Platform Independent, Higher-Order, Statically Checked Mobile Applications

Dean Kramer, Tony Clark, and Samia Oussena

***Abstract*—There is increasing interest in establishing a presence in the mobile application market, with platforms including Apple iPhone, Google Android and Microsoft Windows Mobile. Because of the differences in platform languages, frameworks, and device hardware development of an application for more than one platform can be a difficult task. In this paper we address this problem by the creation of a mobile Domain Specific Language (DSL). Domain analysis is carried out using two case studies, leading to the identification of basic requirements for the language. The language is defined as an extension of a λ-calculus in terms of an operation semantics and a type system. The language is a contribution to the understanding of mobile applications since it precisely defines the essential properties offered by a range of mobile application technologies, and can form the basis for a single language that can target multiple platforms.**

***Index Terms*—Domain Specific Languages, Mobile Computing, Platform-Independence**

I. Introduction

TODAY, the penetration of modern smart phones is vastly increasing with over 172 million smart phones shipped worldwide in 2009 [1], and with the emergence and successes of sources for consumers to install third party applications opens a new market for developers to reach consumers. However, developing an application for multiple mobile platforms can incur different obstacles including differences in development tools available, different languages, platform constraints and availability of software libraries.

Difficulties in producing software for more than a single platform has been evident for many years outside of the mobile realm. For decades, software portability has been a concern during development, mainly due to very large spectrum of different CPU Instruction Set Architectures (ISA) and by the large variety of Operating Systems in use. Recently this has become less of an issue, largely due to factors including the decrease in CPU ISAs, the dominance of a limited number of operating systems, and to commonly used languages including Java. Since the mobile market is relatively immature, there are large differences in implementation languages and development environments used for different applications and technology platforms.

Software porting and cross platform development remains the most common method for multi-platform development. For large software companies this is not an issue, but for smaller commercial mobile application businesses this presents a problem. Firstly, each mobile platform has a different implementation language and therefore each developer needs to be multi-lingual or a company needs to set up multiple development teams for each product. Secondly, businesses must invest in multiple types of testing equipment for the different platforms. These factors lead to an economic driver for technology that can help to deliver mobile applications over families of target platforms.

D. Kramer and S. Oussena: School of Computing and Technology, University of West London, London, W5 5RF, UK e-mail: dean.kramer@uwl.ac.uk and samia.oussena@uwl.ac.uk.

T. Clark: School of Engineering and Information Sciences, Middlesex University, London, NW4 4BT, UK email: t.n.clark@mdx.ac.uk.

A. Outline of Paper

The outline of this paper is as follows: section II describes the background to mobile application development; section III outlines current approaches to multiplatform methods and defines our contribution; section IV describes two case studies that we have implemented for a company and which leads to the identification of the domain features for our language; section V defines the syntax of the language and gives a simple example application; section VI defines how the language executes; finally section VII analyses the language and describes our current and future implementation strategies.

II. Background and Related Work

A Software Development Kit (SDK) provides the environment for developing applications through the use of libraries, emulators, and debuggers and is the basis for the development of the majority of current mobile applications. However an unfortunate drawback of SDKs is that they are platform dependant: Apple provides an Xcode SDK (http://developer.apple.com/devcenter/ios) that includes an interface builder, an iPhone simulator and development environment; the Android SDK comes as an eclipse plug-in, and includes a software emulator; the Windows Phone provides a specialised version of the Microsoft Visual Environment; Blackberry and Symbian also provide different SDK platforms. This variance in terms of specific platforms greatly increases the costs of developing mobile applications to be accessible in a variety of devices. In order to reduce these costs, various frameworks have been proposed in order to produce cross-platform applications.

A. Frameworks

The DIMAG Framework [2] was developed for automatic multiple mobile platform application generation. This was accomplished by creating a declarative definition language which is comprised of 3 distinct parts; firstly a language

DIMAG-root, which provides references to the definitions for workflow and user interface in the application; secondly the language State Chart eXtensible Markup Language (SCXML) defines the workflow by the definitions of states, state transitions, and condition based actions; and finally DIMAG-UI language based on MyMobileWeb's IDEAL language using CSS to control the user interface.The main shortcomings of this method is that it relies on server-side code generation and download. The other difference with our work is that applications developed in this framework are interpreted using a virtual machine.

Other frameworks include XMLVM [3], [4] developed at San Francisco State University and created to support byte-code cross-compilation and avoid source-code translation through the use of a tool chain. This tool chain currently translates Java Class files and .Net executable to XML documents, which then can be output to Java byte code/.NET CIL or to JavaScript and Objective-C. This tool chain was firstly used to cross compile Java applications to AJAX applications [5], because of the lack of IDE support and difficulty in creating an AJAX application. Further work to include Android to iPhone application cross-compilation [6] was completed. API mapping between the two platforms was carried by the creation of a compatibility library.

More recently, since the arrival of HTML5 and WebKit, a number of open source and commercial cross-platform frameworks have been proposed such as the Appcelaterator (http://developer.appcelerator.com), PhoneGap (http://www.phonegap.com) and Rhomobile (http://rhomobile.com). Frameworks, which use either JavaScript or Ruby, and therefore the resulting applications are run in a browser. Furthermore these applications can run offline and access the device's full capabilities; such as a GPS or camera; providing the same look and feel as a native application. Although these frameworks greatly simplify the task of implementing the mobile application, the developer is still required to work with general web application languages which lack specificity and efficiency in terms of mobile applications. Furthermore, these applications suffer limited visibility on the market due to the absence of an "official" distribution channel.

There are several different software platforms for mobile applications. Since commercial developers usually wish to develop an application that can work over all platforms there are a number of proposals for single technologies that can target multiple platforms. A recent proposal for a DSL for mobile applications [7] uses XText and Eclipse to implement a DSL that uses code generation techniques to target mobile platforms. This DSL uses fixed GUI structures such as `section` whereas our language leaves the collection of external widgets as a parameter of any use of the system. It is also not clear whether the DSL has a static type system and its semantics is not defined independently of a translation to a target platform.

Mobl (http://www.mobl-lang.org/) is a language that has been designed to support mobile application development and which targets JavaScript. It has many things in common with our language, however the language features for describing GUI components is fixed and the semantics is not defined independently of the target language.

In all cases the currently available DSLs for mobile applications differ from the language presented in this article in that they are not higher-order. By providing functions as first-class data values, our language provides unrestricted abstraction over mobile programs. Therefore our language can express patterns, such as required by product lines, by parameterising over reusable application elements. Existing languages are also fixed in terms of the external widgets. Our language is parameterised with respect to external widgets and they are simply provided at the type-checking phase when they are checked in terms of the events that they can raise.

Our language is textual and does not aim to provide a graphical DSL that can be used to shield developers from the technical details of application design (although it is intended to shield application developers from the details of mobile platforms). Another approach to DSL design involves graphical modelling languages such as that defined in [8], however it is not clear that such approaches can abstract from technical details without resorting to a textual DSL at some level.

Links [9] is a language that has been designed to support web application development where the 3-tier architecture is supported by a single technology. Like the technology described here, Links supports higher-order functions and is statically typed with respect to events and messages. Unlike our language, Links has been designed as a complete language with supporting tools, and indicates a possible future direction for layering a user language on the calculus.

B. Domain Specific Languages and Modelling

Domain-specific languages (DSLs) have provided the support for software development process by raising the level of abstraction and introducing specialised viewpoints of a certain problem space. The benefit of DSL to application development has been described in [10], [11], [12], [13]. More recent DSLs in other areas include [14] that concentrates on the abstraction of web applications to lower the overall complexity of the application and boilerplate code. Further work on this DSL led to the creation of Platform Independent Language (PIL) [15]. PIL was developed as an intermediate language, to provide a scalable method for developing for multiple platforms. A drawback of this method is currently it lacks support for mobile platform development.

Other efforts for making mobile application development easier include Google Simple (http://code.google.com/p/simple/), a BASIC dialect for creating Android applications, and more recently the Google App Inventor (http://appinventor.googlelabs.com/about/), which is based on Openblocks [16] and Kawa (http://www.gnu.org/software/kawa/). Particularly Google App Inventor has vastly abstracted app development, but only supports development of Android applications.

DSLs for mobile application have been limited due to a number of factors, such as the rapid change of the devices, the closeness of the distribution channel. However, with the

popularity of the native-look-a-like web applications more DSLs are starting to be developed. An earlier work in this area, Balagtas-Fernandez have looked at the design of graphical DSL for non-technical users [17]. This work is still at an early stage and the tool to support the DSL is a prototype. However, the initial survey of potential users has shown that non-technical users would be interested in developing their own application and would require an easy interface to do so.

More recently, Brenhs has proposed MDSD, a DSL for iPhone. The language is more specific to data centric applications. Following from that work, they have started the Applause project for developing DSL for iPhone, iPad and Android (http://code.google.com/p/applause/), but this is still not fully developed.

