figure 7. Definition of the new lframe type element

HSS type-specific properties are nested inside a HSS type declaration. They are similar to global properties as presented Section 3.1 with two differences:

- they are only bound for the defined type element;
- instead of returning plain strings, the associated property compilers may return list of strings.

The signature of a type-specific property compiler is:

```
type property compiler:
  string list × string → string + string list
```

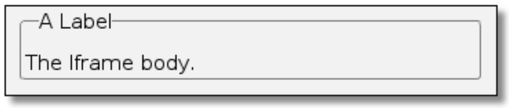
When the returned value is a list, any of the contained strings can either be a HSS property assignment or a plain HSS rule. The first case is used to configure the HTML element associated with the defined HTML element. The second case is used to configure HTML elements embedded inside the HTML element. In the example of Figure 7, `-hop-label-margin` is applied to HTML `DIV` whose `hssclass` is `hop-lframe`. The `-hop-label-border` is applied to HTML `DIV` whose `hssclass` is `hop-lfborder` and that are children of `LFRAME` elements. Section 4 shows how these properties are compiled into CSS.

The `:body` argument is bound to a selector string. It is used to change the HSS compiler aliasing resolution. The `:body` argument is added to a selector rule when the type element is not used in the right-most position or when it is used with a pseudo-selector such as `:first-child`.

The default configuration of `LFRAME` that is given by:

```
lframe {
  background: #edeceb;
  border: 1px solid black;
  padding: 2px;
  box-shadow: 5px 5px 5px #888;
  -hop-label-margin: 10px;
  -hop-label-border: 2px groove #ddd;
  -hop-label-border-radius: 4px;
  -hop-label-align: left;
  -hop-label-offset: 12px;
}
lflabel {
  font-style: roman;
}
```

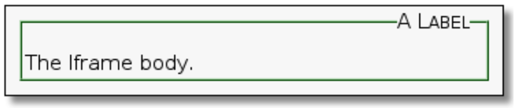
It produces the following rendering:



Changing the graphical aspects of a `LFRAME` no longer requires to be aware of its implementation. For instance

```
lframe.lframe-right {
  -hop-label-align: right;
  -hop-label-border-radius: 0;
  -hop-label-border: 2px ridge green;
}
lframe.lframe-right lflabel {
  font-variant: small-caps;
}
```

changes the default rendering. It flushes right the frame label written using a small caps font and used a green ridge border. It is rendered as:



3.4 Computed values

CSS is a purely declarative language. It cannot be used to express computations. CSS values are thus literals. The fourth extension supported by HSS allows computed values to be *injected* inside generated CSS files.

3.4.1 The $-form

HSS extends CSS statements and values with one additional syntax that *inject* compile-time values in the generated CSS file. The `$-form` permits HSS source files to declare local variables or import variables or to bind values to properties. The following example

```
$(module hss-example)

$(define bg "#999")
$(define (light-color v) ...not given here...)

button {
  background: $bg;
}
button:hover {
  background: $(light-color bg);
}
```

defines a Hop module `hss-example` that declares one global variable `bg` and one function `light-color`. The global variable is used twice in the example to specify the background color of buttons.

3.4.2 Motivating example: skins

Injecting dynamic values roots HSS inside the development kit that embeds it. As demonstrated by the examples, it allows the variables and the values of the programming language to be used inside HSS files. This may be used to customize HSS files. For instance, assuming a Hop data structure that implements skins:

```
(module skinning
  (export (class skin
            button-border
            button-background
            button-font
            toolbar-icon-size
            toolbar-style
            ...)
         (get-current-skin::skin)))
```

Then, HSS files may use these *skins* in source files such as:

```
$(module hss-example
   (import skinning))

$(define skin (get-current-skin))

button {
  border: $(skin-border skin);
  background: $(skin-button-bg skin);
  font: $(skin-font skin);
}
button:hover {
  background: $(light-color (skin-button-bg skin));
}
...
```

This technique for implementing skins allows applications conceived and implemented independently to share graphical configurations. Globally coherent visual aspect can thus be enforced at the environment level.

