# Formal Islands

Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau

UHP & LORIA, INRIA & LORIA, and INRIA & LORIA,
BP 101, 54602 Villers-lès-Nancy Cedex France
{Emilie.Balland,Claude.Kirchner,Pierre-Etienne.Moreau}@loria.fr

**Abstract.** Motivated by the proliferation and usefulness of Domain Specific Languages as well as the demand in enriching well established languages by high level capabilities like pattern matching or strategic rewriting, we introduce the *Formal Islands* framework.

The main idea consists to integrate, in existing programs, formally defined parts called Islands, on which proofs and tests can be meaningfully developed. Then, *Formal Islands* could be safely dissolved into their hosting language to be transparently integrated in the existing user environment.

The paper presents this generic framework and shows that the properties valid on the formal islands are also valid on the corresponding dissolved host codes. Formal Islands can be used as a general methodology to develop new DSL and we show that language extensions like SQLJ—embedding SQL capabilities in Java—, or Tom—a Java language extension allowing for pattern matching and rewriting—are indeed Islands and they can therefore be used for formal software developments.

## 1 Introduction

At all the levels of our social and scientific organizations, the development of formal proofs of program properties is recognized as a priority of fundamental interest. But this faces at least two important difficulties. First is the lack of formal environments for existing widely used programming languages like Java, C or ML. Second is the scalability to allow for the proof of properties of large programs. Third is the fact that on the enormous corpus of active softwares, maintenance and adaptation should be conducted without having to rewrite or deeply transform the existing running codes. Therefore we are in need of having language extensions, formally defined, capable of adaptation to existing largely used programming languages and that do not induce dependence on a new language.

To contribute to solve these problems given the above constraints, we propose the concept of *Formal Islands* and show how it could be implemented and used. Indeed, taking the geography metaphor as well as a terminology already used for island grammars [8], we call *Ocean* the language of interest, typically C or Java, and *Island* the language extension that we would like to define.

As shown in Fig. 1, the Island cycle of life is composed of 4 phases:

- *anchor* which relates the grammars and the semantics of the two languages,
- *construction* which inserts some Island code in an Ocean program,
- *proofs* or program transformations on islands,

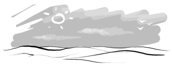
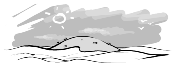
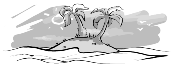
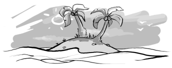
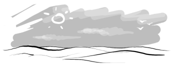
| In pictures | | | | |
|---|---|---|---|---|
|  |  |  |  |  |
| Existing code | Anchor | Construction | Proofs | Dissolution |
| For example | | | | |
| Java | ADT | Rewrite rules | Termination | Compilation |

**Fig. 1.** Formal Islands in picture

- *dissolution* of the islands in the Ocean language.

The anchoring step consists in defining the grammar and semantics of the Island language and in relating it to the existing Ocean one. This step should in particular take care of the data representation correspondence between the Island and Ocean constructions. For instance, we will define a new abstract data type on the Island which correspondence in the Ocean should be made explicit. For example, a list structure on the Island could be implemented as an array in the Ocean. This is typically reminiscent of observational specifications and is quite flexible.

The construction phase consists in writing a program in the combined Island and Ocean languages. For example, we could consider, as it is already possible in Tom[1], to define functions using matching constructs (of the form `%match pattern -> JavaCode`) or using term rewrite rules (of the form `%rule term -> term`). What is quite appealing at this level is the possibility to mix both language constructions to ease either the expressivity or the references to the existing Ocean structures or functionalities.

Then comes the proof phase. It is not necessary used, but defining formally such a framework enables developers of language extensions to formally check their well-formedness and properties. For example, defining in Tom a set of rewrite rules on top of Java, one could check at that step the termination of the rewrite system, therefore ensuring a better confidence in the program behavior.

Last, the Island should be dissolved. This means that the framework should provide a compilation of Island built programs (that may embed Ocean subparts) into pure Ocean ones. For example again, in Tom, a set of rewrite rules will be compiled into a Java program implementing the normalization process for these rules. Of course the framework setting should ensure that the properties proved at the Island level are still valid after dissolution for the concerned Ocean code.

To achieve these goals, after setting the basic notations in Section 2, we present in Section 3 the anchoring mechanism, in Section 4 the dissolution one and in Section 5 the *Island framework*, making precise the properties that the Island should fulfill to be *Formal* and to preserve proofs. From these definitions and result, we illustrate in Section 6 how domain specific languages [12] can be implemented in this formal islands framework. Finally, we present a main application in the context of the Tom project.

[1] http://tom.loria.fr

## 2 Preliminaries

When considering the problem of combining two different languages, we have to understand the relationship that exists between the grammars of these languages, the programs that can be written in these grammars, their semantics, and the objects that are manipulated by these programs.

We assume the reader to be familiar with the basic definitions of languages constructions and first-order term rewriting as given for example in [1]. We briefly recall or introduce notations for the main concepts that will be used along this paper.

A *grammar* is a tuple $\mathcal{G} = (A, N, T, R)$ where $A$ denotes the axiom, $N$ and $T$, disjoint finite sets of respectively, non-terminal and terminal symbols, and $R$ a finite set of production rules of the form $N \rightarrow (N \cup T)^*$. $\texttt{left}(R)$ is the set left-handsides of $R$. We note $\mathcal{L}(\mathcal{G})$ the *language* recognized by the grammar $G$. When a grammar $G$ is not ambiguous, to each valid program we can associate *abstract syntax tree* representations. Assuming given such a representation, we note $\mathsf{AST}(\mathcal{G})$ the set of all abstract syntax trees, and their subtrees.

In the following, we only consider unambiguous grammars. Therefore, we make no distinction between the notions of grammars and valid programs $p$, and the notions of signature and abstract syntax trees. We note $p_{\mathsf{ast}}$ the abstract syntax tree that represents $p$. Given a term $t = p_{\mathsf{ast}}$, $t \in \mathsf{AST}(\mathcal{G})$, we note $\texttt{getSort}(t)$ its sort, which corresponds to the non-terminal generating $p$.

