SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Bringing Scheme Programming to the iPhone — Experience

Engineer Bainomugisha[†], Jorge Vallejos,
Elisa Gonzalez Boix, Pascal Costanza,
Theo D'Hondt, and Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2 / B-1050 Brussels / Belgium

SUMMARY

The iPhone SDK provides a powerful platform for the development of applications that make use of iPhone capabilities such as sensors, GPS, Wi-Fi or Bluetooth connectivity. Thus far we observe that the development of iPhone applications is mostly restricted to using Objective-C. However, developing applications in plain Objective-C on the iPhone OS suffers from limitations such as the need for explicit memory management and lack of syntactic extension mechanism. Moreover, when developing distributed applications in Objective-C, programmers have to manually deal with distribution concerns such as service discovery, remote communication, and failure handling. In this paper, we discuss our experience on porting the Scheme programming language to the iPhone OS and how it can be used together with Objective-C to develop iPhone applications. To support the interaction between Scheme programs and the underlying iPhone APIs, we have implemented a language symbiosis layer that enables programmers to access the iPhone SDK libraries from Scheme. In addition, we have designed high-level distribution constructs to ease the development of distributed iPhone applications in an event-driven style. We validate and discuss these constructs with a series of examples including an iPod controller, a maps application and a distributed multiplayer Scrabble-like game. We discuss the lessons learned from this experience for other programming language ports to mobile platforms.

KEY WORDS: iPhone development; Scheme; Objective-C; language symbiosis; interactive scripting environment; event-driven programming

[*]Correspondence to: Engineer Bainomugisha
[†]E-mail: ebainomu@vub.ac.be
Contract/grant sponsor: Engineer Bainomugisha is funded by the SAFE-IS project in the context of the Research Foundation - Flanders (FWO); Elisa Gonzalez Boix is funded by the Prospective Research for Brussels program of IWOIB-IRSIB, Belgium; Jorge Vallejos is funded by the VariBru project of the ICT Impulse Programme of the ISRIB.

Received 09 January 2010
Revised 14 July 2010

1. INTRODUCTION

The iPhone SDK provides a powerful platform for the development of highly interactive applications that make use of iPhone's hardware capabilities such as sensors (accelerometer, proximity, and ambient light), GPS, Wi-Fi and Bluetooth connectivity. Typical example applications are those providing location-aware services [19], such as the AroundMe application [26] that enables users to locate businesses in the surroundings (e.g., movie theatres, supermarkets, or restaurants).

However, thus far we observe that to program native iPhone applications, one is forced to use mostly Objective-C [21]. While web scripting languages such as JavaScript are an alternative for developing iPhone applications, they have very limited access to the underlying native APIs, mainly because of security reasons. This restriction on the development language implies that programmers have to face the limitations of Objective-C, such as the need for explicit memory management [†], and the lack of syntactic extension mechanism. Moreover, when developing a distributed iPhone application, programmers have little more than a low-level socket API to work with, directly on top of the supported networking protocols (e.g., Bonjour). As a result, programmers have to manually deal with distribution issues such as service discovery, setting up sockets for remote communication, serialising invocations in order to perform remote method invocations, and network failure handling.

In this paper, we present iScheme [‡], our approach to port Scheme programming language to the iPhone platform. iScheme makes it possible to develop iPhone applications using Scheme and Objective-C. Integrating Scheme with Objective-C yields a language blend that boasts of the well-known Scheme benefits such as automatic garbage collection, structural macros, and higher-order functions, while enabling the reuse of existing Objective-C libraries (e.g., the Cocoa framework for GUI construction). Moreover, Scheme provides a good runtime with an interactive and dynamic scripting environment that enables rapid application development.

In the last couple of years, some Scheme implementations have been ported to mobile platforms, namely Gambit-C Scheme [11] and Moby Scheme [22] for the iPhone and Android platforms, respectively. Gambit-C has been used to develop realistic iPhone applications that are accessible in the iPhone App Store. Moby Scheme is mainly used for educational purposes and has been used by beginner students to develop interactive game applications and animations [12]. At a software engineering level, we observe that those ports take a compiler-based Foreign Function Interface (FFI) approach in which Scheme programs are compiled to C and Java source code for the iPhone and Android platforms, respectively. As a result, programmers do not need to deal with portability issues and can easily deploy their Scheme applications on mobile devices. Those ports also provide access to the phone APIs (e.g. GPS data). Typically, the native phone's APIs employ an event-driven programming style, as computation in mobile devices is mostly driven by all kinds of events coming from

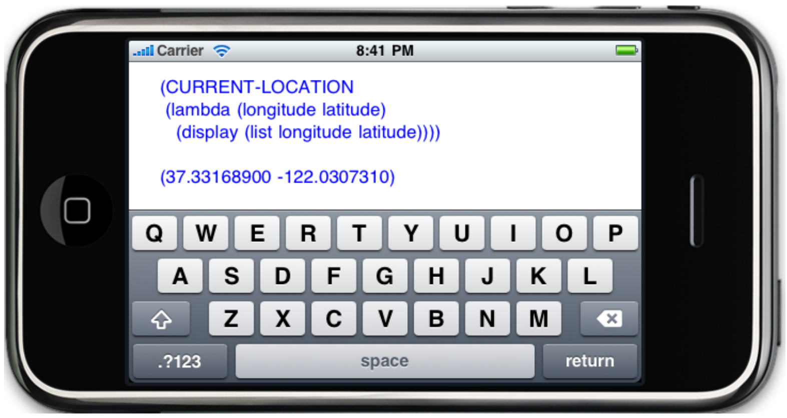
[†]While newer versions of Objective-C for the desktop platforms (Mac OS X v10.5 and later) support automatic memory management, there is no automatic garbage collection in the iPhone runtime system.
[‡]iScheme is based on a Scheme interpreter [9] that is developed at our laboratory. We chose to use a locally developed Scheme implementation primarily because of the availability of the source code as well as its author.

the environment (e.g., changes in the orientation or physical position of the phone, network events, user input, etc.). In this respect, Moby Scheme provides some support for event-driven programming which allows programs to react to a set of predefined events such as changes in the orientation of the phone. While there is some integration of event-driven programming on these existing ports, none of them provides constructs for distribution operating on top of the runtime platforms for distribution for Java or Objective-C. For instance, network disconnections need to be manually dealt with using a classic exception handling mechanism. This results in application code where exception handling is scattered all over the program. We note that disconnections cannot be ignored as in the case of desktop applications, since they are omnipresent in a mobile setting [23, 8].

In this work, we take a different approach to support Scheme and Objective-C interaction. Rather than using the FFI approach, iScheme employs Objective-C reflective API. This enables direct interaction between Scheme and Objective-C without need for writing wrapper functions in C. More concretely, we built a *language symbiosis* layer between Scheme and Objective-C that enables access to the Objective-C APIs from Scheme programs. *Language symbiosis* has been proposed in existing works as a means to enable two programs written in different languages to invoke each other's behaviour and exchange data [18]. With this symbiosis in place, iScheme provides developers with an event-driven programming model for accessing iPhone capabilities, with higher-order functions used as event handlers. In Objective-C, event-driven programs are typically organised around the notion of *delegates*, which serve as callbacks whose methods are invoked when a particular event occurs. Using higher-order functions as event handlers maps well onto such an event-driven architecture while keeping the simplicity of the Scheme programming model. From prior work at our lab, we have found that such an event-driven programming model is also suited for the development of distributed applications running on mobile devices [6]. As such, iScheme provides built-in constructs for service discovery (built on top of the Bonjour framework), asynchronous remote messaging (built on top of TCP/IP), and failure handling. This allows distribution concerns to be encapsulated in high-level constructs while relieving programmers of the difficulties engendered by distribution.

The contributions of this paper are: we port Scheme, which is a small but rich interpreted language, to the iPhone platform. We engineer a language symbiosis between Scheme and Objective-C by way of a reflective approach, which facilitates access to the iPhone APIs in an event-driven style. We design and implement event-driven distribution constructs specially tailored for mobile computing environments. In particular, our distribution constructs have built-in support for peer-to-peer service discovery, asynchronous remote messaging, and failure handling. We discuss our experience and summarise techniques of implementing language symbiosis with Objective-C. Our experience should serve as a stepping stone for Scheme programmers interested in developing applications for the iPhone, researchers that design languages for mobile devices as well docents teaching introductory programming courses. In addition, our experience may serve as a basis for other language ports to mobile devices employing runtime platforms with reflective capabilities, such as those found in the iPhone and the Android devices.

Availability: iScheme packaged as a scripting environment, is available for download at `http://soft.vub.ac.be/amop/ischeme`. The sample applications presented in this paper



```
(CURRENT-LOCATION
 (lambda (longitude latitude)
   (display (list longitude latitude))))

(37.33168900 -122.0307310)
```

Figure 1. Evaluating Scheme expressions on the iPhone. In this example, the `latitude` and `longitude` values are retrieved from the GPS API using the `CURRENT-LOCATION` construct.

have been tested on iPhone and iPod touch devices running iPhone OS 2.0 or later. So far we have used Apple's iPhone Development Certificate [3] to deploy applications to real devices. As of June 7th, 2010 the Apple iPhone Developer Program License Agreement [3] stipulates that applications may embed interpreters. This means that iPhone applications developed in iScheme can be submitted to the App Store [28].

