Internationalization: software, universality and otherness

Adrian Mackenzie

a.mackenzie@lancaster.ac.uk

October 2006

Enumerated entities are historical objects. (Verran, 2001)

The questions of 'otherness' or the Other is rarely posed in relation to software as such. This is because universality figures so large in software. Software makes historically and materially specific claims to actual universality (think of Java's "Write one, run anywhere" promise). This tends to push questions of otherness in software aside. Software, by virtue of the notions of universality attached to numbering systems (decimal or binary), to computation (Universal Turing Machine) and to global technoculture itself, seems virulently universal. When figures of otherness appear around software, they tend to be pathological. Pathological software forms such as viruses, worms, trojan horse or even bugs are one facet of otherness marked in software. Much of the architecture and design, as well as much everyday work, pivots on security measures meant to regulate the entry and presence of these others, and at the same time to permit software to translate smoothly between institutional, political, linguistic and economic contexts.

'greetings', 'inquiry', 'farewell': technical universality

Within the design and architecture of much contemporary software, different strategies of coping with otherness have developed. In the software industry, one of the main strategies for figuring others is a process known as 'internationalization' or 'i18n' (for the 18 letters between i and n in 'internationalization'). Techniques of internationalization allow software to be readily adapted to different local conventions, customs and languages. Take an industry standard programming language of the late 1990s, Java (a cup of coffee, but also the main island of Indonesia), a product of Sun Microsystems Corporation. As a programming language and software platform, Java's claims to technical universality include cross-platform execution, numerous network programming constructs and code portability. As Sun's Java documentation states,

> Internationalization is the process of designing software so that it can be adapted (localized) to various languages and regions easily, cost-effectively, and in particular without engineering changes to the software. Localization is performed by simply adding locale-specific components, such as translated text, data describing locale-specific behavior, fonts, and input methods.
>
> Java internationalization, http://java.sun.com/j2se/corejava/intl/index.jsp

'Internationalized' Java software makes use of classes from the java.util package to separate universal components from local components. Local components may have linguistic, symbolic, cultural and geographic specificities. In the tutorial on Sun's Java Tutorial site, the following code demonstrates this elementary separation:

```
import java.util.*;

public class I18NSample {

    static public void main(String[] args) {

        String language;
        String country;

        if (args.length != 2) {
            language = new String("en");
            country = new String("US");
        } else {
            language = new String(args[0]);
            country = new String(args[1]);
        }

        Locale currentLocale;
        ResourceBundle messages;

        currentLocale = new Locale(language, country);

        messages = ResourceBundle.getBundle("MessagesBundle",
                                            currentLocale);
        System.out.println(messages.getString("greetings"));
        System.out.println(messages.getString("inquiry"));
        System.out.println(messages.getString("farewell"));
    }
}
```

{Sun Microsystems, 006 #21}

This sample code declares variables that hold values for 'language', and 'country' and it invokes classes (bundles of methods, functions and data) that represent combinations of language and country, Locale. A Locale is used to choose appropriate resources from the ResourceBundle, a collection of language specific property files distributed with the program. For instance, a German resource bundle might contain the following entries:

greetings = Hallo.

farewell = Tschüß.

inquiry = Wie geht's?

Java supports a standard set of locales that correlate with well-developed, affluent countries (see http://java.sun.com/j2se/1.5.0/docs/guide/intl/locale.doc.html). Not only messages, writing systems and symbols such as currency displayed to users, but more basic algorithmic processes such as counting, searching and sorting often need to be internationalized. For instance, dates are formatted differently in different locales, and need to be sorted according to their format. The concept of the locale points to another key aspect of internationalization. As software is distributed globally, it has to take into account where and when it is running. Time zones form key parts of the infrastructural relations that situate software geographically. Most software needs to be able to represent where and when it is running. Time zones form part of the cross-hatched texture of actions in other spaces and times articulated in software.

Additionally, practices of sorting (a key consideration in any software) shift radically between writing systems. For instance, sorting alphabetically, a straightforward task in European writing systems, cannot be taken for granted in Asian writing systems. In Java, all text characters are encoded in Unicode, a character set that represents all characters in all written languages by unique numbers (in fact, Unicode itself constitutes a primary component of present day software internationalization processes; it merits discussion in its own right; see). In the character series for European languages, the order of Unicode characters corresponds to alphabetical order. This is not guaranteed for all languages. Sorting strings in non-European languages requires different techniques. Assumptions about order, sequence and sorting go to the heart of the design of software.

Interestingly, the closer one moves to the core of the Java programming environment, the more restricted the set of supported locales becomes. For instance, whereas Java graphic user interface components display messages in roughly a dozen different languages, the messages displayed by the Java Software Development Kit (the bundle of tools used to develop Java software) only display messages from two locales, English and Japanese.

