SOMETHINGit : A Prototyping Library for Live and Sound Improvisation

Tomohiro Oda\textsuperscript{1}  
Kumiyo Nakakoji\textsuperscript{1}  
Yasuhiro Yamamoto\textsuperscript{2}  
\textsuperscript{1}Key Technology Laboratory  
Software Research Associates, Inc.  
Tokyo, Japan  
tomohiro@sra.co.jp  
\textsuperscript{2}Precision and Intelligence Laboratory  
Tokyo Institute of Technology  
Yokohama, Japan  
kumiyo@acm.org  
yxy@acm.org

Abstract—Live programming can be considered an interaction with incomplete code. Dynamic languages embrace the similar style of programming, such as pair programming and prototyping in a review session. Static languages require a certain degree of completeness of code, such as type safety and namespace resolution. SOMETHINGit is a Smalltalk library that combines dynamic Smalltalk and static Haskell and VDM-SL. SOMETHINGit enables programmers to write incomplete but yet partially mathematically sound programs by five levels of bridging mechanisms.

Index Terms—Live Programming, Prototyping, Smalltalk, Sketch

I. INTRODUCTION

In live programming, a performer shares a transient situation in programming or performance with co-workers or audiences. Because the act of live programming is an on-going process of producing code, the code is always incomplete during the session. Incomplete code is the artifact at hand for a live programmer.

The development of a software system at large deals with both the world of making and that of using [1]. The world of making is concerned with molding, constructing and building. The world of using is concerned with engaging, experiencing, and interacting-with. The world of making and the world of using contact with each other at the execution of programs. Live programming is a unique situation, in which the world of making is identical to the world of using. In contrast, the most industrial programming tasks separate the two.

Pair programming in Agile development and prototyping in a review session can be seen as approaches to bridge the world of making and the world of using. For example, prototyping in a review session with a customer representative involves writing pieces of code when the customer representative describes the situation of using the system. In such sessions, it is not very important to make the code complete. It is, instead, essential to make it “work” at the moment. During a live programming session, the working code is hot-fixed without halts, and yet, such incompletely hot-fixed code must continue running. The edit-compile-run cycle could become disadvantage when the cycle is exposed to a live programmer or a live performer.

Another important property of languages for Live, Pair, or Prototype programming is to have intuitive user interfaces. The Smalltalk environment [2] is a pioneer of graphical user interfaces (GUIs) directed toward Dynabook [3] where everyone at any age can enjoy live learning and live programming.

II. IMPROVISATIONAL DESIGN IN SOFTWARE DEVELOPMENT

Acts of design in early stages, such as interface sketching, require a close interaction between a designer and a design artifact [4][5]. Sketches are then narrowed down to more didactic, assertive and refined representations such as prototypes and blueprints. Noncommittal suggestions and tentative proposals are grounded to depictive descriptions and specific tests [6]. Improvisations emerged in software development are thus subject to be recorded and grounded to specific and sound design decisions.

Design also requires the other direction. Specific and sound design artifacts, such as design patterns, domain-specific knowledge, and the repertoire of reusable components, often need to be referred in improvisation, yet keeping freedom from unnecessary constraints. Improvisation and soundness are not binary choices, but depend on each other.

III. THE SOMETHINGit LIBRARY

SOMETHINGit is a Smalltalk library to communicate with the REPL (Read-Eval-Print Loop) interpreters of different languages. The library currently supports Haskell (GHCi) [7] and VDM-SL(VDMJ) [8][9].

Smalltalk and LISP are the two prominent programming languages (as well as environments and libraries) that pioneered interactive and dynamic programming. Many essences of the eXtreme Programming practice came from the Smalltalk community including pair programming and prototyping in customers’ review.

Smalltalk and LISP employ dynamic typing which bring flexibility and simple yet strong meta-expressiveness and reflection mechanisms. For example, it is a common habit among Smalltalk programmers to run a piece of code with a not-implemented-yet method; they write the method only when the system execution calls the method. The code still continues running after defining the new methods.