2. SCHEME PROGRAMMING ON THE IPHONE

In this section we present the interactive iScheme environment with an example Scheme program running on the iPhone. This interactive environment is a simple Read-Eval-Print Loop (REPL) that enables one to load Scheme programs and directly execute them on the device. Such an environment coupled with the portability mobile devices provides a good platform to prototype new ideas of applications and interact with the iPhone APIs as well as scripting existing applications (e.g., SMS and phone).

To showcase iScheme on the iPhone device, we have developed a simple interactive environment, which provides basic functionality to input and evaluate Scheme code, and display the result. The editor embeds a Scheme interpreter and is deployed to the iPhone device like any other third party iPhone application [§]. For convenience, the interactive environment supports loading of Scheme files using the `load` function, which enables editing Scheme source code using any favourite Scheme editor, and afterwards uploading the Scheme files to the

[§]Deploying applications developed in iScheme to the iPhone does not require any modifications to the iPhone OS.

iPhone. For example, a Scheme program contained in a file `"GPS-location.scheme"` can be dynamically loaded as `(load "GPS-location")`.

To give a feel for how to program in iScheme while interacting with the iPhone APIs in an event-driven style, we consider an application that retrieves the GPS coordinates for the current device position. Figure 1 shows the complete implementation of such a program in Scheme and the resulting GPS longitude and latitude coordinates. The key part of this program is the `CURRENT-LOCATION` construct that takes as argument a closure (name of a function or an anonymous function constructed on the fly with the `lambda` keyword). This closure is invoked with the values of latitude and longitude from the GPS APIs. In this example, the latitude and longitude coordinates are composed into a list that is printed to the screen using the `display` function. This simple example already shows the benefits of integrating Scheme with Objective-C. Thanks to the higher-order functions, closures are passed around as event handlers instead of dealing with the delegate callbacks as is the case in plain Objective-C. In the following section, we explain the language symbiosis between Scheme and Objective-C.

3. BRIDGING SCHEME AND OBJECTIVE-C

In order to enable Scheme and Objective-C interaction, we have built a *language symbiosis* layer that is based on the *linguistic symbiosis* model [18]. The linguistic symbiosis model [¶] has been previously used to bridge two languages (e.g., SOUL and Smalltalk [17], AmbientTalk and Java [27]). It adheres to the following principles:

Data mapping which ensures that data from one language can be passed to another. For instance, when an Objective-C object crosses the boundary to Scheme needs to be represented as a Scheme value.

Protocol mapping which ensures that one language has a way to invoke another language's behaviour. For instance, Scheme programs require a mechanism to perform message sends to Objective-C objects and vice versa.

3.1. Linguistic Symbiosis between Scheme and Objective-C

Realising linguistic symbiosis between Scheme and Objective-C is not trivial because of the differences in the programming paradigms. Scheme is based on the functional programming paradigm where operations are performed by function applications, whereas Objective-C is based on the object-oriented programming paradigm where operations are performed by sending messages to objects. As it is not possible to make these differences completely seamless, we therefore provide ways to perform operations from one language to the other. The remainder of this section explains how we achieve language symbiosis between Scheme and Objective-C.

[¶]FFIs that provide a two-way language interaction also qualify as language symbiosis.

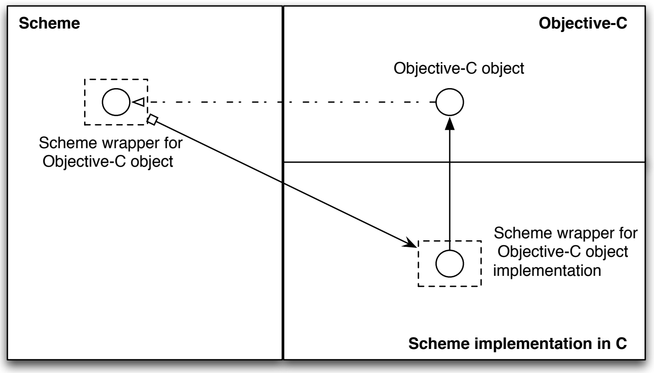

Scheme
Objective-C
Objective-C object
Scheme wrapper for Objective-C object
Scheme wrapper for Objective-C object implementation
Scheme implementation in C

Figure 2. Linguistic symbiosis between Scheme and Objective-C

3.1.1. Data mapping between Scheme and Objective-C

Figure 2 illustrates the representation of an Objective-C object in Scheme. When an Objective-C object crosses the boundary to Scheme, it is wrapped as a Scheme value and therefore can be bound to a regular Scheme variable or passed around as an argument to a Scheme function. The Scheme interpreter we are using is implemented in C and thus the host language for the Scheme values. An Objective-C object is wrapped as a generic Scheme value (`OBJC_TYPE`) that points to the actual Objective-C instance. Objective-C and C are inherently symbiotic hence no changes are required to host Objective-C objects in C.

When a Scheme value crosses the boundary to Objective-C, it is automatically converted to the corresponding Objective-C type, and it can be bound to an Objective-C variable or passed as argument to an Objective-C method. We define type conversion functions for converting Objective-C values to Scheme values and vice versa. We further explain the type conversion functions in Section 3.2.4.

3.1.2. Protocol mapping between Scheme and Objective-C

The difference in the programming paradigms requires constructs to enable Scheme and Objective-C to invoke each other's behaviour. In Scheme we provide `OBJC-CLASS`, `OBJC-INSTANCE`, and `OBJC-SEND` constructs to load Objective-C classes, create objects, and send messages, respectively. On the other hand, we provide `SCHEME_CALL` construct to invoke Scheme functions from within Objective-C. We explain these symbiosis constructs using examples in the remainder of this section.

3.2. Scheme/Objective-C Symbiosis by Example

3.2.1. Instantiating and sending messages to Objective-C objects from Scheme

Scheme programs can instantiate Objective-C objects using the `OBJC-INSTANCE` construct, which performs object allocation and initialisation. In addition, we provide the `OBJC-SEND` construct, which performs message sends to an Objective-C instance. We illustrate these constructs by means of an example of a Scheme function that converts a string to audio using the Objective-C class `NSSpeechSynthesizer`. This implements methods for parsing text and generating synthesised speech, as follows:

```
(define (speak-out-loud text)
  (let ((synthesizer (OBJC-INSTANCE NSSpeechSynthesizer)))
    (OBJC-SEND synthesizer startSpeakingString: text)))
```

The above code snippet shows the definition of the `speak-out-loud` function, which implements the behaviour of converting text to audio. The `OBJC-INSTANCE` construct takes as argument a name of an Objective-C class and returns a new instance of the class. The `OBJC-SEND` construct takes as arguments an Objective-C class or instance, the method name, and arguments to the method to be invoked. The `(OBJC-INSTANCE NSSpeechSynthesizer)` expression creates an instance of the `NSSpeechSynthesizer` class and returns a reference to its instance that is then bound to the Scheme variable `synthesizer`. The expression `(OBJC-SEND synthesizer startSpeakingString: text)` invokes the `startSpeakingString:` method on the Objective-C instance stored in the variable `synthesizer` with `text` (the text to speak out) as the argument. The return value of `OBJC-SEND` is a return value of the Objective-C method being invoked and is wrapped as `OBJC_TYPE` value in Scheme. We explain in Section 3.2.4 the type conversion functions to convert an Objective-C object to the appropriate Scheme value (e.g., Objective-C's `NSString` to a Scheme string).

3.2.2. Loading Objective-C classes from Scheme

The symbiosis layer provides the `OBJC-CLASS` construct which takes as argument an Objective-C class, and returns a reference to its class object. The following example illustrates the use of the `OBJC-CLASS` construct.

```
(define NSSynthesizer (OBJC-CLASS NSSpeechSynthesizer))
```

In the above code snippet, the expression `(OBJC-CLASS NSSpeechSynthesizer)` returns a reference to the `NSSpeechSynthesizer` class that is bound to the Scheme variable `NSSynthesizer`. In case the class name does not exist, then the return value is a reference to `nil` (i.e. a null instance) [‖]. The reference to the class object can be used to invoke class methods. Note that it is also possible to perform object instantiation by sending the `alloc` and

[‖]In Objective-C, sending a message to `nil` has no runtime effect `init` messages to the Objective-C class object, but we provide the `OBJC-INSTANCE` construct to this end.

3.2.3. Invoking Scheme functions from Objective-C

Objective-C programs can call Scheme functions with the `SCHEME_CALL` construct, which takes a Scheme function name and a variable number of optional arguments.

```
SCHEME_CALL(function-name, args)
```

The function name can be any globally defined Scheme function or a closure that is passed as an argument to an Objective-C method.

3.2.4. Type conversions

When Scheme values are passed to Objective-C methods, implicit conversion is performed to appropriate types in Objective-C (e.g., number to `NSNumber` and `string` to `NSString`). In addition, we provide functions to perform explicit conversions of Scheme values to Objective-C values. For example, the `string->NSString` function converts a Scheme string to an Objective-C `NSString`. The `number->NSNumber` function converts a Scheme number to an Objective-C `NSNumber`. The `list->NSMArray` function converts a Scheme list to an Objective-C `NSMutableArray`.