4. The HSS compiler

This section presents the whole HSS compiler. The compilations algorithms are given Figures 10 through 15. They use the syntactic notations presented Figure 9. Readers only interested by the HSS design and not concerned by the technical aspects of its implementation can safely skip this section.

The first step of the compilation *normalizes* HSS programs. It produces rules that contain only one selector and it evaluates global Hop expressions. After that stage programs are described by the grammar H^0 given in Figure 8.

H, the complete syntax for HSS and H^0 are related by:

$$H^0 \subset H$$

H^0 ::= S {D_1; ... D_m; } *rule*

Figure 8. Grammar for normalized HSS.

In addition, C, the actual CSS syntax is related to H by:

$$C \subset H$$

$\mathcal{C}$, the compilation function that transforms a HSS program into a CSS program, has the following prototype:

$$\mathcal{C} : H^0 \times \tau \times \rho \times \phi \rightarrow C$$

H^0 is the program to be compiled, τ is the element type environment that is extended by `define-hss-type`, ρ is the property environment that is extended by `define-hss-property`, and ϕ is the function environment that is extended by `define-hss-function`.

The compilation algorithm handles each rule separately. It resolves type element aliasing and it invokes property and value compilers.

In this presentation, the compilation of the rule

$$S_1\ op_1\ \ldots\ op_{n-1}\ S_n\ \{D_1;\ \ldots\ D_m;\}$$

in the type environment $_\tau$, the property environment $_\rho$, and the function environment $_\phi$ is written:

$$\mathcal{C}[\![S_1\ op_1\ \ldots\ op_{n-1}\ S_n\ \{D_1;\ \ldots\ D_m;\}]\!]_{\tau\rho\phi}$$

$S\downarrow_{type}$	the *type* of a selector S. *e.g.,* div#foo$\downarrow_{type}$ = `div`
$S\downarrow_\rho$	the property set of the type of the sector S. $S\downarrow_\rho \equiv \rho\ (S\downarrow_{type})$.
τ(IDENT)$\downarrow_{selector}$	the selector of user-defined types *e.g.,* τ(lframe)$\downarrow_{selector}$ = "div[hssclass=hop-lframe]"
τ(IDENT)$\downarrow_{body}$	the optional body of user-defined types *e.g.,* τ(lframe)$\downarrow_{body}$ = "div[hssclass=hop-lfbody]"
ρ(IDENT)$\downarrow_{selector}$	the selector of a type property.
ρ(IDENT)$\downarrow_{compiler}$	the compiler of user-defined property.
T$\downarrow_1$ and T$\downarrow_2$	the first and second projection. *e.g.,* <`a` $\times$ `b`>$\downarrow_2$ = `b`
§	the concatenation of lists.

Figure 9. Notations. This figure presents the notations used by the compilation algorithms.

S_n drives the compilation of rules such as:

$$\mathcal{C}[\![S_1\ op_1\ \ldots\ op_{n-1}\ S_n\ \{D_1;\ \ldots\ D_m;\}]\!]$$

- If S_n is a regular element then the compilation consists in resolving selector aliasing and expanding global HSS properties. This is implemented by the $\mathcal{C}_{simple}$ function.
- If S_n is a user-defined element then more important modifications are involved. In particular one HSS rule of this type may be compiled into several CSS rules. The function $\mathcal{C}_{user}$ is used to compile such rules. Figure 10 presents the part of the algorithm that dispatches according to rules kind.

```
C[[S1 op1 ... opn-1 Sn {D1; ... Dm;}]]τρφ =
  if Sn↓type ∉ τ or pseudo(Sn)
      then Csimple[[S1 op1 ... opn-1 Sn {D1; ... Dm;}]]τρφ
      then Cuser[[S1 op1 ... opn-1 Sn {D1; ... Dm;}]]τρφ
```

