



PEPITO
IST-2001-33234
PEer-to-Peer Implementation and TheOry

Deliverable no: D3.12

Requirements, type systems, and semantics for peer-to-peer versioning support

REPORT VERSION: first

REPORT PREPARATION DATE: 2005.2.28

CLASSIFICATION: Public

DELIVERABLE NO: D3.12 DUE DATE: Month 38 DELIVERY DATE: Month 38

PROJECT START DATE: 2002.01.01 PROJECT DURATION: 36+4 months

RESPONSIBLE PARTNER: UCAM

PARTICIPATING PARTNERS: INRIA, UCAM

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge
UK



**Project funded by the European Community under the 'In-
formation Society Technologies' Programme (1998–2002)**

Project Number: IST-2001-33234

Project Acronym: PEPITO

Title: PEer-to-Peer Implementation and TheOry

Deliverable No: D3.12

Requirements, type systems, and semantics for peer-to-peer
versioning support

Due date: project month 38

Delivery date: 2005.2.28

Responsible Partner: UCAM

Participating Partners: INRIA, UCAM

9th March 2005

Editor: Peter Sewell

Authors: James Leifer, Peter Sewell.

Contents

1	Introduction	3
2	Summary of main achievements and relation to other parts of PEPITO	3
3	Detailed description of Obverse: coexisting multiple object versions	6
3.1	Summary	6
3.2	Technical ideas	7
3.3	Relation to the state of the art and future directions	8
4	Detailed description of Dynamic Rebinding for Marshalling and Update with Destruct-time λ	9
4.1	Overview	9
4.2	Technical details	9
4.3	Comparison with the state of the art	12
4.3.1	Lambda Calculi	12
4.3.2	Dynamic Rebinding and λ_{marsh}	12
4.3.3	Dynamic Update	14
5	Detailed description of Proteus: hot update of code versions	16
5.1	Summary	16
5.2	Technical details	16
5.3	Comparison with the state of the art	18
6	Detailed description of Acute: high-level language design for safe, distributed communication	19
6.1	Overview	19
6.2	Versions and version constraints	21
6.3	Comparison with the state of the art	23

1 Introduction

In the Technical Annex, we wrote:

As connectivity changes, there will inevitably be multiple coexisting versions of software components, both of peer-to-peer application programs and of the underlying distributed services. To construct reliable systems therefore requires precise notions of when two components are sufficiently consistent to interact or when it is possible for them to negotiate a common interface.

This deliverable's goal is to address fundamental language design problems that arise from peer-to-peer computing, in which the *construction of systems is carried out in decentralised ways* and for which the parts of these systems *evolve over time* and yet need to be able to *communicate amongst themselves*. To this end, we have identified and successfully given solutions to the following key questions:

coexisting multiple object versions: A program may be forced to load later versions of libraries in order to adequately manipulate data arriving from distributed sources over time. What typing disciplines do we need to follow to manage having multiple versions of the same library? If the libraries are objects, what are the interactions between object inheritance and versioning?

hot update of code versions: Updating parts of a program with new versions of code as the program runs is a difficult operation, but one that is necessary when it is impossible to take the system down and restart it. How can we cleanly specify what should be updated and when? How can the language ensure the type safety of such updates?

safe communication between peers with differing versions: When peers engage in communication of complex language-level values and the peers have potentially differing versions of libraries, how can we ensure the type safety of such communication? How can we express how much of the (transitive closure of) dependency graph of modules should be shipped with a value that depends on them, and how much should be rebound at the receiver's end? When is it safe, or desirable, to allow rebinding to libraries of *differing versions* from those on the sender's side?

In the following sections, we summarise and then present in details the main technical contribution of our work in these areas and compare our approach to the state of the art.

2 Summary of main achievements and relation to other parts of PEPITO

We now list the main technical achievements of this deliverable, show where they have been diffused, and relating them to other parts of PEPITO.

Obverse: coexisting multiple object versions

- ▷ Matthew Parkinson, Peter Sewell, and Gareth Stoye. Versioning for objects. Technical draft. 2005. Available from <http://www.cl.cam.ac.uk/users/pes20/PEPITO/obverse.pdf>.

We have designed a calculus for analysing the problem of managing *multiple versions* of the same library or object that have been dynamically loaded into a single run-time. In addressing this problem, we have studied the disciplines and expressivity required to extend smoothly Java-like languages with support for multiple versions of objects. We have presented a series of solutions that statically, and at link-time, ensure the absence of run-time errors.