In addition to the definition of grammars, we use a big-step reduction relation *à la* Kahn, written $\mapsto_{bs}$, to characterize the semantics of the Ocean and the Islands languages. Given a set $\mathcal{O}$ of objects manipulated by a program, corresponding to all possible instances of the data-model, an *environment* is a function from $\mathcal{X}$ to $\mathcal{O}$, where $\mathcal{X}$ is a set of variables. $\mathcal{E}nv$ denotes the set of all environments. The reduction relation $\mapsto_{bs}$ is defined using a set of inference rules of the form:

$$\langle \epsilon, i \rangle \mapsto_{bs} \epsilon' \text{ with } i \in \mathsf{AST}(\mathcal{G}), \text{ and } \epsilon, \epsilon' \in \mathcal{E}nv$$

In the following, we consider two languages $\mathsf{il}$ and $\mathsf{ol}$, the Island and the Ocean languages, to which corresponds respectively a grammar $\mathcal{G}_{\mathsf{il}}$ (resp. $\mathcal{G}_{\mathsf{ol}}$), a set of variables $\mathcal{X}_{\mathsf{il}}$ (resp. $\mathcal{X}_{\mathsf{ol}}$), a semantics $bs_{\mathsf{il}}$ (resp. $bs_{\mathsf{ol}}$) based on a set of objects $\mathcal{O}_{\mathsf{il}}$ (resp. $\mathcal{O}_{\mathsf{ol}}$), and a set of inference rules $R_{\mathsf{il}}$ (resp. $R_{\mathsf{ol}}$).

## 3 Anchor

Given two languages $\mathsf{il}$ and $\mathsf{ol}$, we introduce the notions of *syntactic anchor* and *representation mapping*, which make a connection between $\mathsf{il}$ and $\mathsf{ol}$ in a syntactically and semantically ways.

### 3.1 Syntax

The *syntactic anchor* consists in associating $\mathsf{ol}$ non-terminals to $\mathsf{il}$ non-terminals, to obtain $\mathsf{ol}$ programs with $\mathsf{il}$ parts. In Definition 1, we introduce two types of anchors corresponding to two types of islands. One called *simple island*, corresponding to pure il constructs and the other called *islands with lakes*, corresponding to islands which can recursively contain ol constructs.

**Definition 1.** *Given two grammars $\mathcal{G}_{ol} = (A_{ol}, N_{ol}, T_{ol}, R_{ol})$ and $\mathcal{G}_{il} = (A_{il}, N_{il}, T_{il}, R_{il})$, we define two kinds of syntactic anchors:*

- *A* simple syntactic anchor *is a relation $\mathtt{anch}(\mathcal{G}_{ol}, \mathcal{G}_{il}) \subset N_{ol} \times N_{il}$ where we assume that $(T_{ol} \cap T_{il}) = \emptyset \wedge (N_{ol} \cap N_{il}) = \emptyset$,*
- *A* syntactic anchor with lakes *is a relation $\mathtt{anch}(\mathcal{G}_{ol}, \mathcal{G}_{il}) \subset N_{ol} \times N_{il}$ where we assume that $(T_{ol} \cap T_{il}) = \emptyset \wedge (N_{ol} \cap \mathtt{left}(R_{il})) = \emptyset$.*

*From this definition, the grammar $\mathcal{G}_{oil}$, resulting from the combination of ol and il, is specified as follows:*

$$\mathcal{G}_{oil} = (A_{ol}, N_{ol} \cup N_{il}, T_{ol} \cup T_{il}, R_{ol} \cup R_{il} \cup \mathtt{anch}(\mathcal{G}_{ol}, \mathcal{G}_{il}))$$

Therefore, the syntax of the language oil, combination of ol and il is function of the grammars $\mathcal{G}_{\mathsf{ol}}$ and $\mathcal{G}_{\mathsf{il}}$, and of the syntactic anchor noted anch.

*Example 1.* As a first example, let us consider the two grammars, $\mathcal{G}_{\mathsf{ol}} = (\{A\}, \{A\}, \{a\}, \{(A ::= a), (A ::= AA)\})$ and $\mathcal{G}_{\mathsf{il}} = (\{B\}, \{B\}, \{b\}, \{(B ::= b)\})$. The language $\mathcal{L}(\mathcal{G}_{\mathsf{ol}})$ is the set of sequences a, aa, aaa, ... The language $\mathcal{L}(\mathcal{G}_{\mathsf{il}})$ contains only b. By considering the *simple syntactic anchor* $\mathtt{anch}(\mathcal{G}_{\mathsf{ol}}, \mathcal{G}_{\mathsf{il}}) = \{(A ::= B)\}$ we define the language $\mathcal{L}(\mathcal{G}_{\mathsf{oil}})$ which consists of words like a, b, aa, bb, ab, and more generally of any sequence of a or b.

For *simple syntactic anchors*, the condition $T_{\mathsf{ol}} \cap T_{\mathsf{il}} = \emptyset \wedge N_{\mathsf{ol}} \cap N_{\mathsf{il}} = \emptyset$ ensures that there is no conflict between the two grammars. But in some cases, it is interesting to allow the embedding of Ocean constructs inside Island code. We call *lakes* such constructs that are not modified by the dissolution phase. In term of syntactic anchor, this means that the il grammar can use non-terminals from $\mathcal{G}_{\mathsf{ol}}$. For this notion of *syntactic anchor with lakes*, the non conflict condition becomes $T_{\mathsf{ol}} \cap T_{\mathsf{il}} = \emptyset \wedge N_{\mathsf{ol}} \cap \mathtt{left}(R_{\mathsf{il}}) = \emptyset$.

*Example 2.* To illustrate the notion of *anchor with lakes*, we now consider an *Ocean language* ol which allows to manipulate arrays of integers. The considered Island language il allows to manipulate lists of integers, where the notion of integers comes from the Ocean language: this is why it is considered as a lake. The grammars of both languages are given in figure 2.

In ol, an array can be allocated and filled with 0 using the construction $array(n)$. Given an array $t$ and an integer $n$, $t[n]$ allows to read the contents of $t$. Similarly, $t[n] = i$, with $i \in \mathbb{N}$, allows to modify the contents of $t$.

In il, data structures are lists, which are classically defined by two constructors *nil* and *cons*. This language defines Islands with lakes since the non-terminal $\langle int \rangle$ comes from $\mathcal{G}_{\mathsf{ol}}$. To interconnect the two languages, we define the anchor $\mathtt{anch} = \{(\langle instr \rangle ::= \langle instruction \rangle), (\langle array \rangle ::= \langle list \rangle), (\langle int \rangle ::= \langle expr \rangle)\}$.