By default, return values from Objective-C methods are wrapped as a generic `OBJC_TYPE` Scheme type, which is a Scheme value representation of Objective-C objects in the Scheme interpreter. Objective-C objects are not automatically converted to Scheme values whenever they cross from Objective-C to Scheme world. This is mainly because: the identity of an Objective-C object may be lost when they are converted back and forth, a single Objective-C type may correspond to many Scheme values, and the conversion overhead involved (we discuss the preliminary benchmarks on method call overhead Section 7.2). We instead provide type conversion functions for converting Objective-C values to their Scheme counterparts. For example, `NSString->string` function converts an Objective-C `NSString` to a Scheme string. The `NSNumber->number` function converts an Objective-C `NSNumber` to a Scheme number. Explicit type conversion implies that the programmer decides to convert Objective-C objects to Scheme values when it is necessary. For example, an Objective-C instance that is passed to Scheme to be later passed as argument to an Objective-C method, does not require a conversion. Not only does this imply that unnecessary conversions may be avoided, but also the object identity of the Objective-C instance is preserved.

Let us illustrate the conversion functions with an example of retrieving the iPhone's configuration settings (e.g., device name, model, and iPhone OS version). The *UIKit* framework provides the `UIDevice` class with methods to access the device's information such as the name, the device model, and the operating system name. We implement this example in Scheme using the symbiosis constructs as follows:

```
; This is an example retreiving the iPhone device
; configuration settings from Scheme
```

```
(define (show-my-iphone-details)
  (let* ((UIDevice (OBJC-CLASS UIDevice))
         (device (OBJC-SEND UIDevice currentDevice))
         (device-name (OBJC-SEND device name))
         (name-string (NSString->string device-name))
         (device-model (OBJC-SEND device model))
         (model-string (NSString->string device-model))
         (system-name (OBJC-SEND device systemName))
         (system-string (NSString->string system-name))
         (device-details (string-append
                          "device name: " name-string
                          "model: " model-string
                          "system: " system-string)))
    (display device-details)))
```

The above code snippet shows the definition of the `show-my-iphone-details` function that makes use of the type conversion function `NSString->string`. The `(OBJC-CLASS UIDevice)` expression returns a reference to the `UIDevice` class that is then bound to a Scheme variable `UIDevice`. The `(OBJ-SEND UIDevice currentDevice)` expression invokes the class method `currentDevice` on the class `UIDevice` and returns the instance representing the current device. We first send the messages `name, model`, and `systemName` to retrieve the Objective-C's `NSString` objects for the device name, model, and the operating system, respectively. We then use the function `NSString->string` to convert each of the device details to a Scheme string. Evaluating the the expression `(show-my-iphone-details)` displays the iPhone configuration settings as follows:

```
(show-my-iphone-details)
=> device name: Engineer-iPhone model: iPhone system: iPhone OS
```

3.3. Scripting the Native iPhone Applications Using Scheme

One of the benefits that Scheme brings to the iPhone is the ability to create Scheme scripts that dynamically interact with the native iPhone applications (such as the iPod, and the address book). The ability to interact with the native applications from Scheme opens the way for interesting applications. For instance, one can easily develop a variation of the native iPod application enriched with location information (e.g., to stop playing music when a user walks into a meeting room). In the remainder of this section, we describe an example Scheme application that interacts with the iPod application.

Objective-C provides the *Media Player* framework that enables access to the iPod library and methods to play movies, music, audio podcasts, and audio books. This framework enables developers to build applications that make use of the iPhone's media facilities. However, developing such an application in plain Objective-C requires one to go through complexities of GUI programming. Using the symbiosis constructs in Scheme, one can build such applications on the fly. The implementation of such an application in Scheme is shown below.

```
(define (ipod-controller)
  (let* ((Controller (OBJC-CLASS MPMusicPlayerController))
         (Query (OBJC-CLASS MPMediaQuery))
         (query (OBJC-SEND Query songsQuery))
         (musicPlayer (OBJC-SEND Controller iPodMusicPlayer)))
    (OBJC-SEND musicPlayer setQueueWithQuery: query)
    (lambda (action)
      (case action
        ((play)  (OBJC-SEND musicPlayer play))
        ((stop)  (OBJC-SEND musicPlayer stop))
        ((pause) (OBJC-SEND musicPlayer pause))
        ((play-next) (OBJC-SEND musicPlayer skipToNextItem))))))
```

The `ipod-controller` function reifies the behaviour of the iPhone's built-in iPod application. The expression `(OBJC-CLASS MPMusicPlayerController)` loads the `MPMusicPlayerController` Objective-C class and binds it to the `Controller` variable. The `MPMusicPlayerController` class implements methods for retrieving the instance of the media player.

The `(OBJC-CLASS MPMediaQuery)` expression loads the `MPMediaQuery` class and binds it to the variable `Query`. The `MPMediaQuery` class implements methods for constructing media query types (such as albums, artists, or songs). In this example, we create a media query of the music items grouped and sorted by the song name, by invoking the method `songsQuery` on the `MPMediaQuery` class. Invoking the method `iPodMusicPlayer` on `MPMusicPlayerController` returns the reference to the device's iPod music player instance that is bound to the `musicPlayer` variable. Next, we set the playback queue by invoking the method `setQueueWithQuery:` on the music player with the query type. In this example, the playback queue contains all songs. In addition, the music player provides methods `play, pause, skipToNextItem` to control the playback queue.

The `ipod-controller` function returns a dispatcher function [**] that takes one argument and performs the corresponding action (play, pause, stop, skip to next item) depending on the specified argument. For example, the following scripts can be evaluated on the iPhone to play and forward to next media items:

```
;; example usage
(define my-ipod-controller (ipod-controller))

; to start playing
(my-ipod-controller 'play)

; to play next song
```

[**]iScheme provides a small prototype-based object system, which eliminates the need to write a dispatcher function. For didactical reasons we write all examples in a functional style.

```
(my-ipod-controller 'play-next)
```

3.4. Summary

In this section we have introduced and explained the Scheme and Objective-C symbiosis that is based on four language constructs: `OBJC-CLASS`, `OBJC-INSTANCE`, `OBJC-SEND` and `SCHEME_CALL`. We have demonstrated with examples how to create and send messages to Objective-C instances from Scheme. We discussed the type conversion functions that make it possible to convert Objective-C values to Scheme values and vice versa. We subsequently presented the iPod controller example that demonstrates how Scheme programs interact with the native iPhone applications using the symbiosis constructs. In the next section, we demonstrate constructs built on top of the symbiosis layer, which ease the development of iPhone applications.

4. EVENT-DRIVEN PROGRAMMING FOR THE IPHONE DEVELOPMENT

As introduced in Section 1, Scheme's support for higher-order functions and closures, maps well onto the event-driven programming model employed in the iPhone APIs. In this section we present an event-driven programming model for accessing iPhone capabilities and programming distributed applications on the iPhone. First, we explain constructs for accessing the iPhone capabilities (e.g., GPS location information) in an event-driven style. Secondly, we explain the distribution constructs to discover software services, communicate, and deal with network failures by means of events.

4.1. Constructs for Accessing iPhone Capabilities

4.1.1. GPS location information

The *Core location* framework in the iPhone SDK, makes use of events to deliver the current GPS coordinates to the event handlers. As shown in Section 2 iScheme provides the `CURRENT-LOCATION` construct, which is built on top of the Core location framework using our symbiosis constructs. It takes a Scheme closure as its argument and invokes it whenever the new location information becomes available from the underlying GPS APIs. The following code shows the full implementation of the `CURRENT-LOCATION` construct.

```
(define-macro CURRENT-LOCATION
  (lambda (event-handler)
    `(let ((Ulocation (OBJC-INSTANCE Ulocation)))
       (OBJC-SEND Ulocation currentLocation: ,event-handler))))
```

`CURRENT-LOCATION` is defined as a macro that accepts as argument a two-parameter Scheme closure (the event handler). `Ulocation` is an Objective-C class that implements a method `currentLocation:` which takes as argument a Scheme closure that is invoked with the latitude and longitude coordinates as soon as they become available from the iPhone's GPS receiver. For example, evaluating an expression `(CURRENT-LOCATION list)` returns a list of the GPS coordinates as explained in Section 2.

4.1.2. Interacting with the native iPhone applications

The iPod controller example presented in Section 3.3 already shows how to interact with the native iPhone applications using our symbiosis construct. However, writing Scheme programs using the symbiosis constructs (such as `OBJC-SEND`) makes the programmer think in terms of Objective-C message sends. To alleviate this problem, we have implemented a generic `openApp` construct, that enables interaction with the native applications (e.g., phone, SMS, mail, maps) in a pure Scheme programming style. The following code shows the definition of the `make-call` and `send-sms` constructs that enable access to the phone and SMS features.

```
;; example  (make-call "026291241")
(define (make-call to)
  (openApp phone to))

;; example (send-sms "026291241")
(define (send-sms to)
  (openApp sms to))
```

In the above example, the `make-call` and `send-sms` constructs take a string argument representing the phone number and starts the phone call or launches the SMS application with the specified recipient. For example, evaluating the expression `(make-call "026291241")` starts a phone call to the phone number `"026291241"`. Similarly, evaluating `(send-sms "026291241")` launches the SMS application with the specified recipient.

The `openApp` macro takes as argument the application name and optional arguments required by the application (e.g., in the above example the phone number for the phone or SMS applications). It is built using the symbiosis constructs as follows:

```
(define-macro openApp
  (lambda (app . args)
    `(let* ((url-str (create-url ',app ,@args))
            (UIApp (OBJC-CLASS UIApplication))
            (appInstance (OBJC-SEND UIApp sharedApplication))
            (NSURL (OBJC-CLASS NSURL))
            (url (OBJC-SEND NSURL URLWithString: url-str)))

       ;; launch the application
       (if url-str
           (OBJC-SEND appInstance openURL: url)
           (display "iphone application not supported!")))))
```

The `openApp` construct is implemented as a macro using the `define-macro` form. The function `create-url` returns an application URL string given the application name and a list of the required arguments. The `UIApplication` class provides methods to control the current running applications on the iPhone OS. The `sharedApplication` method returns the `UIApplication` instance associated with the current running application that is then bound to `appInstance`. The `URLWithString:` method creates `NSURL` instance from the `url-str` string. Evaluating the expression `((OBJC-SEND appInstance openURL: url))` invokes the method `openURL` on the `UIApp` instances and launches the specified application.