Relation to other work in PEPITO: The techniques designed here have a direct bearing on Java-like object oriented languages and could be integrated in the future into EPFL's Scala language, which is a part of WP3.

Acute: High-level programming language design for distributed computation

- ▷ J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP, 2003*. Available from <http://www.cl.cam.ac.uk/users/pes20/acute/>.
- ▷ Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation: design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, October 2004. Also published as INRIA RR-5329. 193pp. Available from <http://www.cl.cam.ac.uk/users/pes20/acute/>.
- ▷ Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Source release of the Acute system, January 2005. Available from <http://www.cl.cam.ac.uk/users/pes20/acute/distro/acute-0.50-1-src.tar.gz>.

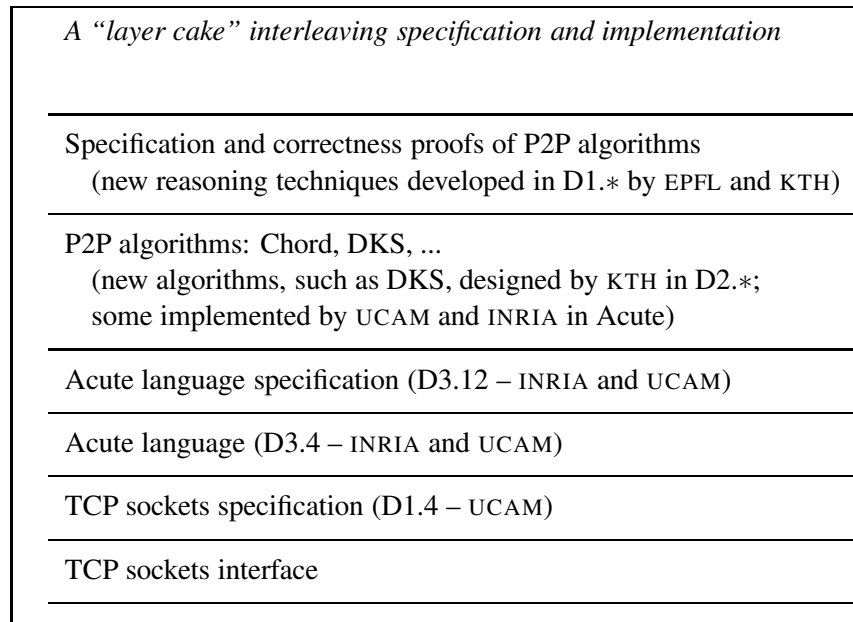
We have studied key issues for distributed programming in high-level languages, defining a fully formal semantics and producing a prototype implementation, for an experimental language, Acute.

Acute extends an OCaml core to support distributed development, deployment, and execution, allowing type-safe interaction between separately-built programs. It is expressive enough to enable a wide variety of distributed infrastructure layers to be written as simple library code above the byte-string network and persistent store APIs, disentangling the language runtime from communication.

This requires a synthesis of novel and existing features: (1) type-safe marshalling of values between programs; (2) dynamic loading and controlled rebinding to local resources; (3) modules and abstract types with abstraction boundaries that are respected by interaction; (4) global names, generated either freshly or based on module hashes: at the type level, as runtime names for abstract types; and at the term level, as channel names and other interaction handles; (5) versions and version constraints, integrated with type identity; (6) local concurrency and thread thunkification; and (7) second-order polymorphism with a namespace construct. We deal with the interplay among these features and the core, and develop a semantic definition that tracks abstraction boundaries, global names, and hashes throughout compilation and execution, but which still admits an efficient implementation strategy.

Relation to other work in PEPITO: Our vision, thus, has been to build a full language that extends the powerful safety and expressiveness properties of the ML dialects with advanced features necessary for P2P computing.

Consistent with its ambitious nature and with the *Annex's* description of work to be carried out, only parts of this vision have been realized during PEPITO's lifetime. Nonetheless, one can see in the following diagram how this vision cuts across the research partners and deliverables in PEPITO:



The diagram shows the interleaving layers of implementation and specification work — rising from the lowest level, namely Unix socket-based TCP communication, to the highest, namely the behavior of P2P algorithms.

