SOMETHINGit : A Prototyping Library for Live and Sound Improvisation

Tomohiro Oda[1] Kumiyo Nakakoji[1] Yasuhiro Yamamoto[2]
[1]Key Technology Laboratory
Software Research Associates, Inc.
Tokyo, Japan
[2]Precision and Intelligence Laboratory
Tokyo Institute of Technology
Yokohama, Japan
tomohiro@sra.co.jp 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 programmer or 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 *DoIt*, *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 1: A summary of comparison among Smalltalk, Haskell and VDM-SL

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

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.

```
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.

```
SIHaskell evaluate: 'take 2 [4, 4, 1]'
➔ 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.

```
'take' asHaskellExpression
       value: 2
       value:#(4 4 1) asOrderedCollection
➔ anOrderedCollection ( 4 4)

take := 'take' asHaskellExpression.
take2 := take value: 2.
take2 value: #(4 4 1) asOrderedCollection
➔ 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.

```
SIHaskell evaluate: '[4, 4, 1]'
➔ anOrderedCollection (4 4 1 )
```

Please note that the identity of Object which is mapped to SIObject in Haskell is kept, as demonstrated below.

Table 2: Mapping between Haskell Values and Smalltalk objects

Language	*Smalltalk*	*Haskell*
List	OrderedCollection	list
Set	Set	Data.Set
Map	Dictionary	Data.Map
Algebraic Data Type	SIAlgebraicData	*
Integer	Integer	Int
Decimal	Float	Fraction
Char	Character	Char
String	String	String
Smalltalk Object	Object	SIObject

```
| 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
```

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.

```
| hcode at |
hcode := SIHaskell prelude.
hcode program:
    'let at array index = messageSend array
"at:" [index] :: IO Int'.
at := 'at' asHaskellExpressionTyped: '[Int]-
>Int->IO Int' in: hcode.
at value: #(4 4 1) value: 3
➔ 1
```

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.

Table 3: SOMETHINGit Overview

Operating System	Linux or Mac OS X
Smalltalk System	Squeak 4.3 or highter Pharo 1.4 or highter
Required Smalltalk Packages	OSProcess
Haskell interpreter	GHCi 7.4.1 or higher
VDM-SL interpreter	VDMJ 2.0.1

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
    - SIExternalInterpreter
      - 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.

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

```
    | x `mod` 3 == 0 = Fizz
    | otherwise = Number x
```

The fizzbuzz function can be wrapped as a Smalltalk closure-like object by

```
| hcode |
hcode := SIHaskell source: '
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).

Fig 1. A GUI for FizzBuzz game

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

[1] Nakakoji, K., Interactivity, continuity, sketching, and experience (keynote abstract), in Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), pp. 621-621, 2011

[2] A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation. Addison-Wesley Longman Publishing Co., Inc., 1983

[3] Kay, A., A Personal Computer for Children of All Ages. In Proceedings of the ACM annual conference - Volume 1 (ACM '72), Vol. 1., 1972

[4] Yamamoto, Y., Nakakoji, K. Interaction Design of Tools for Fostering Creativity in the Early Stages of Information Design, International Journal of Human-Computer Studies (IJHCS), Special Issue on Creativity L. Candy, E. Edmonds (Eds.), Vol.63, No.4-5, pp.513-535, October, 2005.

[5] Nakakoji, K. and Yamamoto, Y., Conjectures on How Designers Interact with Representations in the Early Stages of Software Design, in Software Designers in Action: A Human-Centric Look at Design Work, M. Petre, A. van der Hoek (Eds.), Chapman & Hall 2013 (in print).

[6] Buxton, B., Sketching User Experiences – getting the design right and the right design, Morgan Kaufmann Publishers Inc., 2007

[7] Hudak, P., Hughes, J., Jones, S. P. and Wadler, P., A history of Haskell: being lazy with class. In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III), 2007

[8] Fitzgerald, J. and Larsen, P. G., Modelling Systems: Practical Tools and Techniques in Software Development. Cambridge University Press, 1998

[9] Agerholm, S. and Larsen, P. G., A Lightweight Approach to Formal Methods, in Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods, pp. 168-183, 1999

[10] Kay, A. Squeak etoys, children, and learning, http://www.squeakland.org/resources/articles.

[11] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y Kafai, Scratch: programming for all. Commun. ACM 52, 11 (November 2009), pp. 60-67, 2009

[12] Nielsen, C. B., Lausdahl, K. and Larsen, P. G, Combining VDM with executable code, in Proceedings of the Third international conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ'12), pp. 266-279, 2012