Safely Composable Type-Specific Languages (Technical Report)

Cyrus Omar **Darya Kurilova** **Ligia Nistor**
Benjamin Chung **Alex Potanin**[†]
Jonathan Aldrich

July 27, 2014
CMU-ISR-14-106

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Victoria University of Wellington

Abstract

We present additional details on our static semantics and our corpus analysis that were omitted from the main body of the paper *Safely Composable Type-Specific Languages* for concision [1].

We acknowledge the support of the United States Air Force Research Laboratory and the National Security Agency lablet contract #H98230-14-C-0140, as well as the Royal Society of New Zealand Marsden Fund. Cyrus Omar was supported by an NSF Graduate Research Fellowship.

Keywords: extensible languages; parsing; bidirectional typechecking; hygiene

A Static Semantics

e elaborates to i and synthesizes type τ
e elaborates to i if analyzed against type τ

A.1 Mutually Recursive Type Declarations

$\boxed{\vdash_\Theta \theta \sim \Theta}$

$$\frac{\vdash_{\Theta_0} \theta \sim_{\text{names}} \Theta_{\text{names}} \quad \vdash_{\Theta_0\Theta_{\text{names}}} \theta \sim_{\text{defs}} \Theta_{\text{defs}} \quad \vdash_{\Theta_0\Theta_{\text{defs}}} \theta \sim_{\text{metadata}} \Theta}{\vdash_{\Theta_0} \theta \sim \Theta} \textit{ rec-decls}$$

$\boxed{\vdash_\Theta \theta \sim_{\text{names}} \Theta}$

$$\frac{}{\vdash_\Theta \emptyset \sim_{\text{names}} \emptyset} \textit{ empty-names}$$

$$\frac{\vdash_\Theta \theta' \sim_{\text{names}} \Theta' \quad T \notin \text{dom}(\Theta) \quad T \notin \text{dom}(\Theta')}{\vdash_\Theta \textbf{objtype}[T, \omega, e_m]; \theta' \sim_{\text{names}} T[?, ?]; \Theta'} \textit{ OT-names}$$

$$\frac{\vdash_\Theta \theta' \sim_{\text{names}} \Theta' \quad T \notin \text{dom}(\Theta) \quad T \notin \text{dom}(\Theta')}{\vdash_\Theta \textbf{casetype}[T, \chi, e_m]; \theta' \sim_{\text{names}} T[?, ?]; \Theta'} \textit{ CT-names}$$

$\boxed{\vdash_\Theta \theta \sim_{\text{defs}} \Theta}$

$$\frac{}{\vdash_\Theta \emptyset \sim_{\text{defs}} \emptyset} \textit{ empty-def}$$

$$\frac{\vdash_\Theta \omega \quad \vdash_\Theta \theta' \sim_{\text{defs}} \Theta'}{\vdash_\Theta \textbf{objtype}[T, \omega, e_m]; \theta' \sim_{\text{defs}} T[\textbf{ot}[\omega], ?]; \Theta'} \textit{ OT-def}$$

$$\frac{\vdash_\Theta \chi \quad \vdash_\Theta \theta' \sim_{\text{defs}} \Theta'}{\vdash_\Theta \textbf{casetype}[T, \chi, e_m]; \theta' \sim_{\text{defs}} T[\textbf{ct}[\chi], ?]; \Theta'} \textit{ CT-def}$$

$\boxed{\vdash_\Theta \theta \sim_{\text{metadata}} \Theta}$

$$\frac{}{\vdash_\Theta \emptyset \sim_{\text{metadata}} \emptyset} \textit{ empty-metadata}$$

$$\frac{\emptyset \vdash_{\Theta_0, T[\textbf{ot}[\omega], ?], \Theta} e_m \rightsquigarrow i_m \Rightarrow \tau_m \quad \vdash_{\Theta_0, T[\textbf{ot}[\omega], i_m : \tau_m], \Theta} \theta' \sim_{\text{metadata}} \Theta'}{\vdash_{\Theta_0, T[\textbf{ot}[\omega], ?], \Theta} \textbf{objtype}[T, \omega, e_m]; \theta' \sim_{\text{metadata}} T[\textbf{ot}[\omega], i_m : \tau_m]; \Theta'} \textit{ OT-metadata}$$