At the bottom, the specification work formalizes the properties of TCP network communication; of particular interest is the characterization of communication failures that can occur. Above, the Acute interpreter (though in future work this will become a separate compiler and run-time) implements an expressive and safe language that provides features for managing abstraction-safe marshalling, dynamic binding, and versioning; at the same time it exposes the underlying TCP sockets interface, giving user-level communication libraries the flexibility to handle low-level network failure. The Acute language itself is formally specified by an operational semantics which, in concert with the specification of TCP, constitutes a *mathematically precise and executable* system for writing communication libraries, P2P algorithms, etc. As an example of the latter, we reported in D1.2 that we have implemented well-known P2P algorithms in our Acute language above the Acute Sockets API, which is essentially identical to that of the low-level failure semantics of Task 1.1. These were:

1. An implementation of key parts of the classic *Chord* P2P algorithm.
2. An Acute library providing primitives for inter-site asynchronous channel-based communication and migration of running computations between sites, providing essentially the capabilities of the Nomadic Pict programming language but as a concise library.
3. An Acute library providing primitives based on the *Ambient* calculus.

All of these are decentralized implementations, with no central coordinator, and Acute provides guarantees of type safety even for interaction between distinct, separately compiled, and separately executed programs.

Finally, at the top, new proof techniques introduced by PEPITO research may allow formal properties of process calculi idealizations of such algorithms to be verified. Ultimately, in future work, we would like to be able to verify the actual Acute code of, for example, a DKS implementation *directly* in terms of

Acute's semantics.

Hot update

- ▷ G. Bierman, M. Hicks, P. Sewell, G. Stoye, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ . In *Proc. ICFP*, 2003. Available from <http://www.cl.cam.ac.uk/users/pes20/>
- ▷ Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoye. Formalizing dynamic software updating. In *Proc. 2nd International Workshop on Unanticipated Software Evolution (USE 2003)*, April 2003.
- ▷ Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoye, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ . Technical Report UCAM-CL-TR-568, University of Cambridge Computer Laboratory, February 2004. Available from <http://www.cl.cam.ac.uk/users/pes20/UCAM-CL-TR-568.pdf>.
- ▷ Gareth Stoye, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *Proceedings of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Long Beach)*, January 2005. Available from <http://www.cl.cam.ac.uk/users/pes20/update-popl-2005.pdf>.

We developed core dynamic rebinding mechanisms as extensions to the simply-typed call-by-value λ -calculus. To do so we must first explore refinements of the call-by-value reduction strategy that delay instantiation, to ensure computations make use of the most recent versions of rebound definitions. We introduce *redex-time* and *destruct-time* strategies. The latter forms the basis for a λ_{marsh} calculus that supports dynamic rebinding of marshalled values, while remaining as far as possible statically-typed.

We have designed an extension of λ_{marsh} with concurrency and communication, giving examples showing how wrappers for encapsulating untrusted code can be expressed. Finally, we have given a high-level semantics for dynamic updating can also be based on the destruct-time strategy, defining a λ_{update} calculus with simple primitives to provide type-safe updating of running code.

We went on to apply this work to a calculus for dynamic software updating in C-like languages that is flexible, safe, and predictable. It supports dynamic “hot” updates to functions (even active ones), to named types and to data, allowing on-line evolution to match source-code evolution as we have observed it in practice.

Relation to other work in PEPITO: The evaluation strategy outlined here plays a direct role in the semantics and implementation of Acute (described above). Our method of hot code update could be integrated in the future into EPFL's Scala language, which is a part of WP3.

3 Detailed description of Obverse: coexisting multiple object versions

3.1 Summary

A significant problem that arises as soon as we have dynamic loading of libraries or objects is the management of multiple versions of the same library. In order to address this problem, we study the disciplines and

expressivity required to extend smoothly Java-like languages with support for multiple versions of objects. We present a series of solutions that statically, and at link-time, ensure the absence of run-time errors.

3.2 Technical ideas

Versioning issues in Java arise due to two forms of distribution:

1. distributed programmers; and
2. distributed computation.

(1) refers to different programs, or components, having conflicting or incompatible library requirements. This problem is often called “DLL hell”, or in Java: “JAR Hell”. Consider deploying a web-server using the Java servlet engine, Tomcat. You install two servlets and they both depend on an XML library. Unfortunately they depend on different and incompatible versions of this library. You copy both versions into the relevant directory, but one of the servlets fails as only one version of the library can be loaded. Software is available to detect these issues early e.g. Krysalis-Version, but they do not allow multiple coexisting versions.

The problem worsens when we consider distributed computation, (2). We have multiple sites interacting. Ensuring all the sites have the same version of the code is often impractical. In this world we will have multiple