In Section 5.1 we present a concrete maps application that illustrates the usage of the `CURRENT-LOCATION` and the `openApp` constructs.

4.2. Constructs for Distributed Programming

As previously explained, we have built high-level constructs on top of the symbiosis layer that alleviate the difficulties of distribution. More concretely, iScheme provides a reactive event loop distributed model that is based on the AmbientTalk language event loop model [6]. We have built remote communication around the concept of asynchronous message passing in order to abstract over network failures without blocking the control flow. In order to deal with the fact that services may need to be discovered in the environment as the user moves about without relying on predefined infrastructure, iScheme has a built-in publish/subscribe engine to allow applications discover services in a peer-to-peer manner. Next, we explain the different constructs iScheme provides to programmers for distributed programming by means of a simple news service application. In this application, news editors can submit articles to news publishers while moving about. Then the news publisher can broadcast news items, which are printed on the screen of iPhones of nearby potential customers that have announced their interest in the current news.

4.2.1. Exporting Functions as Services

Distributed computation in iScheme is expressed in terms of functions. A function represents a certain service offered by a device. A device can acquire a remote reference to a function owned by a remote device, and then interact with it by performing remote function invocations. As fixed name servers may not be available when two iPhones come in communication range and set up a collaboration, iScheme identifies exported functions by means of service types. Service types are a lightweight classification mechanism used to categorise functions explicitly by means of a nominal type.

In the example of the news service application, the news publishers need to make available their publishing service to other devices. The code snippet below shows how a programmer can explicitly export the function representing the news publisher service.

```
(define news-service (service-type iPhone-news))
(export-service news-publisher news-service)
```

A service type is defined using the `service-type` construct. In the above code snippet, the variable `news-service` stores the service type `iPhone-news`. The `export-service` construct publishes onto the network a given function as the given service type. From the moment a function is exported, it is discoverable by functions residing in other devices by means of its associated service type. In this example, the `news-publisher` function is exported on the network as a `iPhone-news` service. The `export-service` construct returns a closure that can be used to take the function offline by invoking the `cancel-publication` construct.

4.2.2. Service Discovery

iScheme employs a publish/subscribe service discovery protocol. A publication corresponds to exporting a function by means of a service type (which serves as a “topic” known by both the publisher and the subscriber [10]). A subscription corresponds to registering an event handler on a service type, which is triggered whenever a function exported under that type is encountered in the network. In the news service application, an editor can be notified whenever a news publisher is discovered as follows:

```
 (when-discovered news-service
       (lambda (publisher-ref)
           (submit-news publisher-ref)))
```

The `when-discovered` construct takes as arguments the service type to search for and a one-parameter closure that serves as an event handler. Such a closure is invoked with a remote reference to the newly discovered remote function associated with that service type. In the above code snippet, whenever a `iPhone-news` service is discovered, the `submit-news` function is invoked, passing along the parameter `publisher-ref` remote reference received. Similar to the `export-service` construct, the `when-discovered` construct returns a closure that can be used to cancel the subscription, by invoking `cancel-subscription` construct.

4.2.3. Asynchronous Remote Function Invocation

Once a reference to the remote function is obtained, remote function invocations can be performed by means of the `remote-send!` construct as follows:

```
(define (submit-news publisher-ref)
  (for-each
   (lambda (article)
     (remote-send! publisher-ref receive-article article))
   list-of-articles))
```

The `remote-send!` construct takes as arguments a remote reference, a function name, and optional variable number of arguments. Arguments specified in a remote function invocation are parameter passed by copy. Currently only a subset of the Scheme first-class values (Booleans, Numbers, Characters, Symbols, Strings, Pairs and Lists[††]) can be parameter

[††]Pairs and Lists that contain circular references are not supported in our current serialisation mechanism.

passed in a remote method invocation. In this example, the `submit-news` function iterates over a list storing news articles to be published, and invokes the `receive-article` function on the `publish-ref` reference corresponding to the newly discovered news publisher. The `remote-send!` construct performs a non-blocking asynchronous remote function call. This means that `remote-send!` enqueues a remote function call in the Scheme interpreter and it immediately returns `nil`. As such, callers do not wait for the remote function call to be remotely performed nor for the return value of such computation.

In order to get the return value of a remote invocation, we provide the `when-resolved` construct which registers an event handler that is invoked when the return value of the remote function invocation becomes available. In our running example, this is used to acknowledge the reception of articles sent to the news publisher.

```
(define (submit-news publisher-ref)
  .... ;;iterator over each news
  (when-resolved
   (remote-send publisher-ref receive-article article)
   (lambda (receipt)
     (set! receipts (cons receipt receipts)))
   (catch
       (lambda (exception)
         ;;exception handling code
         )))
  ...)
```

The `remote-send` construct works similar to the `remote-send!` construct but it returns a `future` instead. A future is a placeholder for the return value that will be computed asynchronously. Once the return value is computed, it replaces the future object, and the future is then said to be resolved with the value. Note that registering a future on a `remote-send` construct does not block the caller of the remote function call. It is possible to register a block of code which is triggered when the future becomes resolved by means of the `when-resolved` construct. The `when-resolved` construct takes a future and two closures and registers an event handler on that future. If the future is resolved to a value, the first closure is invoked, passing along the return value of the remote computation. In this example, the `receipt` value is received as the return value of the `receive-article` remote function call. If the remote function invocation raises an exception, the corresponding future is said to be ruined with the exception and the `catch` closure is applied to the exception. This enables applications to catch asynchronously raised exceptions and apply some correcting actions in a way similar to the well-known `try-catch` construct.

As explained before, performing a remote function call using `remote-send` enqueues a remote function invocation that is enqueued in the Scheme interpreter. When a network failure of the device hosting a function that needs to be remotely invoked occurs, the remote method invocation is still stored in the caller side. When the network partition is restored at a later point in time, the accumulated function invocations are transparently flushed to the remote device in the same order as they were originally performed. However, sometimes disconnections may take unexpectedly longer or devices may not move back into communication range again. To deal with these long-lasting disconnections, programmers can attach a timeout to the remote function invocation to limit the time to wait for the reception of the return value. Such a timeout can be attached by means of the `due-in` construct in the `when-resolved` construct as follows:

```
(define (submit-news publisher-ref)
  .... ;; iterator over each news
  (when-resolved
   (remote-send publisher-ref receive-article article)
   (lambda (receipt)
     (set! receipts (cons receipt receipts)))
   (due-in 20.0)
   (catch
       (lambda (exception)
         ;;exception handling code
         )))
  ...)
```

The `due-in` construct expects as parameter a number denoting a timeout in seconds. If the return value is not received within the timeout specified, the future is automatically ruined and the `TimeoutException` is raised which can be handled with the `catch` construct as explained before.

In Section 5.2, we present a concrete distributed application developed using these constructs.

4.3. Summary

In this section we have presented an event-driven programming model in Scheme that eases the development of interactive and distributed iPhone applications. The constructs for accessing the iPhone capabilities (the GPS, phone, and SMS) further hide away the symbiosis constructs. This yields the advantage that Scheme programmers do not have to think in terms of Objective-C programming. We have built non-trivial distribution constructs for service discovery, asynchronous remote messaging, and network failure handling. We further illustrate these constructs using concrete examples in the next section.

5. APPLICATIONS

In this section we present two concrete iPhone applications that we develop using the constructs (explained in Section 4). The first example is a maps application that demonstrates writing an iPhone application in Scheme while making use of the iPhone GPS capabilities. The second example is a distributed peer-to-peer digital SCRABBLE®-like game that demonstrates the use of distributed programming constructs in iScheme.

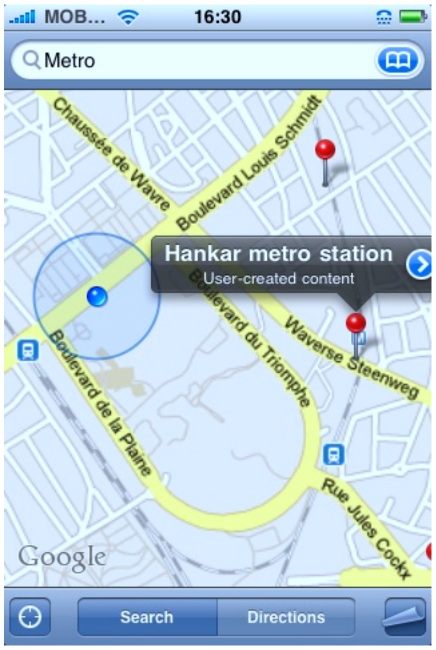


Figure 3. `(show-map "Metro")` displays the Google map annotated with current location and nearby metro stations.

5.1. Building a maps application

We built a maps application that combines a Google map and places of interest in the neighbourhood. The application works as follows: (1) Using the iPhone GPS, the application retrieves the current user's location. (2) Using the user specified query of the places of interest (e.g., hotels, transport stops, hospitals), the application displays a Google map [16] annotated with the nearby places and the current user location. Figure 3 shows the screen shot of the resulting maps application running on the iPhone device.