Using the grammar defined in Fig. 2, the following program is valid in the Ocean language: `t=array(5); t[0]=3; t[1]=7`. This program can be extended by

`The Ocean language`

$$\begin{array}{lcl}
\langle \mathit{instr} \rangle & ::= & \langle \mathit{instr} \rangle ; \langle \mathit{instr} \rangle \\
 & \mid & \langle \mathit{vararray} \rangle = \langle \mathit{array} \rangle \\
 & \mid & \langle \mathit{varint} \rangle = \langle \mathit{int} \rangle \\
 & \mid & \langle \mathit{array} \rangle [\langle \mathit{int} \rangle] = \langle \mathit{int} \rangle \\
\langle \mathit{array} \rangle & ::= & \mathit{array}(\langle \mathit{int} \rangle) \\
 & \mid & \langle \mathit{vararray} \rangle \\
\langle \mathit{vararray} \rangle & ::= & x \in \mathcal{X} \\
\langle \mathit{varint} \rangle & ::= & x \in \mathcal{X} \\
\langle \mathit{int} \rangle & ::= & i \in \mathbb{N} \\
 & \mid & \langle \mathit{varint} \rangle \\
 & \mid & \mathit{size}(\langle \mathit{vararray} \rangle) \\
 & \mid & \langle \mathit{array} \rangle [\langle \mathit{int} \rangle]
\end{array}$$

`The Island language`

$$\begin{array}{lcl}
\langle \mathit{instruction} \rangle & ::= & \langle \mathit{varlist} \rangle \leftarrow \langle \mathit{list} \rangle \\
\langle \mathit{list} \rangle & ::= & \mathit{nil} \\
 & \mid & \mathit{cons}(\langle \mathit{expr} \rangle, \langle \mathit{list} \rangle) \\
 & \mid & \mathit{tail}(\langle \mathit{list} \rangle) \\
 & \mid & \langle \mathit{varlist} \rangle \\
\langle \mathit{varlist} \rangle & ::= & x \in \mathcal{X} \\
\langle \mathit{expr} \rangle & ::= & \langle \mathit{int} \rangle \\
 & \mid & \mathit{head}(\langle \mathit{list} \rangle)
\end{array}$$

`The syntactic anchor relation`

$$\begin{array}{lcl}
\langle \mathit{instr} \rangle & ::= & \langle \mathit{instruction} \rangle \\
\langle \mathit{array} \rangle & ::= & \langle \mathit{list} \rangle \\
\langle \mathit{int} \rangle & ::= & \langle \mathit{expr} \rangle
\end{array}$$

**Fig. 2.** Syntax of the combination of the tool languages

`l←cons(t[1],cons(t[2],nil)); x=l[1]; y=head(l)`. This shows that a list of the Island language can be considered as an array by the Ocean language (`l[1]`). The integer `t[1]` and `t[2]` are lakes in the Island `l←cons(t[1],cons(t[2],nil))`.

### 3.2 Semantics

As for the syntax, we assume given a semantics definition for each language. In the most general case, the objects manipulated by these two languages are not of the same nature. For example, the Ocean language can manipulate tuples and the Island language, algebraic terms. Before giving a semantics to the extended language, we have to make precise the data-structure representations of Island objects in Ocean (the *representation mapping*) and how the data-structure properties in $\mathsf{il}$ are mapped to data-structure properties in $\mathsf{ol}$ (the *predicate mapping*).

**Definition 2.** *Given a set of Island objects $\mathcal{O}_{\mathsf{il}}$ and a set of Ocean objects $\mathcal{O}_{\mathsf{ol}}$, a* representation mapping $\lceil\ \rceil$ *is an injective mapping from $\mathcal{O}_{\mathsf{il}}$ to $\mathcal{O}_{\mathsf{ol}}$.*

*Example 3 (from example 2).* Every list from the Island language can be represented by an array in the Ocean language which contains exactly the same integers in the same order. The second kind of objects manipulated by the Ocean language is the integers whose representation is the same in the two languages. We note this representation mapping $map_1$

In Definition 2, the notion of representation mapping has been introduced to establish a correspondence between data structures in the Island and their representation in the Ocean language. However, we did not put any constraint on the representation of objects.

In particular, the function $\lceil\ \rceil$ does not necessarily preserve structural properties of Island objects. In practice, we need to consider mappings such that properties are preserved. Therefore, for each language we consider a *set of predicates* noted $\mathcal{P}_{\mathsf{ol}}$ and $\mathcal{P}_{\mathsf{il}}$ corresponding to structural properties, and we introduce the notion of *predicate mapping*.

**Definition 3.** *Given a set of Island predicates $\mathcal{P}_{\mathsf{il}}$ and a set of Ocean predicates $\mathcal{P}_{\mathsf{ol}}$, a* predicate mapping *$\phi$ is an injective mapping from $\mathcal{P}_{\mathsf{il}}$ to $\mathcal{P}_{\mathsf{ol}}$ such that $\forall p \in \mathcal{P}_{\mathsf{il}}, arity(p) = arity(\phi(p))$. This mapping is extended by morphism on first-order formulae, using the representation mapping:*

$$\begin{array}{l|l}
\multicolumn{2}{l}{\forall p \in \mathcal{P}_{\mathsf{il}}, \forall t_1, \ldots, t_n, \phi(p(t_1, \ldots, t_n)) = \phi(p)(t'_1, \ldots, t'_n)} \\
\multicolumn{2}{l}{\quad \textit{where } t'_i = \lceil t_i \rceil \textit{ if } t_i \in \mathcal{O}_{\mathsf{il}} \textit{ and } t_i \textit{ otherwise (i.e. } t_i \textit{ is a variable),}} \\
\phi(\forall x\ P) = \forall x\ \phi(P), & \phi(\exists x\ P) = \exists x\ \phi(P), \\
\phi(P_1 \vee P_2) = \phi(P_1) \vee \phi(P_2), & \phi(P_1 \wedge P_2) = \phi(P_1) \wedge \phi(P_2), \\
\phi(\neg P) = \neg \phi(P), & \phi(P_1 \rightarrow P_2) = \phi(P_1) \rightarrow \phi(P_2).
\end{array}$$

**Definition 4.** *Given a predicate mapping $\phi$, a representation mapping $\lceil\ \rceil$ is said $\phi$-*formal *if $\forall p \in \mathcal{P}_{\mathsf{il}}, \forall o_1, \ldots, o_n \in \mathcal{O}_{\mathsf{il}}$ with $n = arity(p)$*

$$p(o_1, \ldots, o_n) \Leftrightarrow \phi(p)(\lceil o_1 \rceil, \ldots, \lceil o_n \rceil)$$

*Example 4 (from example 2).* Consider the relations of equality $=_{\mathsf{il}}$ and $=_{\mathsf{ol}}$ as an example of predicates respectively defined on lists and arrays, and the predicate mapping $\phi_1 = \{(=_{\mathsf{il}}, =_{\mathsf{ol}})\}$. The representation mapping $map_1$, introduced in example 3, is $\phi_1$*-formal* because two lists are equal with $=_{\mathsf{il}}$ (composed by the same integers) if and only if their representations are equal with $=_{\mathsf{ol}}$.