$$\frac{\emptyset \vdash_{\Theta_0, T[\textbf{ct}[\chi], ?], \Theta} e_m \rightsquigarrow i_m \Rightarrow \tau_m \quad \vdash_{\Theta_0, T[\textbf{ct}[\chi], i_m : \tau_m], \Theta} \theta' \sim_{\text{metadata}} \Theta'}{\vdash_{\Theta_0, T[\textbf{ct}[\chi], ?], \Theta} \textbf{casetype}[T, \chi, e_m]; \theta' \sim_{\text{metadata}} T[\textbf{ct}[\omega], i_m : \tau_m]; \Theta'} \textit{ CT-metadata}$$

Figure 1: Mutually Recursive Type Declaration Checking

The type declaration judgement in the paper only supports recursive types. Here, we include support for mutually recursive types by splitting the three key premises of the paper rule into three

distinct judgements, which each process the entire list of type declarations all the way through before going on to the next one. The three additional judgements in Fig. 1 operate as follows:

1. The judgement $\vdash_{\Theta_0} \theta \sim_{\text{names}} \Theta_{\text{names}}$ creates a named type context Θ_{names} containing only type names from θ, checking only that they are unique. Each binding in Θ_{names} is of the form $T[?, ?]$.

2. The judgement $\vdash_{\Theta_0\,\Theta_{\text{names}}} \theta \sim_{\text{defs}} \Theta_{\text{defs}}$ creates a named type context Θ_{defs} containing only type names and their definitions, checking only that any named types mentioned in the type definitions are available. Each binding in Θ_{names} is of the form $T[\delta, ?]$.

3. The judgement $\vdash_{\Theta_0\,\Theta_{\text{defs}}} \theta \sim_{\text{metadata}} \Theta$ finally checks that the metadata is well-typed. Metadata can explicitly refer to metadata of a type listed earlier in the list of type declarations, θ, but any other reference is a type error.

A.2 Context Formation

$\boxed{\vdash \Theta}$

$$\frac{}{\vdash \emptyset}\ \textit{Th-empty} \qquad \frac{\vdash \Theta \quad T \notin \text{dom}(\Theta) \quad \vdash_{\Theta, T[?,?]} \delta \quad \vdash_{\Theta, T[\delta,?]} \mu}{\vdash \Theta, T[\delta, \mu]}\ \textit{Th-extend}$$

$\boxed{\vdash_\Theta \delta}$

$$\frac{}{\vdash_\Theta ?}\ \textit{def-unknown} \qquad \frac{\vdash_\Theta \omega}{\vdash_\Theta \mathbf{ot}[\omega]}\ \textit{def-ot} \qquad \frac{\vdash_\Theta \chi}{\vdash_\Theta \mathbf{ct}[\chi]}\ \textit{def-ct}$$

$\boxed{\vdash_\Theta \mu}$

$$\frac{}{\vdash_\Theta ?}\ \textit{metadata-unknown} \qquad \frac{\emptyset \vdash_\Theta i \Leftarrow \tau}{\vdash_\Theta i : \tau}\ \textit{metadata}$$

$\boxed{\vdash_\Theta \Gamma}$

$$\frac{}{\vdash_\Theta \emptyset}\ \textit{G-empty} \qquad \frac{\vdash_\Theta \tau}{\vdash_\Theta \Gamma, x : \tau}\ \textit{G-extend}$$

Figure 2: Context Formation

In our metatheory, we need judgements expressing well-formed contexts, shown in Fig. 2. A lemma corresponding to Lemma 3 in the paper applies to our definition here as well:

Lemma 1 (Type Declaration (Mutually Recursive))**.** *If* $\vdash \Theta_0$ *and* $\vdash_{\Theta_0} \theta \sim \Theta$ *then* $\vdash \Theta_0\Theta$.

Proof. We prove the following more explicit lemma after inverting the type declaration derivation. □

Lemma 2 (Type Declaration (Explicit))**.** *If* $\vdash \Theta_0$ *and* $\vdash_{\Theta_0} \theta \sim_{names} \Theta_{names}$ *then* $\vdash \Theta_0\Theta_{names}$ *and if* $\vdash_{\Theta_0\Theta_{names}} \theta \sim_{defs} \Theta_{defs}$ *then* $\vdash \Theta_0\Theta_{defs}$ *and if* $\vdash_{\Theta_0\Theta_{defs}} \theta \sim_{metadata} \Theta$ *then* $\vdash \Theta_0\Theta$.

