Towards a core calculus for implicitly migration-capable applications

Michael Greenberg

University of Pennsylvania

mgree@cis.upenn.edu

Yitzhak Mandelbaum

AT&T Labs - Research

yitzhak@research.att.com

Abstract

Mobile computational devices, like smartphones, tablets and laptops, have become a standard part of the computing landscape. Moreover, many users regularly interact with an assortment of devices, including mobile ones. Therefore, the ability to migrate UI-enabled applications is becoming increasingly important. We describe a design-pattern for applications to simplify support for user-session migration and provide an overview of a lambda calculus for which significant elements of the design pattern can be implemented automatically.

1. Introduction

In a minor scene towards the end of Avatar, a 2009 science-fiction movie, a lab tech is shown transferring a running application between a desktop-style computer and a mobile tablet with a simple swiping gesture. While Avatar is science fiction, this intuitive and appealing feature demonstrated in the movie seems incredibly relevant to current computing realities.

The last decade has seen a veritable explosion in the variety of mobile computing devices. These devices vary in both their computational resources and their interface designs and constraints. Furthermore, many users regularly use more than one device during the course of normal activities. Taken together, the ability of live-migrating applications between devices is becoming increasingly important. For example, a few potentially common use-cases include a user migrating an application from a tethered computer to a mobile device, to avoid interruption when leaving the office; a user migrating a session to the mobile device of a friend in physical proximity, to allow the friend to continue interaction where the user left-off; a user migrating a session from one mobile device to another, more powerful one, or one with different sensors.

In Section 2, we describe some specific applications to which these use-cases could apply. In some cases, developers have already implemented such functionality in an application-specific way. For example, productivity-software suites, like Microsoft's Office and Apple's iWork, have versions for multiple platforms and "migrate" sessions via save and restore of documents. Apple even partly automates this process by pushing and pulling open documents between devices pre-registered to a given user [2].

However, these capabilities are only possible because developers have invested significant effort in designing document formats for their applications, and all the requisite documentation and code to support these formats. Moreover, even with all this effort, arbitrary decisions demarcate state worth saving from discarded state, including session state often integral to the user's experience.

On the surface, this problem closely resembles process checkpointing and migration, an area that has been extensively studied in the literature. Yet, the combination of a number of crucial elements place this challenge beyond the reach of solutions from that area:

Portability The solution must support migration between disparate hardware and software platforms, potentially including (radically) different UI modalities.

Session preservation The solution must preserve the user's session to present the illusion of the application seamlessly migrating between platforms.

Compactness Migration should happen quickly, even in bandwidth-constrained environments. Therefore, care must be taken to reduce the application footprint.

In Section 4, we will discuss related work that addresses one or two of these issues. However, to our knowledge, no existing solutions address all three of these concerns simultaneously. Fundamentally, the nature of this challenge is quite new, driven in large part by the new reality of mobile computing devices. Even just ten years ago, none of these concerns would have been particularly germane. Process migration was limited almost exclusively to long-running, data-intensive computations or daemons; when it did concern UI-based applications, it occurred on homogenous software and hardware; and, migrations occured over, at worst, a wired WAN. Our setting and aims differ: rather than moving computations in-flight, we are interested in facilitating migration as a way users interact with multiple, heterogeneous devices in a single session.

To address this challenge, we advocate following the model used by productivity suites: develop multiple versions of a given application, and devise a data format that effectively communicates both data and session information between versions, or instances of the same version. While this approach might seem obvious, it differs substantially from typical process migration solutions. It separates the issue of developing aesthetically-pleasing and resource-appropriate application versions from the issue of effectively migrating application session state and data. We leave the former for future work, and focus on the latter.

Specifically, we focus on the task of automating the design and maintenance of comprehensive, compact and portable data formats for applications. The benefits of such automation include:

- accelerating and reducing the cost of augmenting applications with save/restore and migration support, even between different versions and platforms;

- protecting session and data integrity by increasing reliability of save/restore functionality;

- reducing bandwidth requirements of backup and migration.

