Acute and TCP: specifying and developing abstractions for global computation

James Leifer[*] Michael Norrish†
Peter Sewell[‡] Keith Wansbrough[‡]

[*]INRIA [†]NICTA [‡]University of Cambridge

[*]http://pauillac.inria.fr/~leifer/
[†]http://web.rsise.anu.edu.au/~michaeln/
[‡]http://www.cl.cam.ac.uk/users/{pes20,kw217}

February 13, 2004

Abstract

This paper describes ongoing work to establish semantic foundations for real-world distributed computation. We are designing and implementing Acute, an expressive and safe programming language (based on an OCaml core) with features for managing abstraction-safe marshalling, dynamic rebinding, and versioning. These permit user-level communication libraries to be written simply as type-safe Acute code, above TCP/UDP sockets.

Further, we are precisely characterizing the properties of TCP and UDP network communication; of particular interest is the semantics of communication failures. Together with the Acute language semantics this constitutes a *mathematically precise* and *executable* system for writing distributed communication libraries, P2P algorithms, etc.

Programming Language: Acute

Distributed computation is ever more widespread and critical, but most programming language design has focussed on problems of local computation, leaving both the *means* of interoperation between programs and the *contents* of messages to the programmer to arrange. Historically, the means have been via system calls or communication libraries; the contents have been in byte-sequence or XML wire formats. We see several key issues:

Firstly, we must deal not just with distributed execution, but with distributed development and deployment, taking place across many administrative domains on a long timescale. It is therefore impossible to synchronise software updates: there may be many coexisting versions of many applications, sharing some libraries but not others, that need to interoperate.

Secondly, we must deal with partial failure, attack, and, sometimes, mobility of code, of devices, and of running computations. This leads to a great variety of communication and persistent store abstractions, with varying performance, security, and robustness properties, used for interoperation in different circumstances. Accordingly, we believe a distributed programming language should indeed not have a built-in commitment to any particular *means* of interaction. Rather, it should be at a level of abstraction that makes distribution and communication explicit, but be expressive enough to enable different abstractions to be provided as library modules.

Thirdly, in constrast to the previous point, providing no language support for the *contents* of exchanged data is unsatisfactory, and discards the benefits of a high-level language. Standardised wire formats have an important place, but it is increasingly useful to be able to exchange arbitrary language values, preserving their structure, without requiring the programmer to translate back and forth. Moreover, distributed programming and debugging is particularly challenging; interaction should be guaranteed to be type-safe, with errors detected as early as possible.

Many existing languages do provide some form of *marshalling* or *serialization*, with a dynamic check at unmarshal-time. However, the constructs are often insufficiently expressive and not fully integrated with the rest of the language, leaving the programmer unable to control what is shipped, what is checked dynamically when unmarshalling, and how the shipped contents are linked in to the receiver. In particular, none deal satisfactorily with *type abstraction*, *rebinding* to local resources, and *version change.*

Type abstraction is an important tool for modular development, preventing accidental dependencies on a module's implementation details. Values of an abstract type can only be operated on by its implementation, which will typically preserve some user invariants. Within a single program this is checked by conventional static typing, but what if values are marshalled and unmarshalled elsewhere? Simply ensuring that marshalling is type-safe is not enough: it should also be *abstraction-safe* by default, respecting abstraction boundaries (and hence preserving user invariants) across a distributed system. When versions change, though, some controlled forms of abstraction-breaking may be needed.

Static binding is desirable for most programming, but when we marshal a value it may have to be dynamically *rebound* to location-dependent resources, and also to local versions of modules – though sometimes we may want to ship the entire code-base it depends on. Thus we require expressive language constructs to specify what to ship and what to rebind. There may be version constraints that the value demands of its new context, and version declarations of the resources the receiver offers.

This argument leads us to consider the design of languages with marshalling of arbitrary language values to byte strings (above which various communication library modules can be expressed), type- and abstraction-safe unmarshalling, and a coherent notion of type equality between separately-developed programs that share some or no libraries, potentially of different versions.

In earlier work we studied two aspects of the problem in isolation. The paper [BHS$^+$03] discussed rebinding marshalled values to local resources, in a simply-typed lambda calculus setting. The paper [LPSW03] treated marshalling of values with abstract types, but without rebinding, using module *hashes* to construct globally-meaningful type names. Neither dealt with version constraints beyond type equality.

Contribution and Approach Our current work, described in more detail in [LSW04], focusses on the integration of rebinding and abstract types. Versions and version constraints are clearly also needed, and included; surprisingly they turn out to play a key role in the integration. We work towards a realistic programming language rather than on purely theoretical calculi. Our contribution takes two forms. On the one hand, we discuss many aspects of the (large!) design space for such languages. On the other, we have designed and implemented a particular language, Acute, in which to exhibit a coherent set of design choices.