Figure 10. Dispatching rules

4.1 Compiling Simple Rules

The function $\mathcal{C}_{simple}$ compiles *simple* rules by resolving type aliasing and expanding attributes name and value. Section 3.3 offers a first example for $\mathcal{C}_{simple}$ that is detailed below. In the rule:

```
div.important warning button {
  background: yellow;
}
```

`warning` is a user defined type but not used in the right-most position of the selector. So it does not drive the rule properties expansion. That rule actually configures buttons, not warnings. Hence, its compilation merely consists in resolving selector aliasing. It then produces:

```
div.important span.warning button {
  background: yellow;
}
```

This example illustrates type aliasing, the Section 3.1.1 provides an example of attribute aliasing and value expansion. In that Section a `black-and-white` property has been defined. It can be used in rules such as:

```
div.box {
  black-and-white: invert;
}
```

Expanding the attribute `black-and-white` produces:

```
div.box {
  color: white;
  background: black;
}
```

The function $\mathcal{C}_{simple}$, in charge of this transformation, is presented in Figure 11.

```
Csimple[[S1 op1 ... opn-1 Sn {D1; ... Dm;}]]τρφ =
  Calias[[S1]]τ op1 ... opn-1 Calias[[Sn]]τ {
     Cdecl[[D1]]τρφ↓2; ... Cdecl[[Dm]]τρφ↓2;
  }
```

Figure 11. Compiling simple rule

4.1.1 Compiling Selectors

The compilation of simple selectors is given by the function $\mathcal{C}_{alias}$ given in Figure 12. As shown in the `warning` example, it replaces user-defined types, *i.e.,* types bound in the τ environment, with the selector expression given at the type declaration.

```
Calias[[IDENT]]τ =
  if IDENT ∈ τ
     then Calias[[τ(IDENT)↓selector]]τ
     else IDENT
Calias[[IDENT.STRING]]τ =
  if IDENT ∈ τ
     then Calias[[τ(IDENT)↓selector.STRING]]τ
     else IDENT.STRING
Calias[[IDENT:STRING]]τ =
  if IDENT ∈ τ
     then let i = τ(IDENT) in
            if i↓body
               then Calias[[i↓selector < i↓body:STRING]]τ
               else Calias[[i↓selector:STRING]]τ
     else IDENT:STRING
```

Figure 12. Compilation of selectors

One subtlety is introduced by pseudo-elements that actually change the alias resolution. This is illustrated by the following example that uses the `lframe` element presented in Figure 7:

```
lframe:first-child {
  border: 1px solid red;
}
```

The intention of this rule is to set a border to the first *user* child of `lframe` elements by opposition to the child of the element that implements the outer box of the `lframe`. This intention is specified in HSS by adding a `:body` argument to the type declaration. The `:body` argument is bound to a selector string which is used to change the HSS compiler aliasing resolution. The `:body` argument is added to a selector rule when the type element is not used in the right-most position or when it is used with a pseudo-selector such as `:first-child`.

In this example, since the declaration of `lframe` specifies a `:body` argument, then `:first-child` must be resolved against that element, not against the element implementing the outer `lframe` box. Hence, remembering that `lframe`'s `:body` attribute is `div[hssclass=hop-lfbody]`, $\mathcal{C}_{\text{alias}}$ compiles the previous declaration into:

```
div[hssclass=hop-lframe]
   div[hssclass=hop-lfbody]:first-child {
  border: 1px solid red;
}
```

Without the `:body` optional argument it would have been compiled into:

```
div[hssclass=hop-lframe]:first-child {
  border: 1px solid red;
}
```