Proof. The proof is by induction on the structure of θ. We give the case $\theta = \textbf{objtype}[T, \omega, e_m]; \theta'$ (the case $\theta = \emptyset$ is trivial and the case $\theta = \textbf{casetype}[T, \chi, e_m]$ follows a directly corresponding argument). We have:

- By rule *OT-names* (which is the only rule that syntactically applies) we have that $\Theta_{\text{names}} = T[?, ?]; \Theta'_{\text{names}}$ and $T \notin \text{dom}(\Theta_0)$ and $T \notin \text{dom}(\Theta'_{\text{names}})$ and by the IH, $\vdash \Theta_0\Theta'_{\text{names}}$. Therefore, $\vdash \Theta_0\Theta'_{\text{names}}, T[?, ?]$ by rules *Th-extend*, *def-unknown* and *metadata-unknown*. Thus, $\vdash \Theta_0\Theta_{\text{names}}$ by suitable application of an exchange lemma, which we have assumed by metatheoretically defining Θ as a finite map over type names.

- By rule *OT-defs* (which is the only rule that syntactically applies) we have that $\Theta_{\text{defs}} = T[\textbf{ot}[\omega], ?]; \Theta'_{\text{defs}}$ and $\vdash_{\Theta_0\Theta_{\text{names}}} \omega$ and by the IH, $\vdash \Theta_0\Theta'_{\text{defs}}$. Therefore, $\vdash \Theta_0\Theta'_{\text{defs}}, T[\textbf{ot}[\omega], ?]$ by rules *Th-extend*, *def-ot* and *metadata-unknown*. By exchange, we have that $\vdash \Theta_0\Theta_{\text{defs}}$.

- By rule *OT-metadata* (which is the only rule that syntactically applies), we have that $\Theta = T[\textbf{ot}[\omega], i_m : \tau_m]; \Theta'$ and $\emptyset \vdash_{\Theta_0\Theta_{\text{defs}}} e_m \rightsquigarrow i_m \Leftarrow \tau_m$ and by the external type preservation lemma, $\emptyset \vdash_{\Theta_0\Theta_{\text{defs}}} i_m \Leftarrow \tau_m$. By rules *Th-extend*, *def-ot* and *metadata* we have that $\vdash \Theta_0, T[\textbf{ot}[\omega], i_m : \tau_m]$. By the IH, we then have that $\vdash \Theta_0\Theta$.

□

A.3 Metatheoretic Functions

We use several metatheoretic functions. Their properties are defined below.

Definition 1. *If* $\texttt{parsestream}(body) = i_{ps}$ *then* $\emptyset \vdash_{\Theta_0} i_{ps} \Leftarrow \textbf{\textit{named}}[ParseStream]$.

Definition 2. *If* $\vdash_{\Theta_0} i_{ps} \Leftarrow \textbf{\textit{named}}[ParseStream]$ *then there exists a* $body$ *such that* $\texttt{body}(i_{ps}) = body$ *and* $\texttt{parsestream}(body) = i_{ps}$.

Definition 3. *If* $\texttt{eparse}(body) = e$ *then* e *is the abstract syntax corresponding to the concrete syntax in* $body$*, as described in the paper.*

A.4 Notes on Reification and Dereification

Lemma 1 in the paper (Reification) requires additional clauses for completeness:

Lemma 3 (Reification (Full))**.** *If* $\Theta_0 \subset \Theta$ *then*

1. *If* $\vdash_\Theta \tau$ *then* $\tau \downarrow i$ *and* $\emptyset \vdash_\Theta i \Leftarrow \textbf{\textit{named}}[Type]$.
2. $x \downarrow i$ *and* $\emptyset \vdash_\Theta i \leftarrow \textbf{\textit{named}}[ID]$.[1]
3. $\ell \downarrow i$ *and* $\emptyset \vdash_\Theta i \leftarrow \textbf{\textit{named}}[ID]$

[1]Note that this judgement is, perhaps confusingly, about the metavariable x, not the internal term form for variables. Our syntax in the paper does not distinguish these directly, as is conventional, so rule *R-var* looks self-referential.