Acute is not intended as a full proposal for a production language, but as a vehicle for experimentation and a starting point for debate. It is sufficiently expressive (and has enough library support) to allow non-trivial examples to be written, but yet is still reasonably simple. We observe that some design choices turn out to be forced. Others are more adhoc, but are chosen to support a good range of examples.

The core language of Acute is based on ML. We aim towards primitives that would be pragmatically useful, and would be reasonably easy to integrate with either SML or OCaml. Much of the design discussion is applicable also to lower-level languages, e.g. for dynamic linking, kernel modules etc. in the setting of typed intermediate languages or assembly language.

We are primarily concerned with what the user source language should be, and so work with a direct semantics and implementation, eschewing syntactic sugar and encodings. In detail, we:

- support rebinding to local resources, lifting the *lambda-mark* semantics of [BHSSW03] to a module language and using *redex-time* semantics between modules;
- add a simple language of *resolve methods* to describe how to obtain dynamically-required libraries;
- add languages of module versions and version constraints;
- extend the treatment in [LPSW03], of marshalling in the presence of abstract and manifest types, to modules with multiple type fields;
- support controlled abstraction-breaking type coercions, making the `with!` of [LPSW03] precise and integrating it with multiple-field modules; and, most significantly,
- integrate rebinding and the treatment of abstract types, using *import hashes* to define a notion of type equality that makes sense globally, in the presence of version change and of approximate version constraints on imports.

We have developed a full semantic definition for Acute, using the *coloured brackets* of Grossman et al to preserve abstraction during computation, and have implemented it.

The core language of Acute consists of normal ML types and expressions: booleans, integers, strings, tuples, lists, options, recursive functions, pattern matching, references, exceptions, and invocations of OS primitives in standard libraries. To avoid syntax debate we fix on one in use, that of OCaml.

Our module language supports top-level declarations of structures, containing value term fields and type fields, with both abstract and manifest types in signatures.

The main new constructs are as follows. We give just the syntax here, for concreteness; the paper [LSW04] is devoted to the design rationale and semantics. Expressions are extended with marshalling and unmarshalling; a program is a sequence of module definitions followed by an expression; module definitions are either of marks, definitions of modules, or imports of module identifiers:

```
e ::=  ...
 | marshal MK e:T | unmarshal e as T
definition ::= ...
 | mark MK
 | module M:Sig version vne = Str  withspec
 | import M:Sig version vce  likespec
       by resolvespec = Mo
```

Here *Sig* and *Str* are ML-style module interfaces and bodies, with type and term fields. Marks `MK` name boundaries in module definition contexts at which dynamic rebinding might occur, and imports constrain and specify how rebinding happens. Version numbers and constraints *vne* and *vce* interact with types and with the other `withspec` and `likespec` components.

Network Semantics: TCP/UDP

In contrast to the *pre-hoc* semantic design work of Acute, our work on *network semantics*, with Bishop and Fairbairn, is developing rigorous *post-hoc* behavioural specifications of the ubiquitous UDP and TCP protocols and sockets API, to supplement the existing informal and partial RFCs and texts. The specifications cover interactions across the sockets interface and on the wire; to be useful they must be (and are) closely based on the *de facto* standard — the deployed implementations — rather than the idealisations common in the theoretical literature. They precisely characterise the semantics of partial failure that is visible across the sockets interface, and necessarily deal also with many other details, e.g. of congestion control. The HOL proof assistant is used to sanity-check the large higher-order logic definitions and as a basis for automated symbolic model-checking, to validate the specification by comparing it with traces captured from running implementations (BSD, Linux, and Windows XP). Work on UDP has been completed [WNSS02, NSW02]; a TCP specification is also complete and its validation is in progress.

Work is underway, by Michael Compton, to integrate the UDP specification with a semantics of an Acute/OCaml fragment (using the Isabelle proof assistant). This should provide a proof-of-concept example of reasoning about an exemplar *fully specified* and *executable* distributed system.

Acknowledgements We acknowledge support from a Royal Society University Research Fellowship (Sewell), a St Catharine's College Heller Research Fellowship (Wansbrough), EPSRC grants GRN24872 and GRL62290, APPSEM, and EC FET-GC project IST-2001-33234 PEPITO.

1. REFERENCES

[BHS$^+$03] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In *Proc. ICFP*, pages 99–110, 2003.

[LPSW03] James Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. ICFP*, pages 87–98, 2003.

[LSW04] James Leifer, Peter Sewell, and Keith Wansbrough. Marshalling: Abstraction, rebinding, and version control, 2004. draft, available `http://www.cl.cam.ac.uk/users/pes20`.

[NSW02] Michael Norrish, Peter Sewell, and Keith Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proc. 10th ACM SIGOPS European Workshop*, pages 49–53, September 2002.

[WNSS02] Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proc. ESOP 2002, LNCS 2305*, pages 278–294, April 2002.