The SERG group defined a language named Mobl (http://www.mobl-lang.org/) with a declarative DSL for both the user interface and for defining the data. Although mobl is comparable to our language there are substantial differences since we have aimed to capture the essential features of mobile applications through the use of: technology independence; static typing for all features; higher-order functions; formally defined operational semantics; widget libraries.

Our aim is for the language presented in this article to be a formal foundation for each of the current implementations of mobile applications DSLs.

III. Multi-Platform Development and Contribution

Because of the complexities in multiplatform development, in this section we introduce three methods that could be used to help application development for multiple platforms. Our proposal is to select an approach based on Domain Specific Languages.

A. Approaches to Development

Frameworks: The use of frameworks can be seen as a method of software abstraction using common code, which can be overridden and extended by a user. Within mobile development, frameworks have been developed to help with specific tasks including media playback, access to sensors and graphic and UI manipulation. For example, the system described in [6] has been designed to help make code bindings between the different platform specific frameworks. This method concentrates on solving all computational problems, which can increase complexity in application development, further becoming a hindrance to the developer.

Web Applications: A mobile web application essentially is a regular Internet application designed to fit the average screen sizes of most mobile devices, bringing various benefits to the developer. Some applications that require high amounts of processing can greatly benefit from allowing the processing to be handled in the cloud while the device merely has to process the UI. In addition the use of standards such as HTML and CSS may make certain types of applications easier to develop.

Originally this approach was troublesome because of reliance on network connectivity for the application, which in some situations may either not be available or not desired. Web applications also couldn't store local data to the web browser, until the development of HTML5 [18]. In May 2007, Google released a plug-in for the Firefox web browser, Google Gears (http://gears.google.com/). This plug-in supports caching of web applications to allow offline use, and also the ability for a web application to store data in a local database. This idea has been integrated into the development of HTML5, a step in the right direction but there are still remaining issues. Firstly, web applications in general can have shortcomings in the amount of rich UI widgets, with animation for certain widget interaction being increasing difficult to implement in a mobile web application. Other problems are limitations of the web-browser on the mobile devices, possibly leading to inconsistencies in application functionality between different platforms because of lack of API for using different device components (e.g. accelerometers, vibration motors, GPS etc). Because of these limitations and current unsolvable dependencies, the creation of a Domain Specific Language was chosen as our solution.

Domain Specific Languages: A Domain Specific Language (DSL) [19], [20] is primarily designed to be used in a certain application domain (e.g. mobile, telecoms, finance), abstracting away from the software implementation making implementation easier. The abstraction is designed to aid the developer and is to be contrasted with General Purpose Languages (GPLs) whose features are not designed with any particular domain in mind. DSLs have existed for many years. Languages that were created for particular domains include FORTRAN [21] used to allow direct mathematical formula, Structured Query Language (SQL) [22] for database access and manipulation, and Algol [23] for algorithm specification. In recent times, the use of DSLs have been proposed and used in different domains including the production of rich web applications [14], mashups of web applications and services [24], and system integration [25]. Because of the complexities in mobile development, we believe there is room for abstraction in the development for mobile devices.

B. Problem, Proposal and Contribution

There are many different technologies for mobile-application development. Most of these are complex and do not separate out the application logic from the GUI implementation. Such a separation is sensible because the application logic can usually be reused across multiple platforms while the GUI implementation must be changed. Furthermore, most technologies for mobile-application development are event driven, but dynamically match event-handlers with the events when they are raised. Our claim is that the quality of development can be enhanced by statically checking that handlers for all events are defined before the application is executed.

Our proposal is that the essential features of languages for mobile application development have some characteristic elements that can be captured in a *domain specific language*. Furthermore, the language can be constructed as an extension of the simply typed λ-calculus where the extensions are both orthogonal and which characterise the domain. Since our language is based on the λ-calculus it is highly expressive in terms of being able to parameterise over the elements of an

application and thereby encode patterns of data (e.g. reusable widgets) and control (e.g. call-backs and event handlers).

We use an approach based on monads to contain those parts of an application that deal with updating state (SQLite for example). As described in [26] this supports the desirable situation where applications can be built from composable units. The language has a formally defined semantics that is independent of any implementation technology and therefore can be used as a blueprint for an implementation in any target platform, or can be implemented directly as an interpreter.

The language uses external widgets to define platform-specific features such as GUI elements and persistent storage. The external widgets are fully integrated with the type system of the language and therefore, by supplying different collections of external widgets, the language represents a family of related development platforms. Finally, the language has a type system that allows mobile applications to be statically checked. In particular the events raised by GUI widgets and the device can be checked against handlers defined in the application before it is executed.

IV. Domain Analysis

A DSL is defined by performing a *domain analysis* [20] on a target family of applications in order to identify the common characteristic features. The domain analysis leads to the design of a technology that conveniently supports these features. Our aim is to define a language that can be used to represent mobile applications and therefore our domain analysis starts with the construction and analysis of two phone applications developed as part of University business and enterprise activities. This section describes the case studies and then outlines the characteristic features that were identified.

A. Case Studies

Two iPhone application case studies were created for a local Small to Medium sized Enterprise (SME). These applications are described in the following two sub-sections.

Tour de France (TDF2009): This application was created to provide access to information from the 2009 series Tour De France cycle race. Firstly the application required a method of transferring and receiving data from an external server for two different reasons. Firstly for the stage results, and secondly for the general data including information about the Teams/Riders and all the Stages involved in that year, this helped us achieve a very small installation size. The data communications were done via XML files parsed using the iPhone SAX-XML Parser, one created with the static data, and one generated every day with the current results. Inside the stages section, fly-through videos to help illustrate the course and terrain, large high resolution gesture controlled pictures were incorporated. Key features of the TDF2009 are shown in Figure 1 where a main screen provides access to the results of current stages and to fly-through videos. The figure shows that the application is driven by events caused by the user touching the screen and that the application consists of different displays (or states) consisting of a mixture of event processors (buttons) and simple information (images, text).

Lyrical Genius: This application was created as a game, Lyrical Genius (LG), consisting of quiz questions relating to different lyrics in songs. This game though still using the Apple Cocoa Framework is quite different to TDF2009 in many ways. Firstly this application does not use XML files as persistence of data, but uses a SQ Lite database for storing level and question data. Other features of this game include music that is played in the background that can be switched on/off and sound effects for if the user chooses the correct or incorrect answer. These features require threading, which is one issue we must consider in the DSL. The game also includes a timer, for which the user must get a number of correct answers within a time limit. This makes use of threading again, and also another important area as the use of timing can be needed in many different contexts. Features of the application are shown in Figure 2 where a player starts at the main screen. On choosing to play, the user is shown their current score (accessed via the database), if they choose to play then they are offered a difficulty level for each question. Each question has a time-limit; the final screen is reached by either selecting the correct lyric or when the time-out occurs.

B. Domain Features

Based on the case studies above, we can define a set of features that the DSL must support. In the case of GUI implementation, in the iPhone and Android development Openly can be used. Openly is a cross-platform graphics language which supports the ability to draw 2D and 3D objects, but in this paper we are concentrating on the platform framework for the GUI.

Screen Size: Mobiles support only a limited size display. This size leads to a relatively small number of GUI features, therefore there is more scope for building these features into a common language. The standard iPhone resolution is 480 by 320 pixel and the IPA supports a 1024 by 768 resolution. This compares to the Android screens, which vary by hardware vendor but resolutions range to about 480 by 800 pixel. Apple have currently settled the differences in screen display resolution by the use of graphic scaling. This method can seem an effective way of allow iPhone apps to run on a IPA, but this comes with its flaws. Graphic scaling of very small low quality images can make them look unappealing to user. Also II design on IPA, because of its size difference will be slightly different than on the iPhone and will require developers to create applications with interfaces to suit that device. This issue leads us to conclude that the DSL should aim to abstract as much application logic as possible away from layout details that can be provided as platform-specific libraries.

Layout Control: Layout control is an important consideration. Android controls layout through the use of XML files, supporting different layout styles. The main style types consisted of linear, relative and absolute. Android now has now deprecated absolute positioning, due to the fragmentation in different hardware vendor screen resolutions (see above). This compares to iPhone, which can do programmatic layout and XML type interfaces using Interface Builder. Interface Builder can help the user easily create UIs, but these layouts would be less

Figure 1. A tour de france mobile application showing state transitions

dynamic than programmatic ones. Like screen size, the DSL should focus on application logic and factor out the layout control details into external libraries.

GUI Element Containership: Both iPhone and Google Android platforms use a form of GUI element containership. In iPhone development, the emphasise is on the application *Window* and it's *Views*, with *Subviews*. These are then 'stacked' onto each other to create anything from a simple to complex interface. With the Android a similar model is used, except with *Views* and *ViewGroups*. Interface control on both platform have similarities and differences. On the iPhone, views are normally controlled by the use of View Controllers, which are where widget event handlers are implemented. In comparison Android development uses *Intent*s and *Activity*s. This feature leads us to conclude that all mobile application GUIs can be expressed in terms of a tree of `widgets` that manage data and behaviour and whose detailed layout and rendering properties can be factored into platform specific libraries.