Below we present the implementation of the maps application in Scheme:

```
(define (show-map query)
  (CURRENT-LOCATION
   (lambda (latitude longitude)
     (google-maps latitude longitude query))))
```

The above code shows a complete definition of the `show-map` function that implements the behaviour of the maps application. The `show-map` function takes as argument a user query (e.g., “Metro”, “Hotel”). The `CURRENT-LOCATION` construct as explained in Section 4.1 is used to retrieve the latitude and longitude coordinates. The argument to the `CURRENT-LOCATION` construct is a closure that takes two arguments that is invoked with the latitude and longitude coordinates.

The `google-maps` function in `show-map`, given the latitude, longitude and a user query, displays a Google map annotated with the places matching the user query and the current location. The `google-maps` function is implemented using the `openApp` construct (explained in Section 4.1) as follows:

```
(define (google-maps lat longi search-query)
  (openApp maps lat longi search-query))
```

The first argument to the `openApp` construct is the application name (in this example, `maps`). The second and third arguments are the longitude and latitude coordinates while the fourth argument is the search query. The above maps application code can be loaded and evaluated directly on the device. For example, evaluating `(show-map "Metro")` shows the Google map annotated with metro stations in the neighbourhood as shown in Figure 3.

5.2. AmbiScrabble: Building a distributed peer-to-peer game application

To further illustrate the distributed programming constructs integrated in iScheme, we now present a peer-to-peer game for the iPhone called *AmbiScrabble*. AmbiScrabble is a digital version of a Scrabble-like game where players work collaboratively with their iPhones to form words. A demonstration of the game can be found on the iScheme website[‡‡]. Figure 4 shows the screen shot of the AmbiScrabble application on the iPhone. Players are organised in teams and each player has a rack of letters. Letters are consumed by forming valid English words. The team that first consumes all its letters wins. Players belonging to the same team can exchange letters among themselves.

The AmbiScrabble game has been designed in a peer-to-peer fashion without assuming a centralised server to coordinate the game. It is also fault-tolerant such that player failures do not hamper the game progress. These design choices are primarily motivated by the fact that the game runs on iPhones equipped with wireless technology. Connectivity using such a technology is often characterised by frequent network disconnections either as because of limited connectivity or users may continuously move about.

We implement the game logic and distribution concerns of the AmbiScrabble application in iScheme, while the graphical user interface (GUI) is implemented in Objective-C using the Cocoa framework. Before describing how our constructs are used to implement GUI interactions between Objective-C and Scheme and the distribution concerns of the game, we give an overview of the game implementation. The following code snippet summarises the relevant parts of the AmbiScrabble application in iScheme.

```
(define (create-ambiScrabble-game)
  (let ((GUI-proxy (setup-gui))
```

[‡‡]http://soft.vub.ac.be/amop/ischeme/example_applications

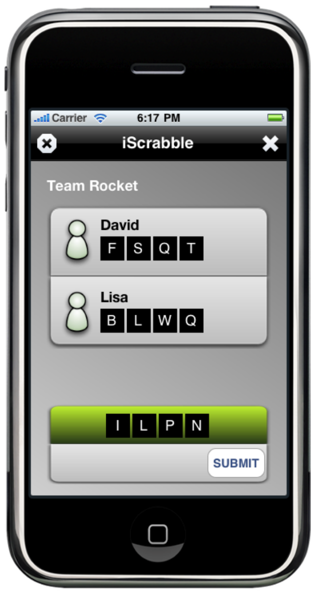


Figure 4. The screen shot of the AmbiScrabble game.

```
    (team-controller (make-team-controller))
    (rack-controller (make-rack-controller)))

; the player's local interface functions
(define (add-letter-to-rack letter)
  (rack-controller 'add-letter letter)
  (notify-team))

(define (add-new-player player)
  (team-controller 'add-new-player player))

(define (get-player-name) ...)
(define (get-player-team) ...)

(define (initialise-game team-name player-name)
  (initialise-player-info team-name player-name)
  ; engage in peer-to-peer discovery of other players
  (go-online)
  (discover-other-players))
```

```
; the player's remote interface functions
(define (get-player-info)
  (let ((name (get-player-name))
        (team (get-player-team)))
    (list name team)))

(define (request-letter player-name letter)
  (process-request player-name letter))

(define (receive-letter from letters status)
  (cond ((= status approved)
         (add-letter-to-rack letters)
         (notify-team)
         (alert-request-approved from letters))
        ((= status refused)
         (alert-request-refused from letters))
        ....)
  'done)

(define (remote-interface)
  (lambda (message . args)
    (case message
      ((get-player-info) (get-player-info))
      ((request-letter)  (apply request-letter args))
      ((receive-letter)  (apply receive-letter args)))))

'started))
```

The `create-ambiScrabble-game` function consists of: the `GUI-proxy` that manages the game's GUI, the `team-controller` that manages the players and teams in the game, and the `rack-controller` that manages the player's rack and the word formation. The application defines local and remote interfaces which consist of a set of functions to interact with the GUI and with the remote players, respectively. For example, the local interface contains the `add-new-player` function that implements the behaviour for adding a player to the appropriate team and the `add-letter-to-rack` function that implements the behaviour of adding a letter to the player's rack of letters. The remote interface contains the `get-player-info` function that is used for obtaining the remote player's name and team, the `request-letter` function that is used for requesting a letter from a remote player belonging to the same team, and the `receive-letter` function that is used to "throw" letters to the nearby players. The `status` argument in the `receive-letter` function indicates whether the request was accepted or not. All the remote interface functions are wrapped in the `remote-interface` closure that dispatches a remote function invocation to the appropriate function.

5.2.1. GUI Interactions

When the `create-ambiScrabble-game` function is invoked, the game GUI is launched by invoking the `setup-gui` function. The return value of `setup-gui` function is a closure that embodies all the behaviour of Objective-C and Scheme interaction. The resulting closure is bound to the variable `GUI-proxy`. We show part of the implementation of the `setup-gui` function below.

```
; manages all the interactions between Objective-C and Scheme
(define (setup-gui)
  ...
  ; Create new instance of gameViewController
  (let ((viewController (OBJC-INSTANCE gameViewController)))

    ; Show new player
    (define (display-new-player name rack)
      (let* ((info (list name rack))
             (array (list->NSMArray info)))
        (OBJC-SEND viewController addPlayer: array)))

    ; Remove player from the screen
    (define (remove-player name)
      (OBJC-SEND viewController removePlayer: name))

    ; Set callback for when letter is selected in word by the user
    ; Letters should be moved back to the rack.
    (define (set-onformword-callback proc)
      (OBJC-SEND viewController setWordCallback: proc))
    ...))
```

The game's GUI implementation in Objective-C contains the class `gameViewController` that implements methods for capturing user input, and updating the GUI whenever the game data in Scheme changes. In the above code snippet, the variable `viewController` holds a reference to the instance of the `gameViewController` class that is created using `(OBJC-INSTANCE gameViewController)`. The `display-new-player` function displays a new player, and a rack of letters on the GUI by invoking the `addPlayer:` method of the `gameViewController` class. The `remove-player` function implements the behaviour of removing a player from the GUI. Functions that need to be called when a user performs certain actions on the GUI (e.g., a pinch on the submit button to form a word) are registered as callbacks to Objective-C methods. For example, the `set-onformword` function registers a Scheme function that is called whenever the user presses the button to form word.

5.2.2. *Distributed Interactions*

When the GUI is launched, the player is prompted to enter a name and a team. This information is used to initialise the game and the player data by invoking `initialise-game` function from Objective-C. The next step of the game setup is to publish the game instance onto the network and to search for other players in the surroundings by invoking the functions `go-online` and `discover-other-players`, respectively. Note that we assume that there is no dedicated centralised server for game coordination and as such, each game instance publishes and subscribes itself to the network. The following code snippet shows the implementation of the `go-online` function that publishes the game.

```
; the service type
(define ambiScrabbleService (service-type ambiSrabble))

; publishing the game instance on the network.
(define (go-online)
  (export-service remote-interface ambiScrabbleService))
```

A game instance is published onto the network with the `ambiScrabble` service type (stored in the `ambiScrabbleService` variable). More concretely, the `go-online` function publishes the `remote-interface` closure as a `ambiSrabble` service using the `export-service` construct.

The `discover-other-players` function uses the `when-discovered` construct to register a subscription to discover other players in the network as follows:

```
; discovering other players in the surroundings
(define (discover-other-players)
  (when-discovered ambiScrabbleService
       (lambda (remote-player)
         (add-new-player remote-player))))
```

Whenever a new `ambiScrabble` service type is discovered, the `add-new-player` function is applied receiving by parameter the newly established reference to the `remote-player` function of the remote player. This function performs the necessary remote function invocation to obtain the remote player's information using the `remote-send` and `when-resolved` constructs as follows:

```
(define (add-new-player remote-player)
  (when-resolved
   (remote-send remote-player get-player-info)
   (lambda (info)
     (let ((name (list-ref info 0))
           (team-name (list-ref info 1)))
       (if (team-exists? team)
           (if (not (player-exists? name))
               (add-player remote-player name team-name))
```

```
          (begin
            (create-new-team team-name)
            (add-player remote-player name team-name)))))))
```

The `add-new-player` function performs a remote invocation of the function `get-player-info` to retrieve the player's name and team, using the `remote-send` construct. The return value of the remote invocation is received by parameter in the `when-resolved` lambda, which checks whether the team and the player already exist in the data structures of the application (i.e. if the player was previously discovered). If the team already exists but the player does not, then the player is added to that team. Otherwise, the player is added to the newly created team.