4.1.2 Compiling Declarations

Declarations are compiled by the function $\mathcal{C}_{\text{decl}}$ (see Figure 13). The environment ρ binds property identifiers. For each identifier it associates a selector and a compiler.

```
$\mathcal{C}_{\text{decl}}[\![\text{IDENT : E}]\!]_{\tau\rho\phi}$ =
  if IDENT $\in \rho$
     then let s = $\rho$(IDENT)$\downarrow_{\text{selector}}$
              c = $\rho$(IDENT)$\downarrow_{\text{compiler}}$ in
            <s $\times$ c(E)>
     else <false $\times$ IDENT : $\mathcal{C}_{\text{expr}}[\![\text{E}]\!]_{\phi}$>
```

Figure 13. Compiling declarations

The second function needed to compile HSS declarations is $\mathcal{C}_{\text{expr}}$ (see Figure 14). It takes in charge the compilation of expressions. It resolves `$`-forms and user-defined functions bound in the environment ϕ.

```
$\mathcal{C}_{\text{expr}}[\![\text{STRING}]\!]_{\phi}$ = STRING
$\mathcal{C}_{\text{expr}}[\![\text{\$expr}]\!]_{\phi}$ = HOP(expr)
$\mathcal{C}_{\text{expr}}[\![\text{IDENT(expr)}]\!]_{\phi}$ =
   let e = $\mathcal{C}_{\text{expr}}[\![\text{expr}]\!]_{\phi}$ in
      if IDENT $\in \phi$
         then let f = $\phi$(IDENT) in f(e)
         else IDENT(e)
```

Figure 14. Compiling property values

4.2 Compiling User Rules

The right most element type of a selector specifies the elements the rule applied to. The left elements limit the set of elements the rule applies to so they do not impact the compilation of rule attributes. In the rest of this section, rules where the right most element is a user declared type are referred to as *user rules*. This section presents the HSS compilation function $\mathcal{C}_{\text{user}}$ that handles them.

As shown in Section 3.3 and in particular in the `lframe` example presented in Figure 7, properties of user defined types may be associated with selectors. For instance, the `padding` property of the user defined `lframe` type is associated with the selector `div[hssclass=hop-lfbody]`. HSS has to generate new selectors that accommodates the selector used in the user type declaration and the selector used in the property declaration. This is illustrated with two examples. The first one illustrates the compilation framework when the property does not come with its selector. The second illustrates the compilation when a selector is associated to a property.

- The `-hop-label-margin` defines a property of the main HTML element that implements the `lframe`, *i.e.,* a `div` element whose `hssclass` is `hop-lframe`. Hence, when the property `-hop-label-margin` is used in a declaration, it is compiled into a declaration applied to a `div[hssclass=hop-lframe]` element. So the declaration:

  ```
  lframe.foo {
    -hop-label-margin: 10px;
  }
  ```

  is compiled into:

  ```
  div[hssclass=hop-lframe].foo {
    padding: 10px;
  }
  ```

- The `-hop-label-border` property is associated with the selector `div[hssclass=hop-lfborder]`. So, when used in a rule, the selector on which this configuration applies is not the `lframe` element itself but the `div[hssclass=hop-lfborder]` that is nested in the `lframe`. Then, the declaration:

  ```
  lframe.foo {
    -hop-label-border: 2px groove #ddd;
  }
  ```

  is compiled into:

  ```
  div[hssclass=hop-lframe].foo
     div[hssclass=hop-lfborder] {
    border: 2px groove #ddd;
  }
  ```