Event Driven Applications: The applications we are targeting are event driven. Most mobile application implementation languages register event handlers dynamically. This method means there is a lack of checking at compile time to prevent an application crashing. An example of a event listener for iPhone:

```
[btnMenu
  addTarget:self
  action:@selector(backToMenu)
  forControlEvents:UIControlEventTouchUpInside];
```

If the action is not registered then the program might go wrong. This is an issue in general with event driven programming and in particular with mobile applications where events can be supplied by both the platform and the user. Contextual events such as platform orientation, GPS, and battery levels should be handled by an application in suitable ways. This places a desirable feature requirement on our DSL whereby the presence or otherwise of event handlers can be detected at compile-time.

Hardware Features: Modern day mobile devices come equipped with many different features. These features include microphones, accelerometers, GPS, camera, and close range sensors. These features tend to be fairly standard in their behaviour if they are supported by the platform. Although many platforms have comparable hardware features, they differ in the details of how to control and respond to them. The DSL should allow the details of hardware to be factored out into platform specific libraries whilst supporting the events and controls associated with them.

Concurrency: The use of concurrency in mobile applications is paramount. This is carried out by the use of threads, for instance a UI thread starts with the execution of an iPhone or Android app. Because this thread is used for the UI elements of the application, heavy or concurrent tasks should be allocated in its own thread. This can help avoid UI halts and a 'laggy' experience for the user. On the iPhone platform, threads can

Figure 2. The lyrical genius quiz application showing state transitions

be implemented in various ways including POSIX Threads and NSThread. The difference between the two are that the pThreads are a C/C++ library and NSThread is a Cocoa-native thread. On Android, concurrency can be implemented through the use of Thread Classes, just as you would do it Java. Example of a thread in iPhone:

```
[NSThread
  detachNewThreadSelector: @selector(playMusic)
  toTarget:self withObject:nil];
```

Although threads are important, we are proposing a DSL for *mobile information systems* rather than applications with real-time features such as games. Therefore, it is not clear that very fine-grain control over threads will be important. Rather, it is likely that lightweight concurrent processes are require where control is fairly simply in terms of thread interruption and resumption. Furthermore, we will assume that threads can be associated with components of an application and their control can be integrated with application events. Therefore, the DSL will support lightweight threads, but not necessarily provide any special purpose features for creating and manipulating them.

Object-Orientation: Mobile applications are typically Object-Oriented (OO). In the iPhone the main language used is Objective-C, though support for C++ and the non-OO C can also be used. This compares to Android, which uses Java, but with different libraries and uses the Dalvik Virtual Machine (VM) instead of the Java VM, because its characteristics support mobile devices more. Applications are built by constructing new and extending existing class/object types. The DSL should have OO features including the ability to encapsulate state and associate methods with GUI widgets.

Transitional Behaviour: State machine transitional behaviour is very common in mobile device applications, and can be found on the Android platform. Each Activity can be viewed as a state machine that stores state and actions by the user, which then causes transitions between different views or activities. The DSL will need to support a state-machine view of mobile applications.

Data Persistence: Mobile applications and increasingly persisting data to physical storage between application invocation. This data can sometimes be as simple as general settings that the application needs, but also can also be quite complex and including high amounts of redundancy. Modern smartphone platforms currently have implementations of a SQLite (as in LG above), a lightweight serverless single file database engine. Other methods of storage can be in the forms of general binary/object files that store serialisable objects and which is not highly portable, and XML (as in TDF2009 above); highly portable but requires more storage space, and also can require large amounts of parsing which is not ideal. Therefore, whilst mobile applications require data persistence, the format of the persistence differs between platforms. The DSL must support data persistence and allow the compete state of an application to be saved between invocations; however the details of the data format should be factored out into application-specific libraries.

Contextual Events: Within a mobile application, not all events are directly invoked by the user. Mobile platforms have to deal with event invocation from a range of different sources based on its current contextual environment. For example, when the battery is lower on a phone normally the phone will display a message to the user to recharge the battery.

Static Typing: type systems are used in programming

languages as a method of controlling legal and illegal program behaviour. Static typing differs to dynamic typing in many ways. Firstly, static typing requires all type checking to be carried out during run time, as opposed to dynamic typing that requires checking at run-time. Static typing requires explicit declaration of types unlike dynamic typing shown below:

Static	Dynamic

`// static declaration`
`// of integer foobar`
`int foobar;`
`foobar=10;`
	`// dynamic declaration`
	`// and use of foobar`
	`// simultaneously`
	`foobar=10;`

Though the use of dynamic typing may look attractive as variable initialisation is not required, issues can arise from mistyped variable names. With static typing, if a mistyped variable name is used within a method, the source will not compile, whereas using dynamic typing it will compile making software debugging a much harder task. Therefore the DSL should have a type system that allows as many errors to be caught at compilation-time as possible.

C. The Mobile Application Domain: Conclusion

This section has reviewed two mobile applications and has identified features that are common to the application domain. We have identified the domain as including mobile information systems: the family of applications that process information, are event driven, use contextual information from the platform, have relatively simple hierarchical GUIs, and use simple concurrency. This excludes applications with sophisticated GUIs or that require complex real-time processing, for example games. The proposed DSL should exhibit these characteristic features without unnecessary implementation considerations, factoring them out into libraries, so that it can form the basis of analysis and implementation of more practical languages for mobile applications. The rest of this article describes the DSL, illustrates its use with some examples, and analyses its features.

V. Mobile DSL

A. Overview

Although each mobile platform has a different implementation language, the key features are the same. Our hypothesis is that we can capture the characteristic features in a single language that forms a basis for all other mobile application languages. In order to do so our language must be highly expressive and contain features that support those outlined above. A standard starting point for such a language design is the λ-calculus which is highly expressive, flexible, supports analysis, and is readily extended.

Our language is an extension of a λ-calculus where characteristic features described above are supported as follows:

Widgets: Our language provides a special widget-definition feature. A widget consists of state, behaviour and event handlers. The details of how a widget is rendered and how it runs on the particular mobile platform is outside the scope of our language, however the application logic for each widget is completely defined in the language.

External Widgets: The language is parameterized with respect to the basic widget library that is used in any given application. This allows issues such as screen size, layout, hardware features and concurrency to be separated from the application logic. The external widgets are characterized by their type signatures that allows the application to be statically checked.

Containment: Our language uses a simple notion of containment via widgets, records and lists to express the structure of a mobile application GUI. Furthermore, the semantics of our language (described below) uses the widget containment tree as the basis for handling events.

Events: The semantics of the language is partially defined in terms of handling a sequence of externally generated events. An event occurs at a particular widget, for example the user presses a button. If the widget defines a handler for the event then the event can be processed. If the widget does not define a handler then the search proceeds up the widget containment tree until a suitable handler is found. Static type checking guarantees that an event handler will be found. An event handler is responsible for performing any state updates and then returning a replacement widget for the receiver of the event.

State: Each widget has local state that can be updated by performing commands. The extent of the local state is the life-time of the application. State that has wider extent than the life-time of the application is accommodated by providing the application with predefined variables, for example a database that is shared between applications.

Types: The language has a type system defined in terms of a relation that associates a type to each program. Our claim is that we can construct a static type checker for this type system.

Since the language is an extended λ-calculus, it is very expressive (although expressivity is restricted via static typing). The extensions defined above are novel and allow characterise features of mobile applications to be conveniently expressed. Another novel feature of the language is the semantics that follows a cycle, given a program `prog` and `state` that contains update-able memory locations:

```
screen := empty;
while true {
  command         := reduce(prog);
  (widget,state)  := perform(command,state);
  screen          := replace(screen,widget);
  event           := wait_for_event();
  prog            := handle(screen,event);
}
```

The program `prog` is reduced (evaluated) to produce a command. A command must be performed with respect to a current state (the database) resulting in an updated state and a widget. The first time the loop is performed, the widget must be a home screen, otherwise `widget` is a replacement for the receiver of the most recently processed event. The mobile platform then waits until it received an `event`. The event is delivered to the widget that defined an appropriate handler for it returning the body of the handler which is a new program.

B. Syntax

This section defines the syntax of our language. The complete language that can be used to write mobile applications is defined in V-B1. The complete language is an expression language that evaluates to produce a value; the language of values is defined in section V-B2. Some values are commands as defined in section V-B3. Together, program expressions, values and commands form the basis of the evaluation cycle defined in the previous section. The distinction between programs and commands is required in order to know when it is safe to perform state-updates (since in general the λ-calculus does not enforce an order of execution) and is defined in terms of static types.