As a counterexample we consider the representation mapping $map_2$ that associates a list to an array, but whose elements are in reverse order.

- $eq_{head}(l, l') \equiv (head(l) = head(l'))$,
- $eq_{elt}(t, t') \equiv (t[0] = t'[0])$

When considering the predicate mapping $\phi_2 = \{(eq_{head}, eq_{elt})\}$. The representation mapping $map_2$ is not $\phi_2$*-formal* because we can construct two lists $l_1 = (1, 2), l_2 = (1, 3)$ such that $eq_{head}(l_1, l_2)$ is true but $eq_{elt}(\phi(l_1), \phi(l_2)) = eq_{elt}([2, 1], [3, 1])$ is false.

Given a representation mapping $\lceil\ \rceil$, we can simulate the behavior of $\mathsf{il}$ programs in the $\mathsf{ol}$ environment. Suppose we have a big-step semantics for each language (with their respective reduction relation $bs_{\mathsf{ol}}$ and $bs_{\mathsf{il}}$ in their respective set of environments $\mathcal{E}nv_{\mathsf{ol}}$ and $\mathcal{E}nv_{\mathsf{il}}$).

To define the evaluation of $\mathsf{il}$ programs in an $\mathsf{ol}$ environment $\epsilon_{\mathsf{ol}} \in \mathcal{E}nv_{\mathsf{ol}}$, we need to translate $\epsilon_{\mathsf{ol}}$ in an $\mathsf{il}$ environment $\epsilon_{\mathsf{il}} \in \mathcal{E}nv_{\mathsf{il}}$. Therefore, we extend the representation mapping to environments.

**Definition 5.** *The extension of the representation mapping to environments also noted $\lceil\ \rceil \in \mathcal{E}nv_{\mathsf{il}} \rightarrow \mathcal{E}nv_{\mathsf{ol}}$ is such that:*

$$\forall \epsilon_{\mathsf{il}} \in \mathcal{E}nv_{\mathsf{il}}, \forall x \in \mathcal{X}_{\mathsf{il}}, \forall v \in \mathcal{O}_{\mathsf{il}},\ \langle x, v \rangle \in \epsilon_{\mathsf{il}} \Leftrightarrow \langle x, \lceil v \rceil \rangle \in \lceil \epsilon_{\mathsf{il}} \rceil$$

Even if $\lceil\ \rceil$ is total and injective, defining the inverse function $\lceil\ \rceil^{-1} \in \mathcal{E}nv_{\mathsf{ol}} \to \mathcal{E}nv_{\mathsf{il}}$ is not immediate because $\lceil\ \rceil^{-1}$ may not be total. To obtain a total function, we extend the inverse with the empty environment for $\mathsf{ol}$ environments that are not in the domain of the inverse.

**Definition 6.** *Given a representation mapping $\lceil\ \rceil \in \mathcal{E}nv_{\mathsf{il}} \to \mathcal{E}nv_{\mathsf{ol}}$, $\lceil\ \rceil^{-1} \in \mathcal{E}nv_{\mathsf{ol}} \to \mathcal{E}nv_{\mathsf{il}}$ is such that $\forall \epsilon_{\mathsf{ol}} \in \mathcal{E}nv_{\mathsf{ol}}$:*

- *if $\epsilon_{\mathsf{ol}} \in \mathtt{range}(\lceil\ \rceil)$, $\lceil \epsilon_{\mathsf{ol}} \rceil^{-1} = \epsilon_{\mathsf{il}}$ such that $\lceil \epsilon_{\mathsf{il}} \rceil = \epsilon_{\mathsf{ol}}$,*
- *if $\epsilon_{\mathsf{ol}} \notin \mathtt{range}(\lceil\ \rceil)$, $\lceil \epsilon_{\mathsf{ol}} \rceil^{-1} = \{\}$ where $\{\}$ is the empty relation.*

We simulate the reduction of $\mathsf{il}$ programs in an $\mathsf{ol}$ environment with the reduction relation of $\mathsf{il}$ by translating the $\mathsf{ol}$ environment with $\lceil\ \rceil^{-1}$. The semantics rules of $\mathsf{ol}$, extended with mapped $\mathsf{il}$ rules, give a semantics to the extended language.

**Definition 7.** *Given two semantics $bs_{\mathsf{il}}$ and $bs_{\mathsf{ol}}$ respectively defined by sets of inference rules $\mathcal{R}_{\mathsf{il}}$ and $\mathcal{R}_{\mathsf{ol}}$, we define the semantics of $\mathsf{oil}$ as:*

- *the reduction relation $bs_{\mathsf{oil}} = bs_{\mathsf{ol}}$,*
- *the set of inference rules $\mathcal{R}_{\mathsf{oil}} = \mathcal{R}_{\mathsf{ol}} \cup \mathcal{R}'_{\mathsf{il}} \cup \{r_1, r_2\}$ where*
  - *$\mathcal{R}'_{\mathsf{il}} = \mathcal{R}_{\mathsf{il}}$ where $\langle \epsilon, i \rangle \mapsto_{bs_{\mathsf{il}}} \epsilon'$ is replaced by $\langle \epsilon, \delta, i \rangle \mapsto_{bs_{\mathsf{il}}} \langle \epsilon', \delta \rangle$ ($\delta \in \mathcal{E}nv_{\mathsf{ol}}$ and $\epsilon, \epsilon' \in \mathcal{E}nv_{\mathsf{il}}$),*
  - *the inference rules $r_1$ and $r_2$:*

$$\frac{\langle \lceil \epsilon \rceil^{-1}, \gamma(\epsilon), i \rangle \mapsto_{bs_{\mathsf{il}}} \langle \epsilon', \delta \rangle}{\langle \epsilon, i \rangle \mapsto_{bs_{\mathsf{ol}}} \lceil \epsilon' \rceil \cup \delta}\ r_1 \qquad\qquad \frac{\langle \lceil \epsilon \rceil \cup \delta, i \rangle \mapsto_{bs_{\mathsf{ol}}} \epsilon'}{\langle \epsilon, \delta, i \rangle \mapsto_{bs_{\mathsf{il}}} \langle \lceil \epsilon' \rceil^{-1}, \gamma(\epsilon') \rangle}\ r_2$$

  *where $\gamma(x) = x - \lceil \lceil x \rceil^{-1} \rceil$ denotes the elements of $x$ which cannot be represented in $\mathcal{O}_{\mathsf{ol}}$.*

The introduction of $\delta$ in the rules of the $\mathsf{il}$ semantics is required for the reduction of lakes. Indeed, we need to keep track of $\mathsf{ol}$ environment when we evaluate an island. Otherwise the lakes would be kept separated from Ocean constructs, and no reference to ocean variables would be possible. The inference rule $r_1$ and $r_2$ link the two semantics: $r_1$ is the bridge from $\mathsf{ol}$ to $\mathsf{il}$ semantics for island evaluation and $r_2$ is the bridge from $\mathsf{il}$ to $\mathsf{ol}$ semantics for the lakes evaluation.