Software for “human beings”: fictitious universality

Technically universal yet abstractly local, commercial internationalization focuses on consumption and use of software, not its distribution or production. Wider distribution may be the purpose of internationalization, but the nature of distribution and production itself does not change through techniques of internationalization, no matter how thoroughly carried through into different aspects of software. Yet, distribution is a, perhaps the, key issue in software today because changes in the nature of distribution of software change what can be done with and through software. Software is becoming social. *Ubuntu*, 'Linux for Human Beings', a project supported heavily by Mark Shuttleworth, a South African entrepreneur , is a Linux/GNU distribution in which internationalization of distribution itself figures centrally as part of the project. *Ubuntu* represents a politically progressive open source or FLOSS alternative to commercial strategies of internationalization represented by Sun's Java or various equivalents found in the Microsoft's .NET, etc. The *Ubuntu* Manifesto states that:

> software should be available free of charge, that software tools should be usable by people in their local language, and that people should have the freedom to customize and alter their software in whatever way they need .

Whereas the techniques of internationalization are concerned with the cost-effective entry of products into different markets, the *Ubuntu* distribution makes use of the 'very best in *translations and accessibility infrastructure* that the Free Software community has to offer, to make *Ubuntu* usable for as many people as possible'. The 'translation and accessibility infrastructure' that the manifesto has in mind are none other than the Rosetta (' Rosetta is a Web-based system for translating open source software into any language') and LaunchPad a 'collection of services' built by Shuttleworth . These software services coordinate the localisation of software by allowing volunteers and other participates to supply the translation of menu items, dialogs and other text-based elements of the user interface and help files. The distribution of *Ubuntu* is predicated partly on the redistribution of the work of translating to cohorts of volunteer translators who are explicitly assured that '*Ubuntu will always be free of charge*' (System->About Ubuntu).

Like i18n, *Ubuntu* also assumes a great deal about the universal relevance of its code. This is a point that Soenhke Zehle has recently highlighted. Code is produced for *Ubuntu* (and many other software projects) in technically advanced contexts in Europe, North America, India or East Asia and then localised for execution in less developed countries by volunteers (who themselves may or may not be local). *Ubuntu* introduces a multinational dimension to the internationalization of software, but the software itself remains universal in its aims and expectations because code and software itself is presumed to be universal as a text and as a practice. In this respect, no matter how distributed it's production might become, and how many eyes and hands contribute to it, there is no other figured in software because software itself now garners universality from that other universal, “human beings”, free individuals who are normalized in important ways. Despite the reorganisation of distribution and production to include collective modes of localization, and the corresponding overcoming of institutional, national and economic discrimination against certain ethnic groups, the code itself makes assumptions about computing platforms, network infrastructures, information environments and people that may not be universally relevant.

Tropically relevant code and ideal universality

Could i18n be done differently? This question touches on political struggles over the value of universals that have been at the heart of much theoretical debate in the last decade. It is difficult to articulate any viable alternative to technical universality (software that 'runs anywhere', as Java claims) or to fictitious universality (Ubuntu's software for human beings) because universality itself is a deeply ambiguous concept . To highlight this ambiguity, I want to point to some of the underpinnings of all software: reliance on practices of numbering, enumerating and sorting.

In Volume 1 of *The Art of Computer Programming,* Donald Knuth writes: '[c]omputer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values; they involve important *structural relationships* between the data elements' . The keys terms are already highlighted by Knuth. Software never deals with amorphous masses of value, but structural relationships. The properties of these relationships, and the value accorded to different relations are not universal. They exist in particular places, histories and contexts. The panoply of data structures, algorithms, database designs, protocols, and network topologies developed by programmers over the last 50 years attest to the singularity of these relationships. Software concatenates every single value, no matter how trivial, in relationships that are essentially social, communicative, and corporeal or living.

These relationships afford some kinds of universality and not others. To understand this, we need only turn to recent anthropological studies of mathematics. Ethnomathematics is motivated by the problem of universality, and in particular, how to make sense of different ways of dealing with unity and plurality without bogging down in relativism. It offer leads on how we might being to think about universality more concretely, and thereby begin to radicalize software internationalization. Such analysis points to forms of universality that ultimately put into question existing figures of consumer, user or human. In *Science and an African Logic,* Helen Verran writes: 'numbers are located in the embodied doing of rituals with hands, eyes, and words, but if this is so, how is it that they seem to have the capacity to be definitive even in the absence of any bodily doings?' . Her answer to this question is highly germane to software. It pivots on the idea that certain practices transform written forms of numerals (Knuth's 'numerical values') into numbers (Knuth's 'structural relationships'): '[e]numeration "transforms" all numerals to numbered bodies by the very precise operation of interpellating, and likewise transforms nonenumerated bodies to enumerated' (103). That is, numerals are elements in a writing system, but numbers are things that marshal, order and define bodies in the most general sense. The translation from inscribed numeral to embodied number occurs through practices of *enumeration* that are lived, singular and specific.