In order to deal with network disconnections of players, the game makes use of the `due-in` construct in the `when-resolved` construct to notify the disconnection of a player. Players whose disconnection exceeds a certain period of time (e.g., 20 seconds) are greyed out in the GUI as shown below:

```
(when-resolved
 (remote-send remote-player get-player-info)
 (lambda (info) ... )
 (due-in 20.0)
 (catch
     (lambda (exception)
       (grey-out-player remote-player))))
```

5.3. Evaluation

The example applications described above demonstrate that the language symbiosis between Scheme and Objective-C yields a number of benefits:

iPhone capabilities are accessible from Scheme. The Scheme and Objective-C symbiosis provides constructs to enable Scheme applications access to the Objective-C frameworks. For example, the Core location framework (for GPS information) and the Bonjour framework (for service publication and discovery) are now accessible using the symbiosis constructs. The maps example shows how to easily obtain GPS coordinates using Scheme with the `CURRENT-LOCATION` construct. The AmbiScrabble application illustrates the use of the `export-service` construct which is built on top of the Bonjour framework that makes it possible to publish Scheme functions as services over the iPhone's Wi-Fi connectivity.

Using Scheme macros to build language constructs. The Scheme macro system provides support for building syntactic extension constructs. This means that new language constructs can be created on top of the symbiosis constructs to ease the development of iPhone applications. For instance, the maps example makes use of the `openApp` construct that enables interaction with the native iPhone applications. The distribution constructs demonstrated in the AmbiScrabble application enables the programmer to write distributed iPhone applications without dealing with the low-level details of distributed programming as would be the case in plain Objective-C.

Using Scheme higher-order functions as event-handlers. The above two examples have demonstrated the use of the constructs that ease the development of event-driven programs. Scheme's higher-order function features and closures make it suitable for the event-driven programming model upon which the iPhone APIs are based. We have shown the use of Scheme closures as event-handlers by building non-trivial constructs for accessing iPhone capabilities and distributed programming. For instance, the `CURRENT-LOCATION` construct registers a closure as the event-handler that is invoked whenever there is change in the current location. Retrieving GPS coordinates using plain Objective-C requires the programmer to define class delegates and implement callback methods. We have further demonstrated the use of closures as event-handlers that react to changes in the environment (e.g., appearance of a new service). For instance, the `when-discovered` construct registers a closure as an event-handler that is invoked whenever a new service is encountered in the network.

Scheme as a scripting environment. The dynamic interactive Scheme environment on the iPhone enables one to evaluate Scheme scripts directly on the device. Scripting on the iPhone provides support to interact with native applications such as SMS, mail, phone and iPod. One immediate benefit of such an environment is that it serves as an experimentation platform for new ideas or rapid prototyping applications without facing the complexities of defining classes and GUIs as is the case in plain Objective-C. For instance, using the constructs `CURRENT-LOCATION`, `make-call` and `send-sms` it is possible to write applications that explore the GPS, phone, and SMS capabilities in a few lines of code. The maps application example clearly demonstrates this.

In this paper, we have realised the above benefits of integrating Scheme and Objective-C using a locally developed Scheme interpreter [9]. However, the symbiosis layer and the event-driven constructs are independent of the Scheme interpreter, and could be implemented on top of any other Scheme implementation or other languages with similar features as Scheme.

6. IMPLEMENTATION

In this section we discuss how the Scheme and Objective-C symbiosis has been implemented. The symbiosis implementation essentially has two sides: Objective-C to Scheme where the interpreter is embedded in Objective-C applications, and the Scheme to Objective-C where Scheme programs have access to the Objective-C runtime. First, we explain how the Scheme interpreter is embedded in Objective-C applications.

6.1. Embedding Scheme in Objective-C Applications

The Scheme interpreter used was developed at our lab and is implemented in ANSI-standard C, and it is thus fully compatible with Objective-C. Embedding a Scheme interpreter in Objective-C enables the use of Scheme code within an Objective-C application. This implies that one can write parts of the application in Objective-C (e.g., the application front-end) and others in Scheme (e.g., the application logic). In Section 2, we presented a simple *iScheme* interactive environment for evaluating Scheme code on the iPhone (in a REPL fashion). At first glance, it looks like a regular Objective-C application but it actually embeds the Scheme interpreter that does the code evaluation. The *iScheme* system is approximately 736kb and can be deployed to the iPhone, the iPod touch or the iPad devices like any other third party application without modifying the iPhone OS.

In what follows, we explain the step-by-step procedure of setting up the Scheme environment in Objective-C.

6.1.1. Initialising the Scheme Interpreter from Objective-C

The Scheme interpreter provides an API that enables an Objective-C application to instantiate and interact with the interpreter. The API is wrapped as an Objective-C class `iScheme` that provides methods to evaluate Scheme expressions, getting the results of expressions and error messages. The interpreter is initialised by creating an instance of the `iScheme` class. Internally, the initialisation involves setting up storage space and the Scheme evaluation environment. Multiple instances of the interpreter can be created and are completely independent. The following code snippet illustrates the initialisation step.

```
iScheme *schemeVM = [[iScheme alloc] initWithDelegate:theDelegate];
```

The `iScheme` object is initialised with the `theDelegate` object that must implement the `receiveResult:` method to handle the result of evaluating a Scheme expression. If there is no delegate specified, the default implementation of `receiveResult:` is used which simply logs the results to the standard output.

6.1.2. Evaluating Scheme Expressions in Objective-C

Once the Scheme interpreter is successfully initialised, Scheme expressions are evaluated by calling the C macro `SCHEME_CALL(function-name, args)` (explained in Section 3) or the `schemeCall:` method as follows:

```
[schemeVM schemeCall:schemeCode];
```

The argument to the `schemeCall:` method is a string representing the Scheme code. The result from evaluating the Scheme expression is returned via the `receiveResult:` delegate method. In case the expression evaluation results in an error, the result string is the error description (e.g., `"undefined variable"`).

6.2. Implementation of the Scheme and Objective-C Symbiosis

So far, we have explained how to embed Scheme programs in Objective-C applications, but a little about the behind the scenes of the Scheme and Objective-C symbiosis. As discussed in Section 3, our symbiosis implementation is based on the previous work on language symbiosis–*the linguistic symbiosis model* [18]. The symbiosis constructs explained in Section 3 are implemented using Scheme macros that translate to native functions which in turn are implemented using Objective-C's reflective capabilities. Let us first briefly explain the reflective capabilities of Objective-C before delving into the implementation details of the Scheme and Objective-C symbiosis.

6.2.1. The Objective-C Reflective Capabilities

Objective-C is an object-orientation extension to C language using Smalltalk-80 [14]-like semantics. It is a proper superset of C meaning that it is possible to include C code within Objective-C code. In fact, the Objective-C compiler translates every method call in Objective-C into a C function call. The Objective-C runtime defines data structures to capture information about classes as well as selectors, instance variables' templates needed for introspection [20]. In this paper, we use Objective-C 2.0, the modern Objective-C version used by iPhone applications and 64-bit programs on Mac OS X v10.5 and later. In what follows, we explain how Objective-C programs can interact with the runtime system.

Interacting with the Objective-C Runtime. Language symbiosis with Objective-C is possible because of its dynamism and reflective capabilities. The Objective-C runtime library provides functions to access information such as name of a class, a number of and what methods are implemented (introspection) by a class. It also allows objects to modify their own structure (intercession) [20] such as add new variables, add new classes, or replace method implementations at runtime.

Messaging. Objective-C is a dynamic language in the sense that messages are not bound to their respective method implementations until runtime. All Objective-C message sends are converted into `objc_msgSend` function calls. For every message send expression `[receiver message]` the compiler generates a call on the messaging function `objc_msgSend` as follows:

```
objc_msgSend(theReceiver, theSelector, arg1....argn);
```

The `objc_msgSend` function takes at least two arguments: `theReceiver` which is the receiver object, `theSelector` which is the method name that handles the message and a variable number of arguments for the specified method. In the remainder of this section we explain how we implement the language symbiosis between Scheme and Objective-C based on `objc_msgSend` and other runtime functions.

6.2.2. Accessing Objective-C Classes from Scheme

The `OBJC-CLASS` construct illustrated in Section 3 is a macro that translates to the native function `objc-string->class` as follows:

```
; an Objective-C class from a Scheme string representing a class name
(define-macro  OBJC-CLASS
  (lambda (class-name)
    `(let* ((class-string ,(symbol->string class-name))
            (objc-class  (objc-string->class class-string) ))
       objc-class)))
```

The `define-macro` construct defines a macro `OBJC-CLASS` that takes an Objective-C class name parameter and converts it into a string that is then used as an argument in the call to the `objc-string->class` function. For instance, `(OBJC-CLASS NSSpeechSynthesizer)` translates to `(objc-string->class "NSSpeechSynthesizer")`. The `objc-string->class` function is based on the Objective-C runtime library function `objc_getClass(const char *class_name)` which takes the string name of a class and returns its definition. The returned pointer to the class definition is reified as an `OBJC_TYPE` Scheme value so that it can be bound to a Scheme variable. The `OBJC_TYPE` Scheme type encapsulates an Objective-C instance in Scheme world and is implemented as a C struct that contains a pointer to the location of the actual Objective-C object.