`t ::=`	terms
`x`	variables
`k`	constants
`fun(`$\mathtt{x}_i^{i\in[0,n]}$`) t`	functions
`t(`$\mathtt{t}_i^{i\in[0,n]}$`)`	applications
`if t then t else t`	conditionals
`let x = t in t`	locals
$[\mathtt{t}_i^{i\in[0,n]}]$	lists
$\{\mathtt{x}_i = \mathtt{t}_i{}^{i\in[0,n]}\}$	records
`t.x`	field references
`get(t)`	memory access
`loc(t)`	new location
`set(t,t)`	memory update
`{ t | x`$_i \leftarrow$ `t`$_i{}^{i\in[0,n]}$ `}`	command sequence
`widget(t,t)`$\{\mathtt{x}_i = \mathtt{t}_i{}^{i\in[0,n]}\}\{\mathtt{x}_i = \mathtt{t}_i{}^{i\in[0,m]}\}$	widgets

Figure 3. Term syntax

1) General Syntax: The syntax of mobile application programs is shown in Figure 3. The features of the language are divided into: basic λ-calculus; data extensions; commands; widgets. The basic λ-calculus consists of variables, constants, functions and applications. Simple extensions to the calculus include conditionals and local variables. A mobile application needs to represent data structures and these are expressed in terms of lists and records. For convenience we allow definitions in `let`, records, and `widget` to be written `f(x,y)=t` rather than `f=fun(x,y)t`.

Commands are used to access and store values in memory locations. A command can allocate a new memory location and initialise it, can access the value of a memory location and can set it. A command sequence is a special type of term that is used to place an ordering on the evaluation of commands (execution as defined below is otherwise unordered). Suppose that we want to create a function that allocates a two dimensional point and provides an operation to move it:

```
fun()                                                    1
  { { x=loc1,                                            2
      y=loc2,                                            3
      mv=fun(dx,dy)                                      4
        { void |                                         5
          xval <- get(loc1),                             6
          yval <- get(loc2),                             7
          void <- set(loc1,xval+dx),                     8
          void <- set(loc2,yval+dy) } |                  9
    loc1 <- loc(0),                                      10
    loc2 <- loc(0) }                                     11
```

When the function is called it returns a record with three fields: `x, y` and `mv`. The record is allocated by a command sequence in lines 2-11. A command sequence has the form `{ t | bs }` where `t` is a term that constructs the return value of the command sequence and `bs` is a sequence of commands. Each command is performed in turn and produces a value that is bound to a variable, the commands on lines 10 and 11 allocate new memory locations, initialised to 0, and then binds them to the variables `loc1` and `loc2`. The variables are scoped over the value returned by the command sequence (and mutually recursively over each other).

The record returned by the function is defined on lines 2-9. The record has three fields. The first two fields on lines 2 and 3 just associate the field names `x` and `y` with the allocated memory locations. The third field called `mv` is defined on lines 4-9. The move function takes two arguments `dx` and `dy` and will move the allocated point by updating the memory locations. To do this it must access the current contents of the locations, add in the deltas and then update the locations. This is done using a nested command sequence in lines 5-9.

The nested command sequence returns `void` on line 5. The value of `void` is bound by the command sequence but is not important since the commands are performed in order to update the locations, not for their return value. The command sequence accesses the current memory contents (lines 6 and 7) and then updates them (lines 8 and 9). Note that these are performed in sequence.

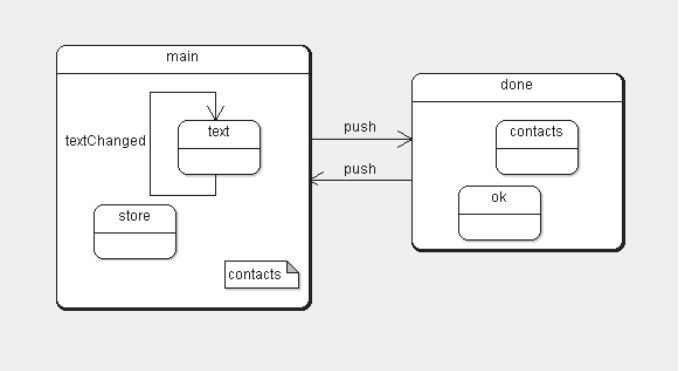

Figure 5. Contacts application state machine

Finally, terms may be widgets. A widget has the form: `widget(ext,id) state handlers` where `ext` is an external widget reference, `id` is an identifier for the widget, `state` is a record of state variables for the widget and `handlers` is a record of functions that implement event handlers for the widget. The external widget is a reference to a library element that determines how to display the widget on the mobile GUI. The external widget places requirements on the state of the widget and handlers that must be implemented. A widget is free to define extra state and handler elements than those defined by its external widget.

For a widget to be correctly formed, each of the handlers must be a function that returns a command. Furthermore, when the command is evaluated it must return a widget. This ensures

```
{ main |                                                                                    1
   main     <- { widget(Screen,0)                                                          2
                 { contacts = cloc;                                                       3
                   contents = table }                                                     4
                 { push() =                                                               5
                    { done([c]+cs) |                                                      6
                       cs   <- get(contacts),                                             7
                       c    <- text.getText(),                                            8
                       void <- set(contacts,[c]+cs)                                       9
                    }                                                                     10
                 } | cloc = loc([])                                                       11
              },                                                                          12
   table    <- { widget(Table,1){ elements = [[text],[store]] }{} | },                    13
   store    <- { widget(Button,2){ label = 'STORE' }{} | }                                14
   text     <- { widget(Text,3)                                                           15
                 { string = sloc }                                                        16
                 { getText() = get(string),                                               17
                   textChanged(t) =                                                       18
                    { text | void <- set(string,t) }                                      19
                 } | sloc <- loc('') },                                                   20
   done     <- { fun(cs)                                                                  21
                  widget(Screen,4)                                                        22
                   { contents =                                                           23
                       widget(Table,5)                                                    24
                         { contents = [[contacts],[ok]] }                                 25
                         {},                                                              26
                     contacts = widget(TextList,6){ lines = cs }{},                       27
                     ok = widget(Button,7){ label = 'OK' }{} }                            28
                   { push() = { main | } } | }                                            29
}                                                                                         30
```

Figure 4. An example application implementing a simple contacts database

that the execution cycle for the language works correctly. These formation rules are enforced by the type system as described below. Finally, a program is correctly formed when it is a command that returns a widget whose external widget is a screen.

Figure 5 shows an example application that implements a simple contacts database. The application has two top-level states: `main` and `done`. The application starts in state `main` and displays a text input field and a button. Each time the text changes in the input field, a `textChanged` event is received. If the button `store` is pushed, then the current database of contacts (labelled `contacts`) is updated and the machine makes a transition to the state labelled `done`. When in the done-state, the application displays the current list of contacts and displays a button labelled `ok`. When the `ok` button is pushed, the application transfers back to the main screen to allow more contacts to be entered.

The program that implements this application is shown in Figure 4. The program consists of a single command sequence that returns the main screen. Each top-level command defines a widget. For example, the main widget (lines 2 - 12) is based on the external widget called `Screen` and with widget id 0.

The main screen widget has two state variables called `contacts` and `contents`. The `contents` variable must be present because the external `Screen` widget requires the contents to be defined as a single sub-widget (in this case a table on line 13). The `contacts` variable is used to store a list of contacts where each contact is a string. Since `contacts` will be updated, it must be a memory address that is allocated by a command which is shown in line 11.

The main screen widget defined a single handler called `push` (lines 5 - 10). A handler is a function. Events occur on external widgets and the owning widget is checked to see if it defines a handler with the appropriate name. In this case the main screen contains a sub-widget called `store` (line 14) that can process a `push` event due to its external widget being of type `Button`. Since the widget does not implement any handlers, the event will be promoted to the most immediately containing parent widget, in this case the table. Event promotion continues until an appropriate handler is found, in this case it will be the definition of `push` in the main screen.