To accomplish this transformation, the function $\mathcal{C}_{\text{user}}$ first splits the initial rule into *m* rules where *m* is the number of properties held by the rule. Second, it computes the selector of these new rules according to property declarations. The whole definition of $\mathcal{C}$ is given in Figure 15 where, for simplifying the presentation, it is assumed that type property compilers only return single strings.

```
$\mathcal{C}_{\text{user}}[\![S_1\ op_1\ \ldots\ op_{\text{n-1}}\ S_{\text{n}}\ \{D_1;\ \ldots\ D_{\text{m}};\}]\!]_{\tau\rho\phi}$ =
  let $\rho$' = $S_{\text{n}}\downarrow_{\rho}$ § $\rho$ in
      d1 = $\mathcal{C}_{\text{decl}}[\![D_1]\!]$ in
    if d1$\downarrow_1$
       then $\mathcal{C}[\![S_1\ op_1\ \ldots\ op_{\text{n-1}}\ S_{\text{n}}$ < d1$\downarrow_1$ {d1$\downarrow_2$;}$]\!]_{\tau\rho\phi}$
       else $\mathcal{C}_{\text{alias}}[\![S_1]\!]_{\tau}\ op_1\ \ldots\ op_{\text{n-1}}\ \mathcal{C}_{\text{alias}}[\![S_{\text{n}}]\!]_{\tau}$ {d1$\downarrow_2$;}
    ...
    let dm = $\mathcal{C}_{\text{decl}}[\![D_{\text{m}}]\!]$ in
       ...
```

Figure 15. User rules compilation

5. Limits and Future work

HSS supports a higher level of abstraction than CSS because it can be used to hide widgets implementation details. However, not all HTML style specifications are given in CSS files. Some are

Name	Syntax	Host	UType	UProp	Nesting	Var.	Fun.	Block	Expr.	Misc
Sass [3]	private	Ruby on Rail Merb	no	no	yes	yes	ruby	yes	SassScript	Property name-spaces
H(aXe)SS [2]	CSS	stand alone	no	no	yes	yes	no	yes	no	
Less [7]	CSS	command line Ruby	no	no	yes	yes	no	yes	ad hoc	Property name-spaces
CleverCSS [6]	python	python	no	no	yes	yes	ruby	-	python	
HSS [8]	CSS	stand-alone hop	yes	yes	no	yes	hop	yes	hop	skinning

Figure 16. This table summarizes the main characteristics of CSS compilers. Syntax denotes the input syntax of the compilers. The "Host" column gives indication on how the tool can be used. The column "Utype" shows which system allows users to define HTML elements. The column "Uprop" shows which system implements user properties. "Nesting" denotes the support of nested selectors declaration. "Var." denotes the possibility to declare variables inside style files. "Fun." denotes the possibility to develop private functions and, for those systems that support this feature, the programming language used for defining functions. "Block" denotes the possibility for a variable to contain a CSS value or a complete CSS block. "Expr." denotes the possibility to embedded expressions in property declarations and the language used for these expressions.

specified statically via the HTML `:style` attribute and others are even computed dynamically on the client-side code. In all these cases, HSS is hopeless.

It could be appealing to resort on the Hop dual compiler that allows a same source code to be compiled for the server (via native code compilation) as well as compiled for the client (via JavaScript code compilation [5]) to run the HSS compiler on the server-side *and* on the client-side of web applications. However, we think that this would be unrealistic because the HSS implementation is too big. It is undecidable if HSS is needed or not on the client side, then HSS would have to be included in the regular client-side runtime system, that is shipped with *all* the web pages delivered to clients. This would dramatically enlarge the load time of all Hop applications as it would clutter the memory of the web browsers than execute them.

User-defined HTML elements can be assigned names using the `define-hss-type` form. Developers refer to these names for specifying graphical tunings, independently of the implementation details. Unfortunately this name binding fails at totally hiding implementation details because it does not prevent the referenced HTML sequence to be impacted by general CSS rules. For instance, a `lframe` as defined in Figure 7 is actually an alias for `div[hssclass=hop-lframe]`. Then, `lframe` is impacted by any general rule mentioning `DIV` element such as:

```
div {
  background-color: red;
}
```

Although users may refer to `lframe` by a dedicated type name, a `lframe` is still a HTML `DIV` that can be configured directly by a CSS file! This let users infer that a `lframe` is actually implemented with a `DIV`. We see no solution to this problem.

The current version of HSS does not support user-defined pseudo-elements. However, this could help improving the backward compatibility between CSS-2 and CSS-3. CSS-3 defines additional pseudo-elements such as `:last-child` or `:nth`. Some of these are essential and cannot be simulated. Some could. This will be subject of a future HSS extensions.