We notice that in $r_1$, $\lceil \epsilon' \rceil \cup \delta$ is a function only if $\mathtt{dom}(\lceil \epsilon' \rceil) \cap \mathtt{dom}(\delta) = \emptyset$. This condition means that a variable cannot represent both an Island and Ocean objects in the same environment. The rule $r_2$ introduces a similar condition. From now on, we consider semantics that verify these two conditions. In practice, it means that islands and lakes have to introduce fresh variables with respect to both $\mathsf{ol}$ and $\mathsf{il}$ environments.

Figure 3 presents how the Ocean and Island semantics are linked by the representation mapping. This shows how $\mathsf{il}$ instructions can be evaluated inside the evaluation of $\mathsf{ol}$ programs. The function $\lceil\ \rceil^{-1}$ gives the corresponding $\mathsf{il}$ environment restricted to $\mathsf{ol}$ objects which are Island object's representations, then the $\mathsf{il}$ construction is evaluated in $\mathsf{il}$ semantics, we obtain a new environment that can be mapped by $\lceil\ \rceil$ to give the target environment in $\mathsf{ol}$ semantics.

The objects in $\epsilon$ that cannot be represented in the Island $(\epsilon' - \lceil\lceil\epsilon'\rceil^{-1}\rceil)$ are given in parameter to Island evaluation in case of lakes. That's why $\epsilon'$ corresponds to the union of part of $\epsilon$ not represented in the Island but that can be modified by lakes (which corresponds to $\delta$) and the representation in the Ocean of the evaluation of il instructions (which corresponds to $\lceil\epsilon'\rceil$).

$$\begin{tikzcd}
\langle\lceil\epsilon\rceil^{-1}, \epsilon - \lceil\lceil\epsilon\rceil^{-1}\rceil, i\rangle \arrow[r, "bs_{il}"] & \langle\epsilon', \delta\rangle \arrow[d, "\lceil\ \rceil"] \\
\langle\epsilon, i\rangle \arrow[u, "\lceil\ \rceil^{-1}"] \arrow[r, "bs_{oil}"] & \lceil\epsilon'\rceil \cup \delta
\end{tikzcd}$$

**Fig. 3.** Reduction of il construction in ol semantics

Finally, the semantics of the language oil, combination of ol and il, is function of big-step semantics $bs_{\mathsf{ol}}$ and $bs_{\mathsf{il}}$, the representation mapping $\lceil\ \rceil$.

We can now use the Island formalism to extend the ol language by new constructs. In the following, we will see how to implement this idea in practice: for this new language, instead of building a new compiler from scratch, we consider a *dissolution* phase which replaces Islands constructs by Ocean constructs. With such an approach, an existing Ocean compiler could be reused. This induces in particular that the use of the Island formalism does not induce a dependence of the user on the island language and tools: after dissolution, the user is again in its original ocean language and can take the benefit of the generated code without depending on run-time libraries or Island language update and maintenance.

## 4 Dissolution

At the syntax level, the dissolution step consists of replacing all the il constructs that appear in the ol AST by ol constructs, in order to obtain a complete ol AST.

**Definition 8.** *Given two grammars $\mathcal{G}_{il}$ and $\mathcal{G}_{ol}$, we call* dissolution *a function* $\mathtt{diss}$ : $\mathit{AST}(\mathcal{G}_{il}) \rightarrow \mathit{AST}(\mathcal{G}_{ol})$.

*Such a function is said* lake preserving *when* $\forall i \in \mathit{AST}(\mathcal{G}_{il}), \forall l \in \mathtt{lakes}(i)$*, we have* $l \in \mathtt{lakes}(\mathtt{diss}(i))$*, where* $\mathtt{lakes}$ *is a function that gives the set of lakes contained in an AST (i.e an il construct).*

In practice, the condition of *lake-preserving* is verified by constructing with the same strategy (for example top-down) a list of lakes in the source and target program and the condition consists simply to test the equality of the two lists. Finding lakes in a dissolved program can be realized by marking generated code during dissolution in order to distinguish lakes from generated code in the target program. With this definition, we preserve the order of lakes. In comparing lists modulo a theory, we can obtain softer conditions. For example, with AC (Associative-Commutative) theory, we authorize lakes to be swapped in the Island. With idempotency, lakes can be duplicated.

*Example 5.* Considering again the previously introduced program:
`t=array(5); t[0]=3; t[1]=7;`
`l←cons(t[1],cons(t[2],nil)); x=l[1]; y=head(l)`,
we can distinguish three ol islands:
`l←cons(t[1],cons(t[2],nil))`, `l` (from `l[1]`), and `head(l)`.
The dissolution of these islands could (depending on the implementation) result in the following program:
`t=array(5); t[0]=3; t[1]=7; x=t[0];`
`l=array(2); l[0]=t[1]; l[1]=t[2]; x=l[1]; y=l[0];`.

In term of semantics, the ol constructs that are generated must have the same evaluation as the il constructs that they replace.

**Definition 9.** *Given a representation mapping* $\lceil\ \rceil$, *a* dissolution *function* $\texttt{diss}$ *is* well-formed *if:*

- *for every* $i \in \mathsf{AST}(\mathcal{G}_{\mathsf{il}})$, *for every environment* $\epsilon \in \mathcal{E}nv_{\mathsf{il}}$, *we have:* $\langle \epsilon, i \rangle \mapsto_{bs_{\mathsf{oil}}} \epsilon' \Leftrightarrow \langle \lceil \epsilon \rceil, \texttt{diss}(i) \rangle \mapsto_{bs_{\mathsf{ol}}} \epsilon'$,
- *the* ol *program resulting from dissolution is syntactically correct. More formally,* $\forall i \in \mathsf{AST}(\mathcal{G}_{\mathsf{il}}), \texttt{getSort}(i) \in \texttt{anch}(\texttt{getSort}(\texttt{diss}(i)))$,
- *the dissolution function is lake preserving.*

Figure 4 shows the link between the evaluation of an il instruction with il semantics (as in Fig. 3) and the execution of the corresponding ol instruction (by dissolution). The states after evaluation are the same.