To accomplish these goals, we propose a method to

1. infer from program source code a complete yet compact accounting of essential session state and corresponding routines for translating a live session to/from essential state;

2. design a portable and compact encoding that can share saved sessions among different versions of application.

We sketch our method in the context of small calculus that models functional programming languages.

In the next section, we discuss in more detail examples of application migration and frameworks which we think would benefit from our approach. Then, in Section 3, we give an overview of our current work towards solving the problem we've outlined. We discuss related work in Section 4.

2. Examples and Applications

Web and cloud applications

Our first example application that benefits from migration is a chat client. Without migration capabilities, a mobile user is either forced to run all chat communications from a mobile device, or interrupt conversations each time she switches locations. With migration, a single conversation can continue seamlessly between office, mobile and home devices. Developers of the Trillian chat client recently announced that the next version of Trillian would have capabilities similar to what we've described [16].

Many web applications would benefit in a similar manner. For example, consider using one's desktop to find recommendations and directions to a set of local restaurants—and then wanting to leave with that information without taking your desktop. Media applications like document readers, games, movies and music all stand to gain similarly from the ability to seamlessly continue media consumption between different devices.

While these examples motivate the need for migration, we also wanted to motivate the need for automating the process of adding migration features to an application. To this end, we manually analyzed two small web applications: a countdown timer, which we wrote ourselves and a lunar lander game which we found from a third party.

Our simple timer application completely implements save and restore. To use the timer, the user first enters a number of seconds in a textbox and sets the timer. Then start, stop, and reset buttons run the timer using `getTimeout` to set a callback to be notified once every second. Every time the callback is called, the count of elapsed seconds is increased and the display updates with the remaining time in the countdown; the textbox remains unaltered. In the session state, we must record whether or not there is a timeout active, as well as the total countdown time, the amount of time elapsed so far, and the contents of the textbox. To support migration, we added a "move" button, which saves the current timer state into the URL bar. We save session state by writing it into the fragment identifier at the end of the URL. The user can restore the state later by navigating to the same URL via copy/paste, a bookmark, or simply reloading the page at the new URL. When the page loads, it checks the fragment identifier, restoring any saved session state it finds there—reinstalling `getTimeout` callbacks, for example.

We also studied an application we didn't write: a lunar lander game, JSLander [3]. The game is written in HTML and JavaScript as a demonstration of the `canvas` element. Players of the game control a small lander's angle and thrust; the goal is to safely land on one of the flat spots of the rocky moon terrain while using limited fuel. Players are scored based on how quickly and gently they can land and how little fuel they can use. JSLander has far more session state than our simple timer: the shape of the terrain, the location of stars in the sky, the amount of time elapsed, and the ship's state (fuel, velocity, current thrust, angle)— not to mention timer and user input callbacks. What's more, JSLander keeps this state distributed throughout the system, managing code and data together in a single object. We didn't completely implement session save and restore, precisely due to the difficulty of reconstructing the object graph.

Our experience with these applications demonstrated that adding (or attempting to add) migration capabilities to even small applications presents a significant challenge to developers. The essential difficulties are correctly identifying all the state that needs to be saved and restored and the tedium of implementing save and restore routines.

Frameworks

On the flip side of existing ad hoc approaches to application migration, there are an increasing number of frameworks to support migration at various levels of granularity. These frameworks resolve the systems issues in play, but do nothing to help programmers manage what must be saved and what can be later regenerated. We believe this is an opportunity for our work: we can help programmers use these systems.

At the coarsest level, many modern operating systems offer a "suspend" feature, which records the current process states and slows or stops the processor. However, suspension is very coarse and cannot be used to close individual applications nor preserve applications between reboots of the operating system. Recent extensions to Apple's OS X Lion and iOS allow applications a more structured way to handle suspend and resume events through an explicit suspend/resume API, thereby overcoming the standard limitations. However, they both place the burden of (correctly) using the API entirely on the programmer.