Similar processing occurs when the text is changed in the text input field defined in lines 15 - 20. In this case the `Text` external widget generates a `textChanged` event that is handled locally by the owning widget. When an event is handled, the associated function is called, supplying it with any associated arguments. In the case of changing the text, the `textChanged` handler receives the text in the field as shown on the screen. The body of a handler must be a command that returns a widget. The commands are performed and the returned widget becomes a replacement for the widget that handled the event. In the case of `textChanged` the body of the handler updates the contents of the `string` location. In the case of `push` in the `main` widget, the command sequence retrieves the current list of contacts (line 7), gets the contents of the text field (line 8), updates the current contacts database

(line 9) and calls the function `done` (line 6). The function `done` maps a list of contacts `cs` to a widget that displays the contact strings in a text list and provides a button that returns to the main screen. Like the `main` widget, the `push` event generated by the button on the `done` screen is handled by a function defined by the top-level widget. The body of the handler returns the `main` screen, therefore, pushing the `OK` button returns to that screen as required.

v ::=	values
x	variables
k	constants
fun(${\tt v}_i^{i\in[0,n]}$) t	functions
[${\tt v}_i^{i\in[0,n]}$]	lists
$\{{\tt x}_i = {\tt v}_i{}^{i\in[0,n]}\}$	records
a	addresses
get(v)	memory access
loc(v)	new location
set(v,v)	memory update
{ t | $\mathtt{x}_i \leftarrow \mathtt{v}_i{}^{i\in[0,n]}$ }	command sequence
widget(v,v) $\{\mathtt{x}_i = \mathtt{v}_i{}^{i\in[0,n]}\}\{\mathtt{x}_i = \mathtt{v}_i{}^{i\in[0,m]}\}$	widgets
ext(k)	external widgets

Figure 6. Values

2) Values: Terms have been defined in the previous section. Terms reduce to produce values. The language of values is a sub-language of that of terms plus addresses and external widgets that cannot be directly denoted using terms as defined in Figure 6. An address can only be produced by performing a new location command and then bindings the result in a command sequence. External widgets can only be referenced by name.

c ::=	commands
get(v)	memory access
loc(v)	new location
set(v,v)	memory update
{ t | $\mathtt{x}_i \leftarrow \mathtt{c}_i{}^{i\in[0,n]}$ }	command sequence

Figure 7. Commands

3) Commands: Commands deal with allocating, accessing and updating memory locations. The command sub-language is defined in Figure 7. A command can be thought of as a function that maps state (in implementation terms, a database) to a value and a new state. Some commands do not change the state, but return a useful value, others do not return a useful value but cause a change to the state, and some do both. By modelling commands as functions from states to states and values, we force a program to be organized in such a way as to implement sequences of commands by threading the state through the commands as we shall see in section V-C below.

Programming languages that are used to implement mobile applications often provide blocks containing local variables and commands. Suppose that we want to implement a cell that manages a memory location that can be updated. The function `mkCell` is a command sequence that allocates a memory location `cloc` and returns a record with two fields: `c` (the memory location); `change` (a function that sets the location contents and returns the previous value). The function change implements a command that accesses the contents `v` of the location and sets the value:

```
mkCell(contents) = {
  { c = cloc,
    change(n) = { v | v    <- get(cloc),
                      void <- set(cloc,n) }
  } | cloc <- loc(contents) }
```

The code above is implemented by the following Java class:

```
class Cell {
  Object c;
  change(Object n) {
    Object v = c;
    c = n;
    return v
  }
}
```

C. Types

α,β ::=	types
X	type variables
$\lambda X.\alpha$	type abstractions
$\alpha[\alpha]$	type instantiations
$\alpha+\beta$	type alternatives
bool,int,str	type constants
$\alpha_i^{i\in[0,n]} \rightarrow \mathtt{a}$	function types
$[\alpha]$	list types
$\{\mathtt{x}_i : \alpha_i{}^{i\in[0,n]}\}$	record types
loc(α)	location types
state$\rightarrow$(α,state)	command types
$\omega(\alpha,\alpha)\{\mathtt{x}_i : \alpha_i{}^{i\in[0,n]}\}\{\mathtt{x}_i : \alpha_i{}^{i\in[0,m]}\}$	widget types
$\xi\{\mathtt{x}_i : \alpha_i{}^{i\in[0,n]}\}\{\mathtt{x}_i : \alpha_i{}^{i\in[0,m]}\}$	external types

Figure 8. Types

Evaluation of programs relies on the following properties:

- A mobile program is a command that processes a state which is a local database for the application. A program is therefore a function from states to states.
- A mobile program must define the correct state components for the widgets it displays. For example, a button must have a label and a text field must be associated with a memory address that can be updated with changes to the text as it is typed.
- The GUI of a mobile application is a hierarchically organized tree of widgets. Not all combinations of widgets are legal, for example a table may not contain screens.
- A GUI program must return a widget of type `Screen` at the top-level.
- A mobile program must define handlers for all the events that can occur when the user interacts with the GUI or when the underlying mobile device changes state (orientation, battery charge levels, GPS, etc).
- A mobile application must specify a state transition when an event occurs. The transition must specify a replacement for the widget that receives the event.

Each of the properties listed above should be checked before the program is executed. Most mainstream mobile application languages only support checking a sub-set of these properties. For example, event handlers are usually registered dynamically which means that not all events may have an appropriate handler defined at run-time.

Our language has a static type system that checks all of the properties listed above. All values have types represented using the type language in Figure 8.

Type variables, type abstraction, type instantiation and type alternatives are used to implement parametric polymorphism to allow, for example, functions over lists of several value types. In particular we need to be able to define widgets, for example `Table`, that are generic with respect to their contents.

Type constants, function, list and record types are standard. The memory address containing a value of type α is of type $\texttt{loc}(\alpha)$. A command must process the application's local database. The type of a command is $\texttt{state}\rightarrow(\alpha,\texttt{state})$ where α is the type of the value returned by the command. The type is intended to imply a function from states to states such that command lists must be ordered by combining functions. This construction is exactly how monads are encoded in functional programming languages.

Widgets have a type that encodes the type of their external widget, the type of their identifier, the type of their state variables and the type of their handlers. An external widget has a type that defines the requirements on the state, methods and the events that are produced.

A type relation for the language is defined in Figure 9. The relation has the form: $\Gamma \vdash t : \alpha$ where Γ is a type judgement mapping variables to types, t is a term and α is an associated type. A *command* is a term with the following type:

$$\Gamma \vdash t : \text{state} \rightarrow (\alpha, \text{state})$$

and a program is a command where α is the following type:

$$\omega(\text{Screen}, \text{int})\{\text{contents} = \beta, \ldots\}\{...\}$$

for some appropriate widget type β and associated handlers. Since the types of generated events are encoded in the external widget types and the types of handlers are statically determined in a program, it is possible to statically check that all events have an appropriate handler, i.e. that no event will be lost.

D. Events

An important feature of many programming languages is the ability to catch errors as early as possible. Programming language types are used to prevent incorrect data being supplied to operations. Languages with dynamic typing leave type checking to run-time whereas static typing allows a program to be checked before it is executed.

Our language has a type system as described in 9 that is intended to be statically checked. Although no type checker is presented here, the type relation is relatively standard and therefore we claim that a static type checker for the language is straightforward.

In addition, the language has been designed to allow events that can be raised by widgets to be statically matched against handlers. This feature is unusual amongst languages that support event driven GUIs and this section describes how the checking is performed.

$$\begin{array}{l}
\texttt{events}(\alpha + \beta) = \texttt{events}(\alpha) \cup \texttt{events}(\beta) \\
\texttt{events}(\texttt{bool}) = \emptyset \\
\texttt{events}(\alpha_i^{i\in[0,n]} \rightarrow \alpha) = \emptyset \\
\texttt{events}([\alpha]) = \texttt{events}(\alpha) \\
\texttt{events}(\{x_i : \alpha_i\}) = \bigcup_{i\in[0,n]} \texttt{events}(\alpha_i) \\
\texttt{events}(\texttt{loc}(\alpha)) = \texttt{events}(\alpha) \\
\texttt{events}(\texttt{state}\rightarrow(\alpha, \texttt{state})) = \emptyset \\
\texttt{events}(\omega(\alpha, _)_\{x_j : \beta_j{}^{j\in[0,n]}\}\{x_i : \alpha_i{}^{i\in[0,n]}\}) = \\
\quad (\texttt{events}(\alpha) \cup \bigcup_{j\in[0,n]} \texttt{events}(\beta_j) - \{x_i : \alpha_i{}^{i\in[0,n]}\} \\
\texttt{events}(\xi_\{x_i : \alpha_i{}^{i\in[0,n]}\}) = \{x_i : \alpha_i{}^{i\in[0,n]}\}
\end{array}$$

Figure 10. Widget events

A mobile program is a command that returns a widget of the following type: $\omega(\text{Screen}, \text{int})\{\text{contents} = \beta, \ldots\}\{...\}$ for some widget type β. This type represents a tree of widgets rooted at a screen. The sub-trees and leaves of the tree are widgets that can raise events when the user interacts with them or when the state of the underlying mobile platform changes. The semantics of the language allows the handlers for events to be defined by either the widget that raises the event or a containing parent widget. Therefore, to check that handlers are defined for each event that can be raised by a program, it is necessary to construct the set of outstanding events for each widget in the tree. This is defined in Figure 10 such that a program p:α is well formed when events(α)=$\emptyset$.

VI. Execution

The execution of terms in the language occurs on a hypothetical virtual machine whose states consist of terms, memory states, and a sequence of events. The machine executes by performing a sequence of steps that reduce the term with respect to the memory and the input events. The execution is performed in a particular order so that the expressive properties of the higher-order language are preserved even though an application performs side-effects with respect to the state and event stream. Preservation is important in order that the mobile applications can take advantage of higher-order features including parameterisation over all language features (for patterns, product lines etc) and first class functions (continuations, control abstractions). Execution occurs in the following sequential phases:

1) *reduction* of a term to a command.
2) *performing* a command with respect to a state to produce a widget and an updated state.
3) *replacing* the receiver of the most recent event with the widget.
4) *handling* the event by calling a function that produces a new term.