4. $C \downarrow i$ *and* $\emptyset \vdash_\Theta i \leftarrow$ ***named***$[ID]$

5. $T \downarrow i$ *and* $\emptyset \vdash_\Theta i \leftarrow$ ***named***$[ID]$

Proof. The proof for types is immediate by inspection. The remaining clauses are assumed definitionally, as we do not wish to prescribe particular grammars for variables, labels, constructor labels and type labels. □

For completeness, we can also state that every reified elaboration can be dereified:

Lemma 4 (Completeness of Dereification)**.** *If* $\Theta_0 \subset \Theta$ *and* $\emptyset \vdash_\Theta i \Leftarrow$ ***named***$[Exp]$ *and* i `val` *then* $i \uparrow \hat{e}$.

Proof. The proof is a simple induction that simply checks for coverage. □

A.5 Notes on Internal Type Safety and Type Preservation

Theorem 1, Theorem 2 and Lemma 2 in the paper require a slightly stronger inductive hypothesis. We can prove the following stronger theorems instead.

Theorem 1 (Internal Type Safety (Strong))**.** *If* $\vdash \Theta$ *then*

1. *If* $\emptyset \vdash_\Theta i \Leftarrow \tau$ *then either* i `val` *or* $i \mapsto i'$ *such that* $\emptyset \vdash_\Theta i' \Leftarrow \tau$.

2. *If* $\emptyset \vdash_\Theta i \Rightarrow \tau$*, then either* i `val` *or* $i \mapsto i'$ *such that* $\emptyset \vdash_\Theta i' \Rightarrow \tau$.

Theorem 2 (External Type Preservation (Strong))**.** *If* $\vdash \Theta$ *and* $\vdash_\Theta \Gamma$ *then*

1. *If* $\Gamma \vdash_\Theta e \rightsquigarrow i \Leftarrow \tau$ *then* $\Gamma \vdash_\Theta i \Leftarrow \tau$.

2. *If* $\Gamma \vdash_\Theta e \rightsquigarrow i \Rightarrow \tau$ *then* $\Gamma \vdash_\Theta i \Rightarrow \tau$.

Lemma 5 (Translational Type Preservation (Strong))**.** *If* $\vdash \Theta$ *and* $\vdash_\Theta \Gamma_{out}$ *and* $\vdash_\Theta \Gamma$ *and* $dom(\Gamma_{out}) \cap dom(\Gamma) = \emptyset$ *(which we can assume implicitly due to alpha renaming at binders) then*

1. *If* $\Gamma_{out}; \Gamma \vdash_\Theta \hat{e} \rightsquigarrow i \Leftarrow \tau$ *then* $\Gamma_{out}\Gamma \vdash_\Theta i \Leftarrow \tau$.

2. *If* $\Gamma_{out}; \Gamma \vdash_\Theta \hat{e} \rightsquigarrow i \Rightarrow \tau$ *then* $\Gamma_{out}\Gamma \vdash_\Theta i \Rightarrow \tau$.

B Corpus Analysis Methodology

Due to space limitations for our ECOOP'14 paper, some of the details of the methodology used to perform the corpus analysis were omitted. Here we restore the omission and provide a more detailed description of the used methodology.

For our analysis we used a recent version (20130901r) of the Qualitas Corpus [2], consisting of 107 Java projects. As the projects in the corpus contained various types of files that are necessary for a correct project setup in a programming environment but we were interested solely in the `.java` files, we ran the following commands in a terminal to remove all the non-Java files:

```
1 find . -type f -not -name "*.java" -exec rm {} \;
2 find . -type d -empty -delete
```

Having obtained a set of exclusively Java files, we ran the following commands to find all constructors in the code:

```
1 grep -r -n '[[:space:]]*public[[:space:]]*[A-Z][a-zA-Z0-9_-]*[[:space:]]*(' . >~/tmp/public-constructors.
      txt 2>/dev/null;
2 grep -r -n '[[:space:]]*protected[[:space:]]*[A-Z][a-zA-Z0-9_-]*[[:space:]]*(' . >~/tmp/protected-
      constructors.txt 2>/dev/null;
3 grep -r -n '[[:space:]]*private[[:space:]]*[A-Z][a-zA-Z0-9_-]*[[:space:]]*(' . >~/tmp/private-
      constructors.txt 2>/dev/null;
4 grep -r -n '^[[:space:]]*[A-Z][a-zA-Z0-9_-]*[[:space:]]*(' . >~/tmp/package-private-constructors.txt 2>/
      dev/null;
```