On the one hand, such flexibility in terms of implementation and execution gives positive impacts to the improvisational style of coding. On the other hand, the flexible
ordering of implementation and execution often sacrifices the soundness of the code, such as type safety and static bindings.

Smalltalk programmers make heavy use of the Workspace, in which programmers write a piece of Smalltalk code and make notes in a natural language. A programmer opens a Workspace, writes some pieces of code among notes, and selects a piece of code to perform Dolt, PrintIt or InspectIt to execute the code. A Workspace gives a flexible choice of orders of implementation and execution to the programmer than REPL interpreters.

SOMETHINGIt is a library to use more sound and rigorous languages like Haskell and VDM-SL in the flexible Smalltalk language/environment/library (See Table 1). A programmer may write pieces of Haskell code among Smalltalk code and evaluate a piece of Haskell code specified by the user with GUI. Another programmer may also call a VDM function from Smalltalk code. It is also possible to manipulate states of a running VDM-SL specification with GUI. SOMETHINGIt makes it possible to write soundness-critical code in Haskell or VDM-SL and flexibility-critical code in Smalltalk. SOMETHINGIt has been designed to provide interactive, flexible and improvisation programming interfaces, both API and GUI, to Haskell and VDM-SL.

<table>
<thead>
<tr>
<th>Name</th>
<th>Smalltalk</th>
<th>Haskell</th>
<th>VDM-SL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Style</td>
<td>Dynamic environment</td>
<td>Compiler + REPL interpreter</td>
<td>IDE with a REPL interpreter</td>
</tr>
<tr>
<td>Paradigm</td>
<td>Object-Oriented Programming</td>
<td>Functional Programming</td>
<td>Formal Methods</td>
</tr>
<tr>
<td>Type system</td>
<td>Dynamic typing</td>
<td>Static typing</td>
<td>Static typing</td>
</tr>
<tr>
<td>UI</td>
<td>GUI</td>
<td>CUI</td>
<td>GUI (CUI interpreter)</td>
</tr>
<tr>
<td>Motto</td>
<td>Dynamism</td>
<td>Purity</td>
<td>Rigor</td>
</tr>
<tr>
<td>Code base</td>
<td>Image</td>
<td>File</td>
<td>File</td>
</tr>
<tr>
<td>Runtime</td>
<td>IDE=Runtime</td>
<td>Binary code</td>
<td>Animation on IDE</td>
</tr>
</tbody>
</table>

Table 1: A summary of comparison among Smalltalk, Haskell and VDM-SL

IV. BRIDGING: FIVE LEVELS OF BINDINGS

SOMETHINGIt has the following five levels of interfaces between Smalltalk and Haskell. The five levels of interfaces allow programmers to mix code in different languages rather than to make a binary choice between improvisation or soundness. There is also no need to split the whole program into the improvisation part and the soundness part. A programmer can mix improvisation and soundness through continuous gradation.

We have also developed a UI prototyping environment (Lively Walk-Through) by combining Smalltalk UI objects and VDM-SL executable specifications based on SOMETHINGIt. Using Lively Walk-Through, UI designers and formal methods engineers can collaboratively build a UI prototype in a live setting. The bindings between Smalltalk UI objects and VDM-SL operations and functions are built at the binding level 3 (closure-like objects) described below.

A. Level 1: GUI (Workspace and Browsers)

SOMETHINGIt adds “Haskell It” and “inspect Haskell It” items to the context menu of standard Smalltalk Workspace. As the name says, HaskellIt evaluates and prints the result on the Workspace. In this paper, the result of an evaluation is denoted after the ➔ symbol.

\[
\text{take } 2 \ [4, 4, 1] \ ➔ \ [4, 4]
\]

B. Level 2: The “eval” Function

The SIExternalInterpreter class and its concrete subclasses provide “evaluate:” and similar messages.

\[
\text{SIHaskell evaluate: } \text{‘take 2 [4, 4, 1]’} \ ➔ \text{anOrderedCollection } (4 4 )
\]

C. Level 3: Closure-like Objects

“‘take’ asHaskellExpression” creates a Smalltalk closure-like object of the Haskell’s take function, which answers to the “value,” “value:,” and “value:value:” family of messages. The closure-like object can deal with higher order functions.