The first time the sequence is performed, there is no receiver therefore step 3 produces a screen. On subsequent iterations, the term produced in step 4 is used in step 1. The rest of this section describes each of the phases in turn.

$$\text{T-VAR}\ \frac{}{\Gamma[x \mapsto \alpha] \vdash x : \alpha} \qquad \text{T-PAR}\ \frac{\Gamma \vdash t : \lambda X.\alpha}{\Gamma \vdash t : \alpha[X \mapsto \beta]}$$

$$\text{T-TRUE}\ \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \text{T-FALSE}\ \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\text{T-FUN}\ \frac{\Gamma[x_i \mapsto \alpha_i]^{i\in[0,n]} \vdash t : \alpha}{\Gamma \vdash \texttt{fun}(x_i^{i\in[0,n]})t : \alpha_i^{i\in[0,n]} \to \alpha} \qquad \text{T-APP}\ \frac{\Gamma \vdash t : \alpha_i^{i\in[0,n]} \to \alpha \quad \Gamma \vdash t_i : \alpha_i \text{ forall } i \in [0,n]}{\Gamma \vdash t(t_i^{i\in[0,n]}) : \alpha}$$

$$\text{T-IF}\ \frac{\Gamma \vdash t_1 : \texttt{bool} \quad \Gamma \vdash t_2 : \alpha \quad \Gamma \vdash t_3 : \alpha}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : \alpha} \qquad \text{T-LET}\ \frac{\Gamma \vdash t_1 : \alpha \quad \Gamma \vdash t_2[x \mapsto \alpha] : \beta}{\Gamma \vdash \texttt{let } x = t_1 \texttt{ in } t_2 : \beta}$$

$$\text{T-REC}\ \frac{\Gamma \vdash t_i : \alpha_i \text{ forall } i \in [0,n]}{\Gamma \vdash \{x_i = t_i{}^{i\in[0,n]}\} : \{x_i : \alpha_i{}^{i\in[0,n]}\}} \qquad \text{T-REF}\ \frac{\Gamma \vdash t : \{x_i : \alpha_i{}^{i\in[0,n]}\}}{\Gamma \vdash t.x_k : \alpha_k}$$

$$\text{T-LIST}\ \frac{\Gamma \vdash t_i^{i\in[0,n]} : \alpha}{\Gamma \vdash [t_i^{i\in[0,n]}] : [\alpha]} \qquad \text{T-GET}\ \frac{\Gamma \vdash t : \texttt{loc}(\alpha)}{\Gamma \vdash \texttt{get}(t) : \texttt{state} \to (\alpha, \texttt{state})}$$

$$\text{T-LOC}\ \frac{\Gamma \vdash t : \alpha}{\Gamma \vdash \texttt{loc}(t) : \texttt{state} \to (\texttt{loc}(\alpha), \texttt{state})} \qquad \text{T-SET}\ \frac{\Gamma \vdash t_1 : \texttt{loc}(\alpha) \quad \Gamma \vdash t_2 : \alpha}{\Gamma \vdash \texttt{set}(t_1, t_2) : \texttt{state} \to (\alpha, \texttt{state})}$$

$$\text{T-MON}\ \frac{\Gamma[x_i \mapsto \alpha_i] \vdash t_i : \texttt{state} \to (\alpha_i, \texttt{state}) \text{ forall } i \in [0,n] \quad \Gamma[x_i \mapsto \alpha_i{}^{i\in[0,n]}] \vdash t : \alpha}{\Gamma \vdash \{t \mid x_i = t_i{}^{i\in[0,n]}\} : \texttt{state} \to (\alpha, \texttt{state})}$$

$$\text{T-WID}\ \frac{\begin{array}{c}\Gamma \vdash t_1 : \xi\{x_i : \alpha_i{}^{i\in[0,k]}\}r \quad \Gamma \vdash t_2 : \alpha \\ \Gamma \vdash u_i : \alpha_i \text{ forall } i \in [0,n] \quad \Gamma \vdash w_i : \beta_i \text{ forall } i \in [0,m]\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{widget}(t_1, t_2)\{t_i = u_i{}^{i\in[0,n]}\}\{t_i = w_i{}^{i\in[0,m]}\} \\ : \omega(\xi\{x_i : \alpha_i{}^{i\in[0,k]}\}r, \alpha)\{t_i : \alpha_i{}^{i\in[0,k]}, t_i : \alpha_i{}^{i\in[k+1,n]}\}\{t_i : \beta_i{}^{i\in[0,m]}\}\end{array}}$$

$$\text{T-EXT}\ \frac{}{\Gamma[k \mapsto \xi] \vdash \texttt{ext}(k) : \xi} \qquad \text{T-OPT-1}\ \frac{\Gamma \vdash t : \alpha}{\Gamma \vdash t : \alpha + \beta} \qquad \text{T-OPT-2}\ \frac{\Gamma \vdash t : \beta}{\Gamma \vdash t : \alpha + \beta}$$

Figure 9. The type system for the mobile application language

A. Term Reduction

Figure 11 shows the definition of an evaluation relation that reduces a term to a value or *normal form*. The relation is a small-step semantics for the language meaning that its reflexive, transitive closure $\to^*$ defines an execution trace for any given term. Notice that the relation does not impose any unnecessary execution ordering on a composite term.

B. Performing Commands

A command is a particular type of term that acts as a function from states to values and states. A state is a mapping from memory addresses to values and a command may access the contents of an address, update the contents of an address or both. In all cases a command returns a value, but in some cases the value is irrelevant because the command is being used for its side effect on the state. Figure 12 defines a relation: $\Sigma \vdash c \Rightarrow v, \Sigma'$ where Σ is a state before the command c is performed to produce the value v and the resulting state Σ'. There are three atomic commands and a command sequence. The atomic commands are: `get` which accesses a memory location but does not change the state, `loc` which allocates a new memory location and returns it; `set` which updates a memory location (and returns the old value which is usually ignored).

A command sequence has the form `{ t | x1 <- c1, x2 <- c2, ..., xn <- cn }` where each command from `cn` down to `c1` is performed in turn. The results of the commands are bound to the associated variables *in parallel* which means that the bindings are mutually recursive whilst command execution is *in sequence*. The result of the command sequence is the value produced by the body `t` in the context

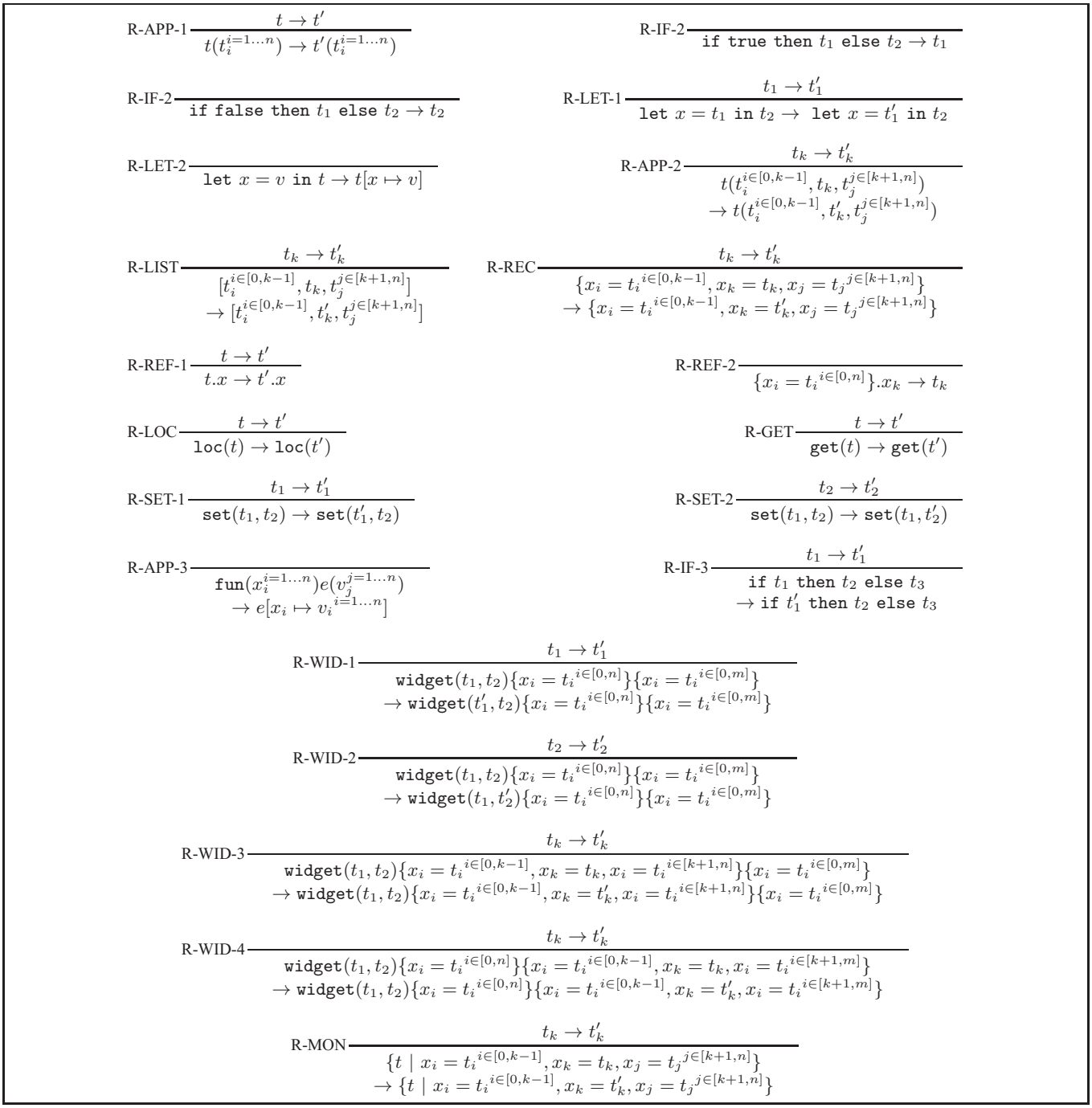

Figure 11. The evaluation rules for the mobile application language

of the variable bindings.