These commands use several observations pertaining to Java constructors:

1. Constructor names start with a capital letter.
2. Constructor names are usually followed by an opening parenthesis.
3. Constructor names may be prepended by scope-related keywords, such as `public`, `protected`, and `private`, or may not have any keywords in front of them (which means they are package private).
4. Constructor names or their scope-related keywords may have zero or more whitespace characters before them and no other characters.
5. There may be zero or more whitespace characters between the constructor name and an opening parenthesis and no other characters.

According to these observations, the first command is to find all public constructors, second all protected constructors, third all private constructors, and fourth all package-private constructors. After the four corresponding files were created, we did a quick visual scan through to verify that only constructors were found, merged the four files into one, and concluded that there were 124,873 Java constructors.

Having obtained a collection of constructors, we used the following *vi* text editor's command to find constructors that take in at least one `String` argument:

```
1 :g/String /
```

Running this command, we found that there were 30,161 constructors, i.e., 24% of the total, that had at least one `String` argument. Then, we searched for constructors that used `String`s that could be substituted with TSLs. We did a visual scan of the constructors' signatures and inferred the functionality of the constructor and the arguments it was taking in. To give an intuition for how the inference process went, below we give a positive example, i.e., an example of a constructor in which we concluded that a TSL could be used instead of the `String` argument, and a negative example, i.e., an example of a constructor in which we concluded that, although the constructor had a `String` argument, using a TSL instead might not have benefited the implementation.

Positive Example Consider the following example, which was positively classified in our code analysis:

Constructor: public IPAddressConversion(String IPaddr)
File: `jtopen-7.1/com/ibm/as400/util/commtrace/IPAddressConversion.java`
Line: 51

This constructor was found in the JROpen project, on line 51 in a file called `IPAddressConversion.java`. The constructor is called `IPAddressConversion` and the name of the `String` argument is `IPaddr`. From these pieces of information, we inferred that the `String` argument representing an IPv4 address, which is usually of the following form where `D` is a number between 0 and 255:

```
D.D.D.D
```

This format could be represented by a Wyvern TSL where we support variable splicing using the format `%x`:

```
objtype IPAddress
  ...
  metadata = new : HasTSL
    val parser = ~
      start <- D '.' D '.' D '.' D
        fn (e1, e2, e3, e4) => ~
          new
            val d1 = %e1%
            val d2 = %e2%
            val d3 = %e3%
            val d4 = %e4%
      D <- numlit
        fn (e1) => e1
      D <- '%' ID
        fn (e1) => e1
```

Hence, the constructor could use a TSL instead of the `String` argument and thus benefit from a guarantee that at the runtime the passed-in argument adheres to the necessary format.

Negative Example A negative example in our inference process, i.e., a constructor that has at least one `String` argument but may not benefit from substituting it with a TSL, is presented below:

Constructor: public InternalEntity(String name, String text, boolean inExternalSubset)
File: `xerces-2.10.0/src/org/apache/xerces/impl/XMLEntityManager.java`

1. **Constructor:** public CatalogEntry(String publicID, CatalogReader catalog)

   File: `netbeans-6.9.1/xml.catalog/src/org/netbeans/modules/xml/catalog/CatalogEntry.java`

   Line: 64

2. **Constructor:** public ConfigurableAwtMenu(String menuID, VariableBundle vars)

   File: `pooka-3.0-080505/net/suberic/util/gui/ConfigurableAwtMenu.java`

   Line: 35

3. **Constructor:** public ExternalRuleID(String id)

   File: `pmd-4.2.5/src/net/sourceforge/pmd/ExternalRuleID.java`

   Line: 11

Figure 3: Examples of constructors in the "Identifier" category (process ID, user ID, column or row IDs, etc.)

Line: 2,486

This constructor was found in the Xerces project, on line 2,486 of a file called `XMLEntityManager.java`. The name of the constructor is `InternalEntity`, and it has two `String` arguments: one called `name` and the other called `text`. The name of the constructor and the names of the passed-in `String` arguments are generic and thus we cannot infer the exact functionality of the constructor. In turn, we cannot suggest a TSL to be used to capture the functionality. Therefore, it is not obvious that the constructor would benefit from using a TSL instead of any of its `String` arguments, and we classify this example as negative.