To summarize, an Island language should fulfill the following requirements:

**Definition 10.** *Given two languages* ol *and* il *described by a grammar, a big-step semantics, a set of objects, and a set of predicates in these objects,* il is an Island language for ol *if there exist:*

- *a syntactic anchor* $\texttt{anch}$, *either simple or with lakes (Definition 1),*
- *a representation mapping* $\lceil\ \rceil$ *for objects (Definition 2),*
- *a dissolution function* $\texttt{diss}$ *(Definition 8).*

## 5 Formal Islands

**Definition 11.** *Given two languages* ol *and* il, il *is a* Formal Island *over* ol *if:*

1. il *is an Island for* ol *(Definition 10),*

$$\begin{tikzcd}
\langle \lceil \epsilon \rceil^{-1}, \epsilon - \lceil \lceil \epsilon \rceil^{-1} \rceil, i \rangle \arrow[r, "bs_{il}"] & \langle \epsilon', \delta \rangle \arrow[d, "\lceil\ \rceil"] \\
\langle \epsilon, i \rangle \arrow[u, "\lceil\ \rceil^{-1}"] \arrow[r, "bs_{oil}"] \arrow[d, "\mathsf{diss}"] & \lceil \epsilon' \rceil \cup \delta \arrow[d, leftrightarrow, "="] \\
\langle \epsilon, \mathsf{diss}(i) \rangle \arrow[r, "bs_{ol}"] & \lceil \epsilon' \rceil \cup \delta
\end{tikzcd}$$

**Fig. 4.** Reduction of a $\mathsf{il}$ dissolution in $\mathsf{ol}$ semantics

2. *there exists a predicate mapping $\phi$ for objects (Definition 3) such that the representation mapping $\lceil\ \rceil$ is $\phi$-formal (Definition 4),*
3. *the dissolution function $\mathtt{diss}$ is well-formed (Definition 9).*

Condition 1 is purely syntactic and simple to verify. Condition 3 is similar to the correctness of a compilation process and condition 2 is more specific to the Island formalism and we are not aware of any example for which it has been already *formally* verified. This definition of formal islands allows us to ensure the preservation of properties.

First, the environment is extended by morphism on first-order formulae:

$$\begin{array}{l|l}
\multicolumn{2}{l}{\forall p \in \mathcal{P}_{\mathsf{il}}, \forall t_1, \dots, t_n, \epsilon(p(t_1, \dots, t_n)) = \epsilon(p)(t'_1, \dots, t'_n)} \\
\multicolumn{2}{l}{\quad \text{where } t'_i = \epsilon(t_i) \text{ if } t_i \in \mathcal{X}_{\mathsf{il}} \text{ and } t_i \text{ otherwise } (\textit{i.e. } t_i \text{ is an object}),} \\
\epsilon(\forall x\ P) = \forall x\ \epsilon(P), & \epsilon(\exists x\ P) = \exists x\ \epsilon(P), \\
\epsilon(P_1 \lor P_2) = \epsilon(P_1) \lor \epsilon(P_2), & \epsilon(P_1 \land P_2) = \epsilon(P_1) \land \epsilon(P_2), \\
\epsilon(\neg P) = \neg \epsilon(P), & \epsilon(P_1 \rightarrow P_2) = \epsilon(P_1) \rightarrow \epsilon(P_2).
\end{array}$$

From this definition, we note $\epsilon \models pre \Leftrightarrow \epsilon(pre)$.

**Proposition 1.** *Given a formal island $\mathsf{il}$ over $\mathsf{ol}$ and $pre, post$ two first-order formulae build over $\mathcal{P}_{\mathsf{il}}$ predicates, $\forall i \in \mathtt{dom}(\mathtt{diss}), \epsilon \in \mathcal{E}nv_{\mathsf{il}}$, we have:*

$$\epsilon \models \{pre\} i \{post\} \Leftrightarrow \lceil \epsilon \rceil \models \{\phi(pre)\} \mathtt{diss}(i) \{\phi(post)\}$$

*Proof.* By induction on the structure of the formulae *pre* and *post*. We prove that for all environment $\epsilon$:

$$\begin{array}{l}
\epsilon \models pre \Rightarrow \epsilon' \models post \ \mathtt{where}\ \langle \epsilon, i \rangle \mapsto_{bs_{\mathsf{il}}} \epsilon' \\
\Leftrightarrow \\
\lceil \epsilon \rceil \models \phi(pre) \Rightarrow \epsilon'' \models \phi(post) \ \mathtt{where}\ \langle \lceil \epsilon \rceil, i \rangle \mapsto_{bs_{\mathsf{ol}}} \epsilon''
\end{array}$$

Given an environment $\epsilon$, in a first step, we prove that $\epsilon \models \mathit{pre} \Leftrightarrow \lceil \epsilon \rceil \models \phi(\mathit{pre})$ by induction on the structure of *pre*. The base case for this induction is $\mathit{pre} = p(t_1, \ldots, t_n)$. $\lceil \epsilon \rceil \models \phi(\mathit{pre}) \Leftrightarrow \phi(p)(t'_1, .., t'_n)$ where $t'_i = \lceil \epsilon \rceil(t_i)$ when $t_i \in \mathcal{X}$, and $t'_i = \lceil t_i \rceil$ otherwise. Note that every $t'_i \in \mathcal{O}_{\mathsf{il}}$, and since the representation mapping is $\phi$-formal, we have $\phi(p)(t'_1, .., t'_n) \Leftrightarrow p(t''_1, \ldots, t''_n)$ with $t'_i = \lceil t''_i \rceil$. We can deduce that $t''_i = \epsilon(t_i)$ when $t_i \in X$, and $t''_i = t_i$ otherwise. We deduce that $p(t''_1, \ldots, t''_n) \Leftrightarrow \epsilon(\mathit{pre})$. By transition of the equivalence, we conclude that $\epsilon \models \mathit{pre} \Leftrightarrow \lceil \epsilon \rceil \models \phi(\mathit{pre})$ for $\mathit{pre} = p(t_1, \ldots, t_n)$.