C. Event Handling

An event consists of a name *n*, a widget identifier *i* and some argument values $v_j^{\{j\in[0,n]\}}$. The event occurs with respect to a widget w somewhere on the current screen w_s. The most deeply nested enclosing parent w' of w (where parent is a reflexive transitive relation) that defines a handler named *n* is selected to handle the message:

$i, n \vdash w_s \Rightarrow w'$

where the event handling relation is defined in Figure 13. In order for no event to be lost we need the following proposition to hold:

Proposition 1. *If* $\Gamma \vdash w_s : \alpha$ *and* $\text{events}(\alpha) = \emptyset$ *then* $w' \neq \epsilon$.

The handler $w'.n \to \text{fun}(\text{x}_\text{i}^{\text{i}\in[0,\text{n}]})\text{t}$ is supplied with the argument values and the result of the handler must be a command c as required by the following type constraint on the program:

Proposition 2. *If* $w'.n \to \text{fun}(\text{x}_\text{i}^{\text{i}\in[0,\text{n}]})\text{t}$ *is a handler then*

$$\text{E-WID-1}\ \frac{0 \le k \le n}{i, x_k \vdash \omega(\xi, i)s\{x_i = v_i{}^{i\in[0,n]}\} \Rightarrow \omega(\xi, i)s\{x_i = v_i{}^{i\in[0,n]}\}}$$

$$\text{E-WID-5}\ \frac{\begin{array}{c} i, y_k \vdash v_k \Rightarrow \epsilon \text{ forall } i \in [0,n] \\ x \neq y_j \text{ forall } j \in [0,m] \end{array}}{i, y_k \vdash \omega(\xi, j)\{x_i = v_i{}^{i\in[0,n]}\}\{y_j = w_j{}^{j\in[0,m]}\} \Rightarrow \epsilon}$$

$$\text{E-WID-3}\ \frac{\begin{array}{c} i, x \vdash v_k \Rightarrow v \\ i \neq j, v \neq \epsilon, 0 \le k \le n \end{array}}{i, x \vdash \omega(\xi, j)\{x_i = v_i{}^{i\in[0,n]}\}h \Rightarrow v}$$

$$\text{E-WID-4}\ \frac{\begin{array}{c} i, y_k \vdash v_k \Rightarrow \epsilon \text{ forall } i \in [0,n] \\ 0 \le k \le m \end{array}}{i, y_k \vdash \omega(\xi, j)\{x_i = v_i{}^{i\in[0,n]}\}\{y_j = w_j{}^{j\in[0,m]}\} \Rightarrow \omega(\xi, j)\{x_i = v_i{}^{i\in[0,n]}\}\{y_j = w_j{}^{j\in[0,m]}\}}$$

$$\text{E-WID-2}\ \frac{x \neq x_i \text{ forall } i \in [0,n]}{i, x \vdash \omega(\xi, i)s\{x_i = v_i{}^{i\in[0,n]}\} \Rightarrow \epsilon}$$

$$\text{E-LIST-1}\ \frac{i, n \vdash v_k \Rightarrow \epsilon \text{ forall } i \in [0,n]}{i, n \vdash [v_i^{i\in[0,n]}] \Rightarrow \epsilon}$$

$$\text{E-LIST-2}\ \frac{i, n \vdash v_k \Rightarrow v\ v \neq \epsilon\ \&\ 0 \le k \le n}{i, n \vdash [v_i^{i\in[0,n]}] \Rightarrow v}$$

$$\text{E-REC-1}\ \frac{i, n \vdash v_k \Rightarrow \epsilon \text{ forall } i \in [0,n]}{i, n \vdash \{x_i = v_i{}^{i\in[0,n]}\} \Rightarrow \epsilon}$$

$$\text{E-REC-2}\ \frac{i, n \vdash v_k \Rightarrow v\ v \neq \epsilon\ \&\ 0 \le k \le n}{i, n \vdash \{x_i = v_i{}^{i\in[0,n]}\} \Rightarrow v}$$

$$\text{E-ATOM}\ \frac{\text{isAtom}(v)}{i, n \vdash v \Rightarrow \epsilon}$$

Figure 13. Event handling

$$\text{C-GET}\ \frac{}{\Sigma[\alpha \mapsto v] \vdash \texttt{get}(\alpha) \Rightarrow v, \Sigma}$$

$$\text{C-LOC}\ \frac{}{\Sigma \vdash \texttt{loc}(v) \Rightarrow \alpha, \Sigma[\alpha \mapsto v]\ \texttt{new}(\alpha, \Sigma)}$$

$$\text{C-SET}\ \frac{}{\Sigma[\alpha \mapsto v] \vdash \texttt{set}(\alpha, v') \Rightarrow v, \Sigma[\alpha \mapsto v']}$$

$$\text{C-MON}\ \frac{\begin{array}{c} \Sigma_i \vdash c_i[x_j \mapsto v_j]^{j\in[0,n]} \Rightarrow v_i, \Sigma_{i+1} \text{ forall } i \in [0,n] \\ t[x_j \mapsto v_j]^{j\in[0,n]} \rightarrow^* v \end{array}}{\Sigma_0 \vdash \{t \mid x_i = c_i\}^{i\in[0,n]} \Rightarrow v, \Sigma_{n+1}}$$

Figure 12. Command processing

$\Gamma \vdash w'.n : \alpha_i^{i\in[0,n]} \rightarrow \text{state} \rightarrow (\beta, \text{state})$
Therefore since $t[x_i \mapsto v_i]^{i\in[0,n]} \rightarrow^* c$ *when* c *is performed with respect to the current state* Σ *the result is a widget* v *and a new state* $\Sigma \vdash c \Rightarrow v, \Sigma'$. *Finally, the screen* w_s *is modified by replacing* w' *with* v *to produce a new display:* $w_s[v/w']$ *(where the substitution operation* _[_/_] *is defined on the equality of terms. Therefore we have formally specified the loop defined in section V-A.*

VII. Analysis

A. Use as an Intermediate Language

The language proposed in this article is a tool that can be used to analyse domain specific languages that support mobile applications. It is not intended to directly support mobile applications in its own right, in that sense it is a *mobile application calculus*. The calculus is executable, and therefore can be used as an intermediate language as shown in Figure 14 where the architecture consists of 3 tiers: (1) the application, written and compiled using a DSL; (2) the DSL specific engine and appropriate libraries; (3) the running platform, Java, C#.NET, Android or iOS (iPhone). For each of the target platforms, the engine will comprise of two major parts. Firstly there will the platform libraries (MobLib) that contain the specific platform API calls. This library will contain the callable display, interface, and underlining methods of that platform. Secondly the engine, that will run the compiled code and make the appropriate platform calls using the the bundled platform library set.

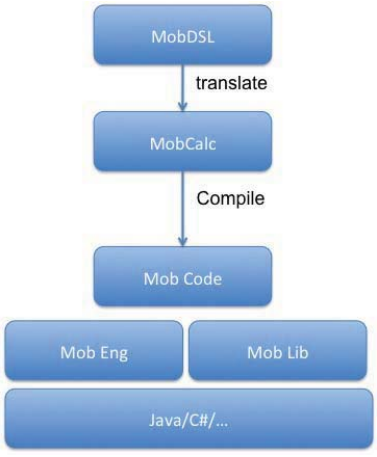

Figure 14. Proposed architecture

Other languages that are used for event-driven systems include the pi-calculus and state-charts. The language presented in this article is different in that it incorporates external widgets into the calculus in a type safe way and includes state via a built-in monad-like mechanism in order that it is a DSL for reactive applications.