To give an insight into all types of `String`s that we identified, we provide examples for each type: Figure 3 presents examples for the "Identifier" category; Figure 4 presents examples for the "Directory path" category; Figure 5 presents examples for the "Pattern" category; Figure 6 presents examples for the "URL/URI" category; Figure 7 presents examples for the "Other" category (containing several subtypes of `String`s).

1. **Constructor:** public VersionRelease(String homeDir)

   File: `jboss-5.1.0/build/VersionRelease.java`

   Line: 72

2. **Constructor:** public DataQueueDocument (AS400 system, String path)

   File: `jtopen-7.1/com/ibm/as400/vaccess/DataQueueDocument.java`

   Line: 140

3. **Constructor:** public ClassPathContextResource(String path, ClassLoader classLoader)

   File: `springframework-3.0.5/projects/org.springframework.core/src/main/java/org/springframework/core/io/DefaultResourceLoader.java`

   Line: 127

Figure 4: Examples of constructors in the "Directory path" category

1. **Constructor:** public RegexFilter(String regex)

   File: `drjava-stable-20100913-r5387/src/edu/rice/cs/drjava/config/RecursiveFileListProperty.java`

   Line: 61

2. **Constructor:** public NameEndsWith(String suffix)

   File: `struts-2.2.1/src/xwork-core/src/main/java/com/opensymphony/xwork2/util/ResolverUtil.java`

   Line: 141

3. **Constructor:** public NumberEditor(JSpinner jSpinner, String decimalFormat)

   File: `netbeans-6.9.1/html/src/org/netbeans/modules/html/palette/items/OLCustomizer.java`

   Line: 303

Figure 5: Examples of constructors in the "Pattern" category (regular expressions, prefixes and suffixes, delimiters, format templates, etc.)

1. **Constructor:** public XConnection(ExpressionContext exprContext, String driver, String dbURL, String user, String password)

   File: `xalan-2.7.1/src/org/apache/xalan/lib/sql/XConnection.java`

   Line: 239

2. **Constructor:** public DOMLocatorImpl(int lineNumber, int columnNumber, String uri)

   File: `xerces-2.10.0/src/org/apache/xerces/dom/DOMLocatorImpl.java`

   Line: 82

3. **Constructor:** public MockHttpServletRequest(ServletContext servletContext, String method, String requestURI)

   File: `springframework-3.0.5/projects/org.springframework.web/src/test/java/org/springframework/mock/web/MockHttpServletRequest.java`

   Line: 223

Figure 6: Examples of constructors in the "URL/URI" category

1. *ZIP code*

   Constructor: public Customer(Integer customerId, String zip)

   File: `netbeans-6.9.1/websvc.rest/test/unit/data/testsrc/com/acme/Customer.java`

   Line: 69

2. *Password*

   Constructor: public WrappedConnectionRequestInfo(final String user, final String password)

   File: `jboss-5.1.0/connector/src/main/org/jboss/resource/adapter/jdbc/WrappedConnectionRequestInfo.java`

   Line: 39

3. *Query*

   Constructor: public JDBCXYDataset(Connection con, String query)

   File: `jfreechart-1.0.13/source/org/jfree/data/jdbc/JDBCXYDataset.java`

   Line: 175

4. *HTML/XML*

   Constructor: public HtmlContentPopUp(java.awt.Frame parent, String title, boolean modal, String html)

   File: `jag-6.1/src/com/finalist/jaggenerator/HtmlContentPopUp.java`

   Line: 88

5. *IP address*

   Constructor: public HostRecord(String ip, String name, boolean ssh)

   File: `netbeans-6.9.1/cnd.remote/src/org/netbeans/modules/cnd/remote/ui/wizard/HostsListTableModel.java`

   Line: 173

6. *Version*

   Constructor: public EncryptHeader(short type, String version)

   File: `jgroups-2.10.0/src/org/jgroups/protocols/ENCRYPT.java`

   Line: 1,147

Figure 7: Examples of constructors in the "Other" category

References

[1] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014.

[2] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference*, 2010.