We continue by induction on the *pre* structure. We just give the case $\mathit{pre} = P_1 \vee P_2$, the others being quite similar. By induction hypothesis, we know that it is true for $\mathit{pre} = P_1$ and $\mathit{pre} = P_2$ and we want to prove it for $\mathit{post} = P_1 \vee P_2$. By definition of the extension of the environment to first order formulae, if $\mathit{pre} = (P_1 \vee P_2)$ is valid, then either $P_1$ or $P_2$ is and therefore by induction hypothesis, $\phi(P_1)$ or $\phi(P_2)$ are true and therefore so is $\phi(P_1 \vee P_2)$. So $\epsilon \models \mathit{pre} \Leftrightarrow \lceil \epsilon \rceil \models \phi(\mathit{pre})$ (1).

Similarly, we prove that $\epsilon' \models \mathit{post} \Leftrightarrow \epsilon'' \models \phi(\mathit{post})$ by induction on the structure of *post*. The proof of the base case is similar to the proof for *pre* with one difference: $\epsilon'' \neq \lceil \epsilon' \rceil$. But as the dissolution function is well-formed, we know that $\epsilon'' = \lceil \epsilon' \rceil \cup \delta$. When $t_i \in \mathtt{dom}(\epsilon')$, we have $t_i \in \mathtt{dom}(\lceil \epsilon' \rceil)$. Since $\epsilon''$ is a function, we deduce that $\epsilon''(t_i) = \lceil \epsilon' \rceil(t_i)$. Due to this property, we can prove the base case as for *pre*. The step case of the induction on *post* structure is similar to the induction for *pre*. We finally obtain $\epsilon' \models \mathit{post} \Leftrightarrow \epsilon'' \models \phi(\mathit{post})$ (2).

From the equivalences (1) and (2), we directly deduce the proposition.

## 6 Domain Specific Languages implemented by Formal Island

A *Domain Specific Language* (DSL) is a programming language designed for a very specific task or domain, contrary to general programming languages like Java. A few papers give an overview on the Domain Specific Languages implementation methodology [11, 6, 4, 12, 10]. Summarizing the main ideas, this can be achieved by language specialization (removing features of an existing language), language extension (adding new features to an existing language), language invention (designed from scratch with no commonality with existing languages) or piggyback (using partially an existing language). Some works on modular and extensible semantics [3, 13] are well-tailored for DSL specifications.

In [11], Spinellis proposes eight recurring patterns to classify DSL design and implementation. One of this pattern is the *piggyback pattern* which corresponds informally to the design of Island languages. The piggyback structural pattern uses the capabilities of an existing language to be a hosting base for a new DSL. Thus, the DSL shares common elements with an existing language and is compiled in the host language.

In the classification of [6], the patterns correspond to different phases of DSL development: decision, analysis, design and implementation. Formal Island gets involved in two related phases of DSL development: the design and implementation phases. Formal Islands correspond in terms of design patterns to the *Language exploitation* (i.e. based on an existing language) and in terms of implementation pattern to the *Embedding* and *Preprocessor* patterns.

To illustrate the link between DSL and Formal Islands, we will detail the language SQLJ [2, 7], an example of DSL implemented using the piggyback pattern. SQLJ is an interface to the JDBC domain-specific-library: it hides the complexity of the API.

To avoid grammar conflicts and make identification of constructs easier, each SQLJ constructs starts with #sql token (which is not a legal Java identifier). The simplest SQLJ executable clauses consist of the token #sql followed by a SQL statement enclosed in *curly braces*. For example, the following SQLJ clause may appear instead of a Java instruction.

```
public void honors(float limit) {
  #sql{
    SELECT STUDENT AS "name", SCORE AS "grade"
    FROM   GRADE_REPORTS
    WHERE  SCORE >= :limit
  }
}
```

The SQL statements can contain variable names that correspond to Java variables (the variable `limit` for example). These variables are prefixed by a colon and they correspond to the notion of *lake* introduced previously.

*Syntactic anchor.* Assuming that we have a grammar of Java where $\langle \mathit{Statement} \rangle$ and $\langle \mathit{Instruction} \rangle$ are Java non-terminals, we can define the following simple syntactic anchor $\mathtt{anch} = \{(\langle \mathit{Statement} \rangle ::= \langle \mathit{Declaration} \rangle), (\langle \mathit{Instruction} \rangle ::= \langle \mathit{ExecutableStatement} \rangle)\}$. The non-terminal $\langle \mathit{Declaration} \rangle$ corresponds to SQLJ declarations used to initialize a JDBC connection. As illustrated above, $\langle \mathit{ExecutableStatement} \rangle$ corresponds to embedded SQL queries.

*Data-structure anchor.* In SQLJ, the representation mapping between SQL objects and Java objects is given by conversions from SQL types to Java types. For example, the SQL CHAR type is converted into a Java String. Therefore, the results of a SQL query have to be translated into Java objects before being store in Java variables.

*Dissolution.* In the SQLJ formalism, the SQL language is not really the embedded language because this is not the SQL requests which are dissolved in Java, but rather the SQLJ instructions which contain SQL requests. The SQLJ pre-processor provides type-checking and schema-object-checking to detect syntax errors and missing or misspelled object errors in SQL statements at translation time rather than at runtime (like in JDBC). Programs written in SQLJ are, therefore, more robust than JDBC programs. We just give the intuition of the translation step by giving the dissolution in Java of the program given previously:

```
public void honors(float limit) {
  java.sql.PreparedStatement ps = recs.prepareStatement(
      "SELECT STUDENT, SCORE "
    + "FROM   GRADE_REPORTS "
    + "WHERE  SCORE >= ? ");
  ps.setFloat(1, limit);
  ps.executeQuery();
}
```

The object recs is a JDBC connection of type java.sql.Connection. The SQLJ translator verifies that in the SQLJ statement, limit is of type float in order to be compared with SCORE, whose SQL type is REAL.

In the case of SQLJ, there is no formal property given for the mapping between types or for the compilation of the SQLJ instructions. We cannot ensure that the Java compiled code is consistent. Our framework is a base to formalize DSL implemented using the piggyback pattern. It gives conditions to ensure that properties established at the DSL are preserved by compilation.

## 7 Tom: a Formal Island for Pattern-Matching

An other example of Island language is Tom, which adds pattern-matching facilities to imperative languages such as C and Java. Indeed, it is in this context that we identified the need to have a notion of *formal Island framework*. This helps up to understand how properties of Tom can be preserved by compilation.

As presented in [9], a Tom program is a program written in a host language and extended by several new constructs. Due to lack of space, we cannot present the language in detail. In the following, it is sufficient to consider that Tom provides three main constructs:

- %op allows to define an algebraic signature (i.e. names of constructors with their profile),
- %match corresponds to an extension of switch/case, well known in functionnal programming languages,
- ‘ allows to build an algebraic term from the host language.