6.2.3. Sending Messages to Objective-C Objects from Scheme

As we introduced in Section 3, sending messages to Objective-C instances from Scheme is achieved by use of the `OBJC-SEND` construct. `OBJC-SEND` is implemented as a macro that translates to the native `SOC` function call. The `SOC` function takes as argument an Objective-C instance or class, a method name as a string, a variable number of arguments for the method, and performs an Objective-C message send using the `objc_msgSend` function.

Note that a programmer can still perform Objective-C message sends using the `SOC` function. However, from our experience, we found it cumbersome to always specify the method name as a string, thus we implemented a syntactic sugar construct, to wit `OBJC-SEND`. The following code shows the definition of the `OBJC-SEND` macro based on the `SOC` function:

```
; Sending a message to an Objective-C instance
; the receiver may be an Objective-c instance or class
(define-macro  OBJC-SEND
  (lambda (receiver method-name . args)
    `(let* ((selector  ,(symbol->string method-name))
            (objc-object  (SOC ,receiver  selector ,@args)))
       objc-object )))
```

The `define-macro` construct defines a macro `OBJC-SEND` that takes on parameters as the receiver, method name and a list of arguments to the named method which is then transformed into a call to the `SOC` function with the selector converted into a string. For example, `(OBJC-SEND UIDevice currentDevice)` translates to `(SOC <UIDevice:class> "currentDevice")`. The `SOC` function is implemented on top of the Objective-C's `objc_msgSend(id theReceiver, SEL theSelector, ...)` function which is used to send messages to objects in memory. Each `SOC` is subsequently transformed into `objc_msgSend` as we illustrate below:

```
(OBJC-SEND receiver selector arg1 arg2 argn)

  ⇓

(SOC receiver selector-string arg1 arg2 argn)
```

⇓

```
objc_msgSend(receiver, NSSelectorFromString(selector-string), arg1, arg2, argn)
```

The return value of `objc_msgSend` is the return value of the method which is reified as `OBJC_TYPE`. The `NSSelectorFromString` method converts a string to a selector name.

6.2.4. Creating Objective-C Instances from Scheme

Creating an Objective-C instance from Scheme is achieved by means of the `(OBJC-INSTANCE class-name)` construct. `OBC-INSTANCE` is implemented as a macro and translates to the native Scheme `make-objc-instance` function as follows:

```
; Creating an Objective-C instance from a string of an Objective-c class name
(define-macro  OBJC-INSTANCE
  (lambda (class-name)
    `(let* ((class-string ,(symbol->string class-name))
            (objc-object  (make-objc-instance class-string)))
       objc-object )))
```

The macro `OBJC-INSTANCE` that takes one argument representing the name of the class and calls the `make-objc-instance` function with the class name converted to a string. The `make-objc-instance` function is implemented using `objc_msgSend` introduced in Section 6.2.1 by subsequently sending the `alloc` and `init` messages to the named class as follows:

```
void *theClass = objc_getClass(className);
void *theObject1 = objc_msgSend(theClass, @selector(alloc));
void *theObject2 = objc_msgSend(theObject1, @selector(init));
```

The invocation of the `objc_getClass` function returns the class definition of the class named `className`. The subsequent lines of code allocate and initialise the instance of the named class using the `alloc` and the `init` messages, respectively. `theObject2` is a pointer to the resulting instance of the class which is reified as `OBJC_TYPE` Scheme type and returned to Scheme.

6.2.5. Memory Management

One substantial advantage of developing applications in Scheme is that the programmer does not need to manually take care of memory management. This is unlike in Objective-C where the programmer is responsible for manually dealing with the memory management issues. In iScheme, when Objective-C objects cross the boundary to the Scheme world, they are wrapped as a generic Scheme value and the Scheme garbage collector takes care of freeing the memory allocations. Internally, when the garbage collector encounters a wrapper Scheme value holding an Objective-C object, it sends the `release` message to the Objective-C object.

6.3. Limitations

The Scheme and Objective-C symbiosis implementation explained in this section is still in its first version that can still further be improved. We highlight some of the limitations of the current implementation:

Serialisation. In our current implementation, the serialisation mechanism of the datatypes that are parameter passed to a remote function does not support all the first-class values in Scheme. For example, there is no support for serialising closures and continuations.

Memory management. We need to further investigate how the Objective-C `retainCount` and `release` methods integrate with the Scheme garbage collector. The fact that `retainCount` and `release` memory management constructs are treated as regular Objective-C methods, makes it possible to invoke these methods using `OBJC-SEND` construct which may be a potential interference with the Scheme garbage collector.

6.4. Summary

In this section we have discussed the step-by-step procedure on how to embed Scheme programs in an Objective-C application. We subsequently explained how the symbiosis constructs in Scheme: `OBJC-CLASS, OBJC-INSTANCE` and `OBJC-SEND` map to primitive functions `make-objc-instance, objc-string->class` and `SOC`, respectively. We further explain how Scheme native functions interact with the Objective-C runtime using its reflection library functions (such as `objc_msgSend`) to perform message sends to instances in memory, and `objc_getClass` to load class definitions at runtime.

7. LESSONS LEARNED

In this section we put forward our experiences gathered from porting Scheme to the iPhone, implementing a language symbiosis between Objective-C and Scheme, and implementing constructs that ease the development of the event-driven applications for mobile devices. We generalise the key concepts to make our experience usable to port other programming languages to the iPhone device.

7.1. On Implementing Language Symbiosis with Objective-C

Implementing language symbiosis with Objective-C is possible because of its dynamism and reflective capabilities. The Objective-C runtime library provides functions to perform introspection (e.g., access to the methods a class implements) and intercession (e.g., adding a class, replacing a method implementation) on Objective-C objects at runtime. Below, we summarise the language constructs that one needs to implement in order to realise an interaction with Objective-C. For each construct we point out the relevant key Objective-C runtime functions required to implement it.

Table I. Preliminary benchmarks on the method call overhead from Scheme to Objective-C

	Number of parameters		
	0	1	2
Method calls in Objective-C (ms)	0.00022	0.00023	0.00025
Method calls from Scheme to Objective-C (ms)	0.13500	0.15500	0.16150

First, the symbiotic language needs to define a language construct for loading Objective-C classes. For this, the Objective-C runtime library provides the function `objc_getClass(`*`const *class_name`*`)` that takes the string name of a class and returns a pointer to the class definition. In our Scheme, we implemented the `OBJC-CLASS` construct.

Second, an construct is required to provide means to send messages to Objective-C instances from the symbiotic language. To this end, the Objective-C runtime provides a function `objc_msgSend(`*`theReceiver, theSelector, args`*`)` that performs message sends to an Objective-C object (the receiver) given a name of a method (the selector), and an optional variable number of arguments. A string representing a method name can be converted to a selector using the `NSSelectorFromString` runtime library function. In our Scheme, we implemented the `OBJC-SEND` construct.

The third set of language constructs is type conversion functions. Type conversion can be implicit – meaning that values of one language are automatically converted to another language as they cross the bridge, or explicit – meaning that the programmer makes use of the conversion functions. In our implementation, we perform automatic conversion when Scheme values cross to Objective-C and provide the programmer with conversion functions (such as `NSString->string`) to convert Objective-C objects to Scheme values.

Other than the language symbiosis constructs that enable interaction between two languages, there are extensions required in the symbiotic language. First, the symbiotic language needs to be extended with a generic representation of the Objective-C objects. For example, we extend the Scheme value types with the `OBJC_TYPE` as a wrapper for Objective-C objects in Scheme. Second, a native function needs to be added to the symbiotic language to handle the calls to Objective-C. For example, we extend our Scheme interpreter with the `SOC` native function that serves as an interface to the Objective-C world.

7.2. On the Method Call Overhead

Implementing language symbiosis between two languages involves a sacrifice on performance. For example, method calls from Scheme to Objective-C involve a significant overhead compared to method calls in plain Objective-C. We performed preliminary benchmarks to quantify the method call overhead caused by the symbiosis.

We measured the method call overhead (in ms) on the iPhone 3G with ARM1176 412MHz and 128MB RAM running iPhone OS 3.1.3. We considered three different methods that vary by the number of parameters (zero, one, and two). All three methods have empty bodies and the parameters are of type `NSNumber`. Table I shows the times per method call for the three methods (in plain Objective-C versus calls from Scheme to Objective-C). For each measurement we performed one million method calls.

There is a significant increase in the times of the method calls from Scheme to Objective-C. We have not applied any optimisation techniques in our current implementation. One possible factor for this increase is the fact that every Objective-C instance passed to Scheme needs to be wrapped as a Scheme generic value `OBJC_TYPE`. In addition, for each Scheme value passed to an Objective-C method, the symbiosis layer needs to check and perform a type conversion to the appropriate Objective-C object. As future, we would like to try performance enhancing techniques (such as caching of the selectors and method implementations).

8. RELATED WORK

In this section we discuss existing work on Scheme implementations for mobile platforms and language bridges to Objective-C or Scheme.

8.1. Scheme Implementations on Mobile Platforms

The Gambit-C Scheme system [11] has been recently used to develop a number realistic iPhone applications (available in the iPhone's App Store). Gambit-C compiles Scheme programs to C code that is compatible with Objective-C, and it can be compiled and deployed to the iPhone OS. The Gambit-C system also includes an interpreter that can used to provide an interactive environment with support for a remote REPL. To interact with Objective-C, Gambit-C employs the foreign function interface (FFI) approach. FFIs in Gambit-C support interaction in either direction, and as such it qualifies as a language symbiosis. In order to access Objective-C methods from Scheme, wrapper C functions are generated. The need to generate wrapper functions in C for Objective-C methods could be eliminated by adopting Objective-C's reflective API that we employ in iScheme. The use of the reflective API also implies that Scheme programs have direct interaction with Objective-C which facilitates the implementation of event-driven constructs (e.g., for accessing iPhone capabilities and distributed programming).