\[
\text{‘take’ asHaskellExpression}
\]

\[
\text{value: 2}
\]

\[
\text{value: #(4 4 1) asOrderedCollection} \ ➔ \text{anOrderedCollection } (4 4)
\]

\[
\text{take := ‘take’ asHaskellExpression.}
\]

\[
\text{take2 := take value: 2.}
\]

\[
\text{take2 value: #(4 4 1) asOrderedCollection} \ ➔ \text{anOrderedCollection } (4 4)
\]

D. Level 4: Object-Value Mapper

SOMETHINGIt comes with the following mapping between Haskell values and Smalltalk objects (Table 2).

The mapping can be overwritten or extended by a user programmer.

For example, Haskell’s list “[4, 4, 1]” will be mapped to an OrderedCollection object with 4, 4 and 1 in the order.

\[
\text{SIHaskell evaluate: } \‘[4, 4, 1]’ \ ➔ \text{anOrderedCollection } (4 4 1)
\]

Please note that the identity of Object which is mapped to SIObj ect in Haskell is kept, as demonstrated below.
A Point instance “1@2” is passed to Haskell, is put into a list, and is returned to the Smalltalk side as a member of the resulting OrderedCollection object. The first member is tested identical to the point object, resulting in “true”, which means the point object kept the identity even though it traveled to the Haskell side and then came back to the Smalltalk side.

### E. Level 5: Callback (Message Sending from Haskell)

An object translated into SIOBJECT can send a message in Haskell and the message is evaluated in Smalltalk.

```haskell
| hcode enlist p list |
| . hcode := SIHaskell source: 'enlist x = [x]' |
| . enlist := 'enlist' asHaskellExpressionIn: hcode. |
| . p := 1@2. |
| . list := enlist value: p. |
| . list first == p |
| . true |
```

In the above example, the “at” function is defined in Haskell using the “messageSend” function which performs a Smalltalk’s message sending to the first argument namely “array”. The Haskell “at” function is wrapped in the Smalltalk side by the “asHaskellExpressionTyped” message. The Smalltalk code then evaluate the “at” function with two arguments: the array “#(4 4 1)” and the index “3”. The Haskell “at” function then sends the “at:” message to the first argument (Smalltalk’s #(4 4 1)) with the argument “3”. The result of the message sending is the third member of the array, in this case, 1. The “messageSend” function is typed as an IO monad because its evaluation involves communication with the Smalltalk side. The result of message sending in Smalltalk is impure and thus it is reasonable to put it into an IO monad.

### V. Implementation

The SOMETHINGit library is built upon the Smalltalk system and communicates with external OS processes. Table 3 shows the summary of the operating environment. Major components of the SOMETHINGit library are as follows.

- **Smalltalk Library**
  - Process management
    - SIEExternalInterpreter
    - SIHaskell
    - SIVDM
  - Object-Value mapper
    - SIParser
    - SIHaskellParser
    - SIVDMParser
    - SIPrinter
    - SIHaskellPrinter
    - SIVDMPrinter
  - Value container
    - SIHaskellExpression
    - SIVDMExpression
  - User interface
    - Workspace (HaskellIt and inspectHaskellIt)
    - SIVDMBrowser
- **Haskell Library**
  - Type
    - SIObject (in Haskell)
  - Function
    - messageSend

### VI. Example: FizzBuzz Game

What follows is a piece of Haskell code for the famous “FizzBuzz” game.