Therefore, a program can be seen as a list of Tom constructs (the Islands) interleaved with some sequences of characters (the Ocean). During the compilation process, all Tom constructs are dissolved and replaced by instructions of the host-language, as it is usually done by a preprocessor. From this point, we consider that the Ocean language is Java and we call JTom this specialized version of Tom.

The following example shows how a simple symbolic computation (addition) over Peano integers can be defined. This supposes the existence of a data-structure and a mapping (defined using %op) where Peano integers are represented by *zero* and *successor*: the integer 3 is denoted by *suc(suc(suc(zero)))* for example.

```
public class PeanoExample {
  %op Term zero() { ... }
  %op Term suc(Term) { ... }
  ...
  Term plus(Term t1, Term t2) {
    %match(t1, t2) {
      x,zero   -> { return ‘x; }
      x,suc(y) -> { return ‘suc(plus(x,y)); }
    }
  }
  void run() {
    System.out.println("plus(1,2) = " + plus(‘suc(zero),‘suc(suc(zero))));
  }
}
```

In this example, given two terms $t_1$ and $t_2$ (that represent Peano integers), the evaluation of `plus` returns the sum of $t_1$ and $t_2$. This is implemented by pattern matching: $t_1$ is matched by $x$, $t_2$ is possibly matched by the two patterns $zero$ and $suc(y)$. When $zero$ matches $t_2$, the result of the addition is $x$ (with $x = t_1$, instantiated by matching). When $suc(y)$ matches $t_2$, this means that $t_2$ is rooted by a $suc$ symbol: the subterm $y$ is added to $x$ and the successor of this number is returned, using the ‘ construct. The definition of `plus` is given in a functional programming style, but the `plus` function can be used in Java to perform computations. This example illustrates how the `%match` construct can be used in conjunction with the considered native language. We can notice that JTom programs contain lakes (the right part of a rule is a Java statement). Note also that lakes can contains Islands, introduced by ‘ for example.

From the definition of formal islands (Definition 11), we define for JTom the syntactic anchor, the representation mapping, the predicate mapping, and gives the intuition of the dissolution function which corresponds to the Tom compiler task.

### 7.1 Syntactic anchor

In the case of JTom, the syntactic anchor `anch` is defined as follow:

$$\mathtt{anch} = \left\{ \begin{array}{c} (\langle \mathit{Statement} \rangle ::= \langle \mathit{OpConstruct} \rangle), \\ (\langle \mathit{Instruction} \rangle ::= \langle \mathit{MatchConstruct} \rangle), \\ (\langle \mathit{Expression} \rangle ::= \langle \mathit{BackQuoteConstruct} \rangle) \end{array} \right\}$$

### 7.2 Data-representation anchor

In JTom, the notion of *term* can be implemented by any data-structure. Once given such an implementation, the data-representation anchor can be defined. Let us consider that terms are implemented using a record `(sym:integer, sub:array of term)`, where the first slot (`sym`) denotes the top symbol, and the second slot (`sub`) corresponds to the subterms. It is easy to check that the following definition of the predicate mapping provides a *formal anchor*:

$$\begin{aligned} \mathtt{eq}(\mathtt{t}_1, \mathtt{t}_2) &\triangleq \lceil \mathtt{t}_1 \rceil.\mathtt{sym} = \lceil \mathtt{t}_2 \rceil.\mathtt{sym} \wedge \forall i \in [1..\mathrm{ar}(\lceil \mathtt{t}_1 \rceil.\mathtt{sym})], \\ &\quad\ \mathtt{eq}(\mathtt{t}_1.\mathtt{sub}[i], \mathtt{t}_2.\mathtt{sub}[i]) \\ \mathtt{is\_fsym}(\mathtt{t}, \mathtt{f}) &\triangleq \lceil \mathtt{t} \rceil.\mathtt{sym} = \lceil \mathtt{f} \rceil \end{aligned}$$

The first definition says that two terms are equal if the representation of their root symbol are equal. In addition, the subterms have to be respectively equal. The second definition says that a term $t$ is rooted by $f$ if the representation of $t$ (which is a record) has the representation of $f$ as first element.

### 7.3 Dissolution

Due to lack of space, we cannot give in detail complete definition of the dissolution function which corresponds to the compilation phase. Therefore, we just give the intuition of the translation step by illustrating the dissolution of the PeanoExample program given previously.

```
public Term plus(Term t1, Term t2) {
  Term tom_x = t1;
  if (is_fsym_zero(t2)) {
    return tom_x;
  } else if (is_fsym_suc(t2)) {
    Term tom_y = subterm_suc(t2,1);
    return make_suc(plus(tom_x,tom_y));
  }
}
```

With these definitions, Tom is an island for Java. We have proved that the anchor is formal. The last condition to obtain a formal island is the proof that the dissolution is well-formed. As shown in [5], a first step in this direction is the development of a certifying compiler which proves, for each compilation, that the dissolution preserves the semantics of the pattern-matching.

## 8 Conclusion and Future works

We have defined the notion of Formal Island to provide a formal framework allowing language designers to base their languages extensions. For this framework to back-up properties proofs, e.g. about safety or security, we have shown that under sufficient conditions, properties established at the Island level are preserved once dissolved into the host language. We have then shown application of this framework to DSL like SQLJ and to Tom.

Amongst the many applications that we can envision, the safe treatment of XML transformations, via appropriate Java based Islands, is particularly promising and is currently under development.

Of course such a framework should be closely linked to proving tools adapted to the properties to be checked: another direction that we are also investigating.

## References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. 1998.

[2] Gray Clossman, Phil Shaw, Mark Hapner, Johannes Klein, Richard Pledereder, and Brian Becker. Java and relational databases (tutorial): SQLJ. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 500. ACM, 1998.

[3] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.

[4] Conal Elliott, Sigbjorn Finne, and Oege De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[5] Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal validation of pattern matching code. In Pedro Barahona and Amy Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 187–197. ACM, 2005.

[6] Anthony M. Sloane Marjan Mernik, Jan Heering. When and how to develop domain-specific languages. Technical report, CWI, 2003.

[7] Jim Melton and Andrew Eisenberg. *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*. Morgan-Kaufmann, 2000.

[8] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, 2001.

[9] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, 2003.

[10] Tim Sheard, Zine-El-Abidine Benaissa, and Emir Pasalic. DSL implementation using staging and monads. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 81–94. The USENIX Association, 1999.

[11] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[12] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[13] Keith Wansbrough and John Hamer. A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association, 1997.