Higher level approaches have attempted to support web and cloud applications. Google's Deep Shot allows a smartphone to photograph a web application and resume the application on another computer based on the photo. But, it critically relies on all application state being encoded in a URI. The task of reifying application session state in the URI is left entirely to the web-app developer [6]. Apple's new iCloud service can actively synchoronize applications running on different devices [1]. Yet, as with suspend/resume, exploiting this functionality of iCloud requires application developers to manually adapt their application to use the iCloud API.

Based on these existing approaches, we believe developers need help managing how their applications save and restore sessions. The next section elaborates on our initial attempts toward automated save and restore.

3. Solution

Our solution begins with the insight that program state can be conceptually classified as either *essential* or *derived*, where the latter is functionally dependent on the former. For example, suppose a program takes a list of numbers and inserts the contents into a binary search tree, in order to present an easily editable, sortable list to the user. The list is the essential data, while the various nodes and connections of the binary tree are the derived data. Or, as a more complex example, consider the relationship between a spreadsheet document and its representation in memory once loaded by a spread-

sheet application. The document is the essential data, and the in-memory represenation is a mix of both essential and derived.

The important thing to note about both examples is that the "essential" data provides two benefits over the complete program data. First, the quantity of essential data is significantly smaller than the derived data. In our simple example above, the memory occupied by the nodes will be at least three times the amount used by the essential data alone (assuming equal-sized integers and pointers). For spreadsheets, the ratio of application memory use to a saved document is usually orders of magnitude. Second, the essential data is easier to understand. Eliminating the derived data abstracts away implementation details, thereby clarifying the remaining state.

These qualities suggest a promising approach to building migratable applications: distinguish essential data from derived data in program state, implement functions to extract essential data (save) and recreate derived data from essential data (restore), and document an external representation (i.e. file format) for essential data. While, for any application, the classification of its state will likely be at least somewhat subjective – that is, there may be many, equally valid, decompositions, we expect this distinction to generally provide the two benefits from our examples: compactness and clarity.

Unfortunately, manually performing these tasks can be difficult, tedious and error-prone. Therefore, we are devising a language to support

1. (programmer-assisted) inference of program state classification;

2. generation of routines to dynamically extract essential state on save and re-compute derived state on restore;

3. generation of a formal data format for externally representing essential data.

Regarding the last goal, we think it is important to format the saved state in a way that is both understandable to humans and "flat", without functional values. Keeping the format human readable will not only help programmers maintain and debug the programs we analyze, but also to develop programs that interoperate with saved states. To achieve this goal, we can leverage the program source code to infer meaningful names, using variable, field, and type names from the source code. Keeping the format flat will aid interoperability between platforms and languages, avoiding issues of what it means to move code. Flat formats are part of what distinguish our approach from checkpointing.

Technical overview

We perform our work in the context of a core calculus: the lambda calculus extended with unit, sums, products, and named values (explained below). Putting names aside for the moment, the operational rules are an entirely standard call-by-value semantics; we believe that we could have just as easily used call-by-need or call-by-name semantics.

Given a term to analyze, our first step is to discover functional dependencies within, and between, the data structures comprising program state. In the context of our calculus, program state is exactly the value returned by the functional expression under analysis. That is, given a program e, such that $e \longrightarrow^* v$, then the value v is the program state. We aim to analyze e to determine the essential parts of v. We specify a provenance-tracking type system for our calculus that exposes the (static) relationships between components of an expression's result value. Next, we provide a mapping from provenance-types to simple types that detects derived components and replaces them with the unit type. We call this the type *mask*, and it exposes only the essential elements of the data. Then, we

$$
\begin{array}{llll}
p & ::= & \mathsf{unit} \mid t_1 \times t_2 \mid t_1 + t_2 \mid t_1 \to t_2 & \text{pre-types} \\
s & \subseteq & 2^{\mathcal{N}} & \text{provenance sets} \\
m & ::= & n \mid \mathsf{anon} & \text{optional names} \\
t & ::= & (s; m; p) & \text{types}
\end{array}
$$

Figure 1. Types

transform this type to a data-format description by employing type isomorphisms to drop unit-fields and then using a standard mapping from sum-of-products data structures to a data format. Correspondingly, we can derive functions to map from the original data structures to the final versions—the save function.