6. Related work

HSS has first been mentioned but not detailed in an early publication [8]. We have not been able to observe other evidence of academic studies concerning CSS in the mainstream conferences and journals dedicated to the web. For instance, we have found no mention of CSS studies in any of the 10 last ACM WWW conferences. However, we think that CSS playing an important role in the production of web applications, it deserves our attention.

No article has been written by the academies or by research institutes but some tools have been produced by private companies or by the free software actors. In this section they are compared to HSS.

- **Sass**, [3] or *Syntactically Awesome StyleSheets*, is "a meta-language on top of CSS that's used to describe the style of a document cleanly and structurally, with more power than flat CSS allows". Sass can be used as a command-line tool or as a plugin for Ruby on Rails or Merb. Sass uses its own syntax based on tabulations and newlines. It supports variables, functions, blocks, and some fancy features such as property name-spaces. As several other tools, Sass supports block nesting which allows specifications of children to be declared inside their parent. This feature could be easily added to HSS but it does not seem essential since its only purpose is to abbreviate selectors.
- **H(aXe)SS**: [2] is a stand-alone pre-processor for CSS. It supports variables definitions and nested blocks.
- **Less** [7] is an extension of CSS that supports nested blocks, arithmetic operations on numbers and colors, lexical scoped variables but it supports no scripting nor functions.
- **CleverCSS** [6] is a CSS generator that relies on a Python based syntax. It supports embedded Python expressions, variables, and nested blocks.

All these systems, including HSS, share many features that help writing more compact and portable CSS files. Figure 16 summarizes their main characteristics. As HSS some of these other systems raise the abstraction level used in CSS files by supporting functions and variables. However, we think that HSS goes one step further by presenting a coherent extension framework that let developers declare functions, and variables as well as new element properties and new element types.

7. Conclusion

This paper has presented HSS that extends CSS with user defined variables, functions, and element types. The paper has shown that generating CSS files improves portability and maintainability and raises the abstraction level of CSS.

The paper has presented the HSS compilation algorithm which is simple enough to be re-implemented with any web-dedicated programming language.

The current HSS source code is shipped with Hop whose development kit is available at `http://hop.inria.fr`.

8. Acknowledgments

Thanks for the anonymous PPDP'10 referees whose help greatly improved this paper.

References

[1] G. Boudol, Z. Luo, T. Rezk, and M. Serrano. Towards Reasoning for Web Applications: an Operational Semantics for Hop. In *Proceedings of the first Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, Toronto, Canada, 2010.

[2] N. Cannasse. Haxe hss, Oct. 2008. URL `http://ncannasse.fr/projects/hss`.

[3] H. Catlin and N. Weizenbaum. Sass – syntactically awesome stylesheets, 2006. URL `http://sass-lang.com/`.

[4] W. W. W. Consortium. Cascading Style Sheets level 2 revision 1 CSS2.1 Specification. Technical Report CR-CSS2-20090423, W3C Recommendation, Apr. 2009.

[5] F. Loitsch and M. Serrano. *Trends in Functional Programming*, volume 8, chapter Hop Client-Side Compilation, pages 141–158. Seton Hall University, Intellect Bristol, ed. Morazán, M. T., UK/Chicago, USA, 2008. ISBN 978-1-84150-196-3.

[6] A. Ronacher. Clevercss, 2007. URL `http://sandbox.pocoo.org/clevercss/`.

[7] A. Selleir and D. Fadeyev. Less, 2009. URL `http://lesscss.org/index.html`.

[8] M. Serrano. The HOP Development Kit. In *Invited paper of the Seventh* ACM SIGPLAN *Workshop on Scheme and Functional Programming*, Portland, Oregon, USA, Sept. 2006.

[9] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, Portland, Oregon, USA, Oct. 2006.