An interactive FizzBuzz GUI can be mocked up in an interactive manner.

```haskell
data FizzBuzz = Fizz | Buzz | FizzBuzz | Number Int
fizzbuzz :: Int -> FizzBuzz
fizzbuzz x |
| . x `mod` 15 == 0 = FizzBuzz |
| . x `mod` 5 == 0 = Buzz |
```

Table 2: Mapping between Haskell Values and Smalltalk objects

<table>
<thead>
<tr>
<th>Language</th>
<th>Smalltalk</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>OrderedCollection</td>
<td>list</td>
</tr>
<tr>
<td>Set</td>
<td>Set</td>
<td>Data.Set</td>
</tr>
<tr>
<td>Map</td>
<td>Dictionary</td>
<td>Data.Map</td>
</tr>
<tr>
<td>Algebraic Data Type</td>
<td>SIA Algebraic Data</td>
<td>*</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Int</td>
</tr>
<tr>
<td>Decimal</td>
<td>Float</td>
<td>Fraction</td>
</tr>
<tr>
<td>Char</td>
<td>Character</td>
<td>Char</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>Smalltalk Object</td>
<td>Object</td>
<td>SIOBJECT</td>
</tr>
</tbody>
</table>

Table 3: SOMETHINGit Overview

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Linux or Mac OS X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smalltalk System</td>
<td>Squeak 4.3 or higher</td>
</tr>
<tr>
<td>Pharo 1.4 or higher</td>
<td></td>
</tr>
<tr>
<td>Required Smalltalk Packages</td>
<td>OSProcess</td>
</tr>
<tr>
<td>Haskell interpreter</td>
<td>GHCi 7.4.1 or higher</td>
</tr>
<tr>
<td>VDM-SL interpreter</td>
<td>VDMJ 2.0.1</td>
</tr>
</tbody>
</table>
The `fizzbuzz` function can be wrapped as a Smalltalk closure-like object by

```haskell
<table>
<thead>
<tr>
<th>hcode</th>
</tr>
</thead>
</table>
| hcode := 'fizzbuzz' asHaskellExpressionIn: hcode.
```

```haskell
data FizzBuzz = Fizz | Buzz | FizzBuzz | Number Int
deriving Show
fizzbuzz :: Int -> FizzBuzz
fizzbuzz x
| x `mod` 15 == 0 = FizzBuzz
| x `mod` 5 == 0 = Buzz
| x `mod` 3 == 0 = Fizz.
| otherwise = Number x'.
fizzbuzz := 'fizzbuzz' asHaskellExpressionIn: hcode.
```

Now the `fizzbuzz` closure can be evaluated by Smalltalk’s value: message.

- `fizzbuzz value: 1` ➔ `Number 1`
- `fizzbuzz value: 3` ➔ `Fizz`
- `fizzbuzz value: 30` ➔ `FizzBuzz`

A GUI for the game can be built with the Morphic Framework (see Fig.1).

![A GUI for FizzBuzz game](image)

**VII. RELATED WORK**

Squeak etoys [10] is a programming environment for children based on Squeak Smalltalk environment. Every “object” placed on the environment is programmable by choosing, arranging and editing visual tiles.

Scratch [11] is another visual programming environment for children in the spirit of turtle graphics and constructionism of the LOGO language. The early version of Scratch was built on Squeak Smalltalk.

While both Squeak etoys and Scratch are instance-based intuitive programming environment for creative and constructive learning, SOMETHINGIt has been built to aim at supporting programmers/performers to build live and sound prototypes.

Nielsen et al. [7] combined VDM specifications with Java code. Their approach as well as SOMETHINGIt use VDMJ as a VDM evaluation engine. While their implementation aims at graphical prototyping and integration between models and existing implementation, SOMETHINGIt is designed toward instance-oriented prototyping with the spirit of nurturing the dynamic nature of a Smalltalk environment.

**VIII. CONCLUDING REMARKS**

Live programming requires both improvisation and soundness of resulting programs, which industrial development often demands. SOMETHINGIt is a Smalltalk library to communicate with the REPL interpreters of different languages. SOMETHINGIt combines the both improvisation supported by dynamic and flexible Smalltalk, and soundness brought by static typing. SOMETHINGIt provides the five levels of interfaces that spans over the user interface level and the coding level.

**ACKNOWLEDGEMENTS**

The authors would like to thank Kazuhiko Yamamoto, Peter G. Larsen and Nick Battle for their valuable advices on technical elements implemented in SOMETHINGIt. We also thank Keijiro Araki, Yoichi Omori and Nobuto Matsubara for intensive discussions and comments.

**REFERENCES**