The final step in the process is to derive functions for recreating a data structure from its essence—the restore function. This portion of the process is still work-in-progress. We hope to employ techniques from program slicing research, but do not examine this issue further here. In the remainder of this section, we discuss the other steps in more detail.

We've designed our analysis as a provenance type system, basing our approach loosely on Cheney et al.'s dependency analysis [8]. In their type system, types have two parts: a raw type describing the structure of values in that type, and a provenance set describing the dependencies of values in that type (really, a conservative approximation thereof). The provenance set is just a set of names, drawn from a fixed set of names $\mathcal{N}$. Cheney et al. work with first-order functions, assigning a unique name n_i to each input v_i—values typed with provenance set $s = \{n_{i_1}, n_{i_2}, \ldots\}$ depend on no other inputs than v_{i_1}, v_{i_2}, etc.

Our type system is similarly structured, but with some critical differences, because we do not share the same goal. They want to know the provenance of output data with respect to input data, in which case intermediate values are anonymous and entirely irrelevant. In contrast, we want to know whether a given datum can be recomputed based solely on other available data, in which case an *available* intermediate value – that is, one that is present elsewhere in the result – is of critical importance. For example, if an expression generates a pair of values and the second component of the pair is generated solely from the first component, then we would like to treat them differently (save the first, recompute the second) even though they have the same provenance.

Therefore, our type system goes beyond Cheney et al. by adding in a notion of explicit naming. Operationally, names are irrelevant, entirely ignored by the operational semantics: $\mathsf{name}\ e \longrightarrow e$. Names mark intermediate results as potentially essential and of relevance to our provenance analysis: in the type system, the term $\mathsf{name}\ e$ assigns a fresh name to e and prevents e's provenance set from propogating to other values.

Our types, defined in Figure 1, comprise three parts: a provenance set, an optional name, and a pre-type. Our latest development supports universal and existential quantification over everything—names, provenance sets, types, and pre-types. We omit these extensions here, for space and simplicity.

Given a typing derivation of a term at a non-functional type and a set s of essential names, we calculate a *mask* on program state to find the essential program state: those values with names in s and those values that cannot be recomputed solely from s, i.e., have provenance sets including names outside of s. Where does the set of essential names come from? Cheney et al. assign arbitrary, unique names to query inputs, deeming exactly the input as essential. Our $\mathsf{name}\ e$ construct allows us to consider intermediate results, since the inputs may not be available in the final result. To find the essential names, we can walk over the type and collect the names in its subparts. These named values form a kernel of the essential

state; we can mask away a value that depends completely on named values.

Named values also allow programmers to control which data is considered essential. While we hope for our techniques to be automatic as possible, application-specific concerns mean that allowing programmers to mark essential state is critical.

We have not yet proved anything about our system, though there are a few properties we expect to hold. Phrased as a functional calculus, we expect that immediately restoring after a save produces a contextually equivalent value. We hope to extend our lambda calculus to an event-driven reactive system. In that setting, we would expect to have the original program bisimulate a saved-and-restored copy, i.e., given the same inputs, both versions would behave exactly the same. Our most ambitious setting is migration between heterogeneous systems (e.g., from a computer to a smart phone). Since the two platforms will be running differing code, we will need some application-specific relation between traces to compare a trace started on one host and restored on another. We think we can take some inspiration from prior work on platform dependence [12].

4. Related work

We break up our discussion of related work into two (distinct) groups: work which addresses similar goals and work which uses similar techniques, but to other ends. In the former group is the huge body of literature in process checkpointing and migration. In the latter group is work on provenance, information flow, and program slicing.