Moby Scheme [22] is an experimental Scheme compiler for smart phones with particular target for the Android OS [15]. It allows developers to write Scheme programs that are fed to the compiler to generate Java source code. The Moby Scheme compiler is mostly written in PLT Scheme [13] – a Scheme implementation that is designed to run on traditional desktop platforms and not on mobile devices. The Moby Scheme compiler itself does not run on Android OS, but it generates Java source code from the desktop platform. The generated Java source code is used to produce the Android `.apk` packages that can take advantage of the native features (e.g., GPS and SMS) available on the device. One important feature of Moby Scheme is its support for event-driven programming based on the notion of "Worlds" [12] that enables one to write programs that react to events (e.g., an incoming SMS event). As in Moby Scheme, we believe that the event-driven programming style is well-suited for a mobile setting because of their inherent interactive nature. In iScheme, we further explore the event-driven programming model for developing distributed applications by providing constructs for peer-to-peer service discovery, asynchronous remote communication, and handling network failures.

8.2. Linguistic Symbiosis

Linguistic symbiosis is not novel and has been explored in existing approaches as a way to enable two languages to invoke each other's behaviour and exchange data [18, 7]. There are several language implementations that explore linguistic symbiosis with Objective-C or Scheme.

Java and Objective-C symbiosis [2] is one of the earliest bridges to Objective-C that allows one to write programs in Java that instantiate and use Objective-C classes from Java, pass Java objects as arguments to Objective-C methods and directly subclass Objective-C classes. PyObjc[25] implements language symbiosis between Python and Objective-C that enables Python programmers to write Cocoa GUI applications on Mac OS X in pure Python. CL-ObjC [5] is a Common Lisp library that enables interaction with Objective-C libraries built on top of a foreign function interface. CL-ObjC provides the `invoke` construct similar to our `OBJC-SEND` to perform message sends to Objective-C instances in a LISP-like way. In addition, CL-OBJC implements an interface that mixes the Common Lisp Object System (CLOS) and Objective-C's object system. Unfortunately, all these existing language symbiosis to Objective-C are limited to desktop development platforms (mostly Mac OS X) and no single implementation ports to mobile platforms. Our Scheme and Objective-C symbiosis mainly aims at providing access to features specific to mobile platforms such as phone, SMS and GPS.

There do exist Scheme implementations that define language symbiosis with other programming languages such as C and Java. We will focus our discussion on approaches that implement symbiosis between Scheme and an object-oriented programming language.

Dot-Scheme [24] is a library that builds an FFI for PLT Scheme [13] with Microsoft .NET Common Language Runtime (CLR). It provides the `import-assembly` construct that loads the assembly code of a class into Scheme. For each loaded class a Scheme proxy is generated that wraps the class as a Scheme value. Dot-Scheme enables invoking the CLR methods using the Scheme-like syntax though it does not provide support to invoke Scheme functions from the CLR.

Kawa [4] and SILK [1] are Scheme implementations that enable interaction between Scheme and Java. Kawa provides functions to invoke Java methods from Scheme. The `invoke` construct in Kawa is similar to `OBJC-SEND` construct in our approach. In addition, Kawa provides different variants of `invoke`, namely, `invoke-static` to call static methods, and `invoke-special` to call only methods in the super class. SILK provides constructs `import` and `class` to load Java packages and classes, respectively. The `(class "java class name")` construct in SILK is similar to the `OBJC-CLASS` construct in our approach.

Surprisingly, neither Kawa nor SILK provides syntactic sugar for Scheme/Java interaction. For instance, in Kawa the Java method name argument for `invoke` construct must be a string or a symbol. SILK requires the programmer to specify the Java package or the class to be imported as a string. However, SILK provides support for defining new methods on Java classes from Scheme and the ability to reflect on Java objects, the features that we have not explored in our current implementation. Our approach differs from most of the existing Scheme and other languages symbiosis in that we aim at building constructs that ease the development of applications that run on mobile devices while exploiting the capabilities specific to mobile platforms.

9. CONCLUSION AND FUTURE WORK

In this paper, we report on our experience of porting a Scheme implementation to the iPhone platform. We have implemented a language symbiosis between Scheme and Objective-C using the reflective API of Objective-C. The symbiosis layer enables developers to write iPhone applications in Scheme, thus benefiting from its well-known features such as automatic garbage collection, higher-order functions (for event-driven programming), and structural macros (for syntactic extension). We have designed constructs for accessing iPhone capabilities (e.g., GPS information) as well as distribution constructs (for peer-to-peer service discovery, remote communication, and handling network failures). These constructs facilitate the development of local and distributed iPhone applications in an event-driven style. We have presented a series of examples that demonstrate the benefits of the Scheme and Objective-C interaction as well as the constructs provided on top of the symbiosis layer. Finally, our discussion of the lessons learned in our setting generalise the key concepts of building a language symbiosis with Objective-C, thus making our experience usable for other ports of languages to the iPhone or other mobile phone platforms.

Given the benefits of event-driven programming constructs that we have realised, we would like to explore further the possibilities of providing a reactive programming model for the iPhone development. More concretely, we would like to integrate a functional reactive programming model with our distribution constructs. We currently extending the iScheme interactive environment to the iPad with enhanced programming support such as syntax colouring, error reporting, and file management.

ACKNOWLEDGEMENTS

We thank Nick De Cooman for undertaking the AmbiScrabble project to validate the Scheme and Objective-C symbiosis. We are grateful to the anonymous reviewers and the editors who made important suggestions to improve the paper. We would like to thank Éric Tanter for the initial ideas about this work and also suggesting the venue of publication. We thank Ellie D'Hondt for the final review of the paper. This work was partially funded by the Research Foundation - Flanders (FWO), the SAFE-IS project, the MoVES project, the STADiUM project, the VariBru project of the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), and the Prospective Research for Brussels program of IWOIB-IRSIB, Belgium.

REFERENCES

1. K. Anderson and T. J. Hickey. SILK – a playful blend of Scheme and Java. In *Proceedings of the Scheme and Functional Programming Workshop*, September 2000.
2. Apple Inc. Using the Java bridge, September 2009. http://developer.apple.com/legacy/mac/library/.
3. Apple Inc. The iPhone development center, 2010. http://developer.apple.com/iphone/.
4. P. Bothner. Kawa: compiling dynamic languages to the Java VM. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 41–41, Berkeley, CA, USA, 1998. USENIX Association.
5. G. Cant. A portable Objective-C bridge for Common Lisp, September 2009. http://www.common-lisp.org/project/cl-objc/.
6. T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *SCCC '07: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
7. T. V. Cutsem, S. Mostinckx, and W. D. Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems & Structures*, 35(1):80–98, 2009.
8. J. Dedecker, T. V. Cutsem, S. Mostinckx, and W. D. Meuter. Ambient-oriented programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP*, pages 230–254. Springer, 2006.
9. T. D'Hondt. The skem interpreter, 2010. http://soft.vub.ac.be/soft/skem.
10. P. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003.
11. M. Feeley. The Gambit Scheme system, September 2009. http://dynamo.iro.umontreal.ca/~gambit.
12. M. Felleisen, K. Fisler, and S. Krishnamurthi. How to design worlds: Imaginative programming in DrScheme., June 2010. http://world.cs.brown.edu.
13. M. Flatt, R. B. Findler, and PLT. Guide: PLT scheme. Introduction PLT-TR2009-guide-v4.2.3, PLT Scheme Inc., December 2009. http://plt-scheme.org/techreports/.
14. W. Golubski. A complete semantics for Smalltalk-80. *Computer Languages*, 21(2):67–79, July 1995.
15. Google Inc. Android OS, September 2009. http://www.android.com/.
16. Google Inc. Google Maps for mobile, September 2009. http://www.google.com/mobile/products/maps.html.
17. K. Gybels. SOUL and Smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003.
18. K. Gybels, R. Wuyts, S. Ducasse, and M. D'Hondt. Inter-language reflection - a conceptual model and its implementation. *Journal of Computer Languages, Systems and Structures*, 32:109–124, 2006.
19. E. Kaasinen. User needs for location-aware mobile services. *Personal Ubiquitous Computing*, 7(1):70–79, 2003.
20. G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.
21. S. Kochan. *Programming in Objective-C 2.0.* Addison-Wesley Professional, 2009.
22. S. Krishnamurthi. The Moby Scheme compiler for smartphones or, is that a parenthesis in your pocket? In *Proceedings of the International Lisp Conference (ILC 2009)*, 2009.
23. C. Mascolo, L. Capra, and W. Emmerich. Mobile computing middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag, 2002.
24. P. Pedro. Dot-Scheme: A PLT Scheme FFI for the .NET framework. In *Proceedings of the Scheme and Functional Programming Workshop*, 2003.
25. PyObjC Project. PyObjC: The Python <-> Objective-C bridge, September 2009. http://pyobjc.sourceforge.net/.
26. TweakerSoft. Aroundme, September 2009. http://www.tweakersoft.com/mobile/aroundme.html.
27. T. Van Cutsem, S. Mostinckx, and W. De Meuter. Linguistic symbiosis between actors and threads. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 222–248, New York, NY, USA, 2007. ACM.
28. Wikipedia. App store, 2010. http://en.wikipedia.org/wiki/App_Store.