B. Implementation Options

Attempting to develop for mobile platforms is a challenging task, and in most cases different approaches come with their advantages, and their disadvantages. With the DSL and the creation of virtual machines on targeted platforms, one particular benefit would be the avoidance of application installation source lock-in, which is applicable to the many users of iPhones/iPads. Through lock-in comes increased security for end-users through the use of application validation by that platform vendor; this can be seen as a method to allow that vendor to decide on the types of applications it believes are right for that user/device.

A VM and downloadable programs (in the form of DSL program definitions), this can be overcome after the user has the VM installed on their device. This requires the VM needs to be downloaded onto the mobile platform; a situation that some vendors take steps to prevent. For example, Sun Microsystems attempted to make available a iPhone version of the Java VM, which includes JavaME, a branch of Java designed for mobile and embedded devices; unfortunately this was blocked. If there was the ability to run a VM on an iPhone/iPad, development of these applications will no longer require a certain platform, as the tools will be operable in multiple desktop platforms. Other advantages of a VM would also include the decrease in application size and faster download times.

An alternative approach to mobile applications involves the use of web-browser technologies such as JavaScript, HTML5 and CCS. These are particularly attractive since an application is portable across many different devices and the introduction of new features in these technologies makes it possible to offer many of the features of native applications. The architecture described in Figure 14 can be used with these technologies in order to offer *abstraction* through a domain specific solution.

C. Current State and Further Work

The calculus language was initially described in [27] and has been prototyped as an interpreter in Java and used to implement a number of simple applications including a simple address book and a mobile platform adjacency notification application. The next step is to develop a mobile Virtual Machine (VM) for platforms including Android and iPhone. Because of the policies in place regarding VM development for the iPhone discussed earlier, the VM may not be accepted by Apple to be on the App-Store. One method of possibly getting around the issues with the App-store policy on VM development, could be incorporating the XMLVM [3] and instead of following a VM approach, use a compilation approach for iPhone/iPad, or targeting JavaScript.

Because of the problems that can occur from dynamic event and event handler association, we hope to implement a type checking system. In iPhone applications, certain UI classes in the UIKit framework can create events, with the developer then can associate and link with a particular event handler. At compile time, these associations are not checked, and can cause an application to hang and crash if and when that event handler does not exist or meet the requirements of the event. A type checker will be needed to detect situations where event handlers are not implemented. The type checker has been implemented and has been shown to work correctly with a number of example applications. A next step is to provide the consistency and completeness of the type system and to integrate it with software engineering tools such as XText on Eclipse.

Other areas of future work include the ability to connect to external services. This will take the form of a method of connecting to RSS/XML feeds including a method for parsing the documents. Features such as external connectivity can be implemented as external widgets that integrate seamlessly with the language presented here.

REFERENCES

[1] H. J. De La Vergne, C. Milanesi, A. Zimmermann, R. Cozza, T. H. Nguyen, A. Gupta, and C. Lu, "Competitive landscape: Mobile Devices, Worldwide, 4Q09 and 2009," Gartner., Stamford, CT, Tech. Rep., Feb. 2010.

[2] P. Miravet, I. Marín, F. Ortín, and A. Rionda, "Dimag: A Framework for Automatic Generation of Mobile Applications for Multiple Platforms," in *Proc. of the 6th International Conference on Mobile Technology, Application & Systems*, 2009, pp. 1–8.

[3] A. Puder, "An XML-based Cross-Language Framework," in *Proc. On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, LNCS vol. 3762, Springer, 2005, pp. 20–21.

[4] A. Puder, "A Code Migration Framework for Ajax Applications," in *Proc. Distributed Applications and Interoperable Systems, 6th IFIP WG 6.1 International Conference*, LNCS vol. 4025, Springer, 2006, pp. 138–151.

[5] A. Puder, "A Cross-Language Framework for Developing Ajax Applications," in *PPPJ '07: Proc. of the 5th International Symposium on Principles and Practice of Programming in Java*, 2007, pp. 105–112.

[6] A. Puder and I. Yoon, "Smartphone Cross-Compilation Framework for Multiplayer Online Games," in *Proc. of the International Conference on Mobile, Hybrid, and On-line Learning*, 2010, pp. 87–92.

[7] H. Behrens, "MDSD for the iPhone: Developing a Domain-Specific Language and IDE Tooling to Produce Real World Applications for Mobile Devices," in *Splash '10, Proc. of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, 2010, pp. 123–128.

[8] F. Balagtas-Fernandez and H. Hussmann, "Evaluation of User-Interfaces for Mobile Application Development Environments," in *Proc. of the 13th International Conference on Human-Computer Interaction. Part I: New Trends*, 2009, pp. 204–213.

[9] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "Links: Web programming Without Tiers," in *Proc. of the 5th International Symposium on Formal Methods for Components and Objects (FMCO)*, 2006, pp. 266–296.

[10] J. R.M. Herndon and V. Berzins, "The Realizable Benefits of a Language Prototyping Language," *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 803–809, 1988.

[11] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler," in *SIGSOFT '94: Proc. of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994, pp. 111–120.

[12] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton, "A Software Engineering Experiment in Software Component Generation," in *ICSE '96: Proc. of the 18th International Conference on Software Engineering*, 1996, pp. 542–552.

[13] J. Gray and G. Karsai, "An Examination of DSLs for Concisely Representing Model Traversals and Transformations," in *HICSS '03: Proc. of the 36th Annual Hawaii International Conference on System Sciences*, 2003.

[14] D. M. Groenewegen, Z. Hemel, L. C. Kats, and E. Visser, "Webdsl: a Domain-Specific Language for Dynamic Web Applications," in *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications*, 2008, pp. 779–780.

[15] Z. Hemel and E. Visser, "Pil: A Platform Independent Language for Retargetable DSLs," in *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, LNCS vol. 5969, Springer, 2009, pp. 224–243.
[16] R. V. Roque, "Openblocks: an Extendable Framework for Graphical Block Programming Systems," M.S. thesis, Dept. of Electrical Engineering and Computer Science., MIT., Massachusetts, USA, 2007.
[17] F. T. Balagtas-Fernandez and H. Hussmann, "Model-Driven Development of Mobile Applications," in *ASE '08: Proc. of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 509–512.
[18] A. Wright, "Ready for a Web OS?," in *Commun. ACM*, vol. 52, no. 12, pp. 16–17, Dec. 2009.
[19] M. Fowler, *Domain Specific Languages*, 1st ed Boston, MA:Addison-Wesley Professional, 2010.
[20] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, December, 2005.
[21] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, "The Fortran Automatic Coding System," in *IRE-AIEE-ACM '57 (Western): Papers presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, 1957, pp. 188–198.
[22] D. D. Chamberlin and R. F. Boyce, "Sequel: A Structured English Query Language," in *SIGFIDET '74: Proc. of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, 1974, pp. 249–264.
[23] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger, "Report on the Algorithmic Language Algol 60," *Commun. ACM*, vol. 3, no. 5, pp. 299–314, 1960.
[24] E. Maximilien, H. Wilkinson, N. Desai, and S. Tai, "A Domain-Specific Language for Web Apis and Services Mashups," in *Service-Oriented Computing – ICSOC 2007*, LNCS vol. 4749, Springer, 2010, pp. 13–26
[25] M. Shtelma, M. Cartsburg, and N. Milanovic, "Executable Domain Specific Language for Message-Based System Integration," in *Model Driven Engineering Languages and Systems* (A. Schürr and B. Selic, eds.), LNCS vol. 5795, Springer, 2009, pp. 622–626.
[26] S. L. Peyton Jones and P. Wadler, "Imperative Functional Programming," in *POPL '93: Proc. of the 20th ACM SIGPLAN SIGACT Symposium on Principles of Programming languages*, 1993, pp. 71–84.
[27] D. Kramer, T. Clark, and S. Oussena, "Mobdsl: A Domain Specific Language for Multiple Mobile Platform Deployment," in *Proc. of the IEEE International Conference on Networked Embedded Systems for Enterprise Applications*, 2010.

Dean Kramer is currently a PhD student in the Centre for Model Driven Software Engineering at The University of West London, UK where he received his BSc Computing and Information Systems in 2008. His research interests include Software Product Lines, Context-Awareness, Mobile Development, MDA, and Domain Specific Languages. Dean has worked on a number of research projects involving the development of mobile applications and social networking.

Tony Clark is a Professor of Informatics and Head of Department for Business Information Systems at Middlesex University, UK. His research interests include System Modelling and MDA, Domain Specific Languages, Language Oriented Programming, Dynamic Modelling, MetaModelling, Software Tools, Formal Methods, Programming Language Design, and Functional Programming. Prof. Clark has been involved in a number of industrial projects including contributing to the UML 2.0 standard and developing commercial tools for model driven development. He has published widely in the fields of software modelling and programming languages.

Samia Oussena is a Reader and the head of the Centre for Model Driven Software Engineering at The University of West London, UK. She manages a number of funded projects for industry and research bodies. She has a research background in methodologies and software application development. Prior to academia, she gained an extensive industrial experience in software development. Samia research interests are in developing software methods for enterprise applications.