For instance, the Yoruba numbering practices described by Verran are multi-base (base 5, base 10 and base 20). This affords highly flexible and rapid mental calculation far surpassing what can be done in base 10 mental calculation that appeared in European cultures sometime around 1300 . This implications of this go far: Yoruba numbers are different to European numbers in the way they deal with unity and plurality. Rather than projecting outwards in long series or sets of numbers as European practices of enumeration tend to, they incorporate inwards, in numbers nested in each other . That is, numbers are generated by differing forms of number-naming that themselves stem from different bodily and linguistic practices. Distinctions between hands and feet, left and right figure directly in Yoruba multi-base numbering, whereas ten fingers 'are treated as a set of homogeneous elements taken as linearly related' (66).

In a less radical difference, programming languages could be analysed in terms of their enumeration strategies and the ways they generate unities and pluralities. Lisp differs from Python by virtue of the emphasis it puts on recursion as a way of enumerating, but recursion is sometimes difficult to invoke. Python or Java makes enumeration a readily available function, invoked countless times by programmers and programs. For instance, the elementary Dictionary datatype in Python defines one-to-one relationships between keys and values that allow mental operations of ordering to be merged with physical operations. Most of the fundamental data structures learned by programmers permit entities to be numbered in some way. Tables, lists, queues, arrays and trees all offer ways of enumerating, as well as sorting, ordering, searching and accessing. It is easy to forget that these structural relationships also interpellate bodies as subjects, citizens, inhabitants, patients, users, clients, workers, events, others, things, parts, animals, organisms, stock, sets, lives, etc. The very same construction and manipulation that transforms numerals (graphic forms) into numbers (things in relations of plurality), constitute bodies in structural relationships. Interpellation is one way of theorising the ritual hailing that brings bodies of all kinds into forms of subjecthood in relation to number. This singularising effect is deeply embedded in the graphical writing systems on which software so heavily draws. The very existence of a numeral zero has intense cultural specificity. It need only by invented in numbering systems that ill-afford mental calculation such as the base 10 systems Western cultures have long used ('[Z]ero seems to emerge with the pressures of the graphic recording of a clumsy calculating system' 64; on this point see).

Enumeration has specificities that relate to rituals of interpellation embedded in language, gesture and writing. This point has quite deep implications for what software does, and how others are designated and predicated in software. If these rituals differ between times and places (Verran discusses Yoruba tallying and counting practices in detail), then relations of unity and plurality differ. The general logic constantly re-enacted in elementary software constructs defined at the level of programming languages and at the level of software architectures make particular ways of enumeration (and sorting, searching, etc) continue to work. Although enumeration practices are usually 'naturalised' (that is, taken for granted as obvious), making particular enumerations work is political: it concerns how people belong together.'In any practical going-on with numbers,' writes Verran, 'what matters is that they can be *made* to work, and *making them work* is a politics. Yet is a politics that completely evades conventional foundationist analysis' (88). The universality that might be at stake here could be called 'ideal' in this sense: 'being always already beyond any simple or "absolute" unity, therefore a source of conflicts forever' .

Problems of actual internationalization

In analysing how software moves from technical to fictitious to ideal universality, internationalization becomes increasingly problematic. The figuring of otherness becomes steadily more deeply embodied. In i18n, the local adaptations of technical universality weave software into the techno-economic realities of globalization. More recent alterations in distribution and certain aspects of production opened up social software through notions of fictitious universality change some of the actors involved and begin to change the way software moves globally yet at the cost of requiring individuals to normal, that is, human beings. However in ideal universality, the construct that animates internationalization is transindividual by nature. That is, it questions the given and seemingly natural rules that constitute software as a convoluted set of practices of tallying, numbering, sorting and searching. This questioning directly concerns embodiment, power and language. It is not easy to point to any practical instance of this questioning. The notion of an ideal universality of software might however frame the problem of software internationalization at a different level.

References

Balibar, E. (1995). "Ambiguous Universality." differences : a Journal Of Feminist Cultural Studies **7**(1): 48-74.

Rossum, G. v. (2006). 2.3.8 Mapping Types -- classdict, Python Library Reference. 2**006**.

Rotman, B. (1987). Signifying Nothing: the Semiotics of Zero. Stanford, Stanford University Press.

Shuttleworth, M. (2006). Rosetta. **2006**.

Shuttleworth, M. (2006). The LauchPad Home page. 2**006**.

Sun Microsystems (006). After Internationalization, The Java Tutorial. **68888006**.

Unicode (2006). What is unicode? x. **2006**.

Van Dijk, J. (2005). The Network society : second edition. Thousand Oaks, CA, Sage Publications.

Verran, H. (2001). Science and An African Logic. Chicago, London, The University of Chicago Press.

Zehle, S. (2005). "FLOSS Redux: Notes on African Software Politics." Mute **06**