Libckpt [13] and Condor [9] are two classic works on process checkpointing and migration. Both differ substantially from our work in that they deal with the entire program image, both code and data, which severely limits both portability and compactness.

The Tui system addresses the portability limitations by adding the ability to translate between instruction-set architectures upon migration [15]. MobJeX [14] takes a similar approach in Java. However, neither of these works address the issue of user-interface portability, nor compactness.

More recent work on virtual machine migration, such as The Collective [5] and Internet Suspend/Resume [10], tackles the similar, if more general, goal of migrating an entire operating system and its running processes between physical machines. However, with respect to our goals, these projects suffer from similar drawbacks to process migration work.

Other works focus on the problem of migrating a process mid-computation, particularly for mobile agents. These works all resort to some version of saving continuations. We consider our work to be orthogonal—we are not interested in how to preserve control flow mid-computation, but how to compactly and portably preserve data between computations. Also, the various solutions from this space rely on homogeneous program versions—a constraint we seek to avoid.

The PLT Racket web server saves web application session states in URIs, mapping string nonces to stored continuations on the server. This approach amounts to a framework for automatically generating the URI's that Deep Shot uses [11]. Their approach isn't migration per se, since the application's state always resides on the server and is never actually transferred.

Closest to our work is that of Chanchio and Sun [4]. They identify data structures used in a high-level program (e.g. written in C) and present a method for encoding and decoding these data structures so that the data may be migrated between source-code identical instances of a program compiled for heterogenous platforms. Their techniques seem directly applicable for automating the transfer of essential data and specifying its format; however, they leave it to the programmer to differentiate essential from inessential data.

Provenance, information flow, and progam slicing make use of related techniques, and our work builds on a number of results in these areas. For space reasons we cannot go into detail, but our type system is inspired by the work of Cheney et al. [8] and similar to the recent work on implicit self-adjusting computation [7]. We expect that prior work on program slicing will be critical in deriving the extract and restore functions from provenance analysis results.

References

[1] Apple Inc. iCloud - Your Content. On all your devices, . URL http://www.apple.com/icloud.

[2] Apple Inc. Documents in the Cloud, . URL http://www.apple.com/icloud/features/documents.html.

[3] J. Brown. JSLander. URL http://www.somethinghitme.com/projects/jslander/.

[4] K. Chanchio and X. Sun. Data collection and restoration for heterogeneous process migration. *Softw. Pract. Exper.*, 2002.

[5] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The collective: a cache-based system management architecture. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI '05, pages 259–272, Berkeley, CA, USA, 2005. USENIX Association.

[6] T.-H. Chang and Y. Li. Deep shot: a framework for migrating tasks across devices using mobile phone cameras. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 2163–2172, New York, NY, USA, 2011. ACM.

[7] Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 129–141, New York, NY, USA, 2011. ACM.

[8] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. In *Proceedings of the 11th international conference on Database programming languages*, DBPL'07, pages 138–152, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] Condor Team. *Condor Version 6.4.7 Manual*, chapter Section 4.2: Condor's Checkpoint Mechanism.

[10] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, WMCSA '02, Callicoon, NY, June 2002. IEEE.

[11] S. Krishnamurthi, P. W. Hopkins, J. Mccarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the plt scheme web server. *Higher Order Symbol. Comput.*, 20:431–460, December 2007.

[12] M. Nita, D. Grossman, and C. Chambers. A theory of platform-dependent low-level software. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 209–220, New York, NY, USA, 2008. ACM.

[13] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[14] C. Ryan and C. Westhorpe. Application adaptation through transparent and portable object mobility in java. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1262–1284. Springer Berlin / Heidelberg, 2004.

[15] P. Smith and N. C. Hutchinson. Heterogeneous process migration: the tui system. *Softw. Pract. Exper.*, 28:611–639, May 1998.

[16] Trillian. The continuous client! URL http://blog.ceruleanstudios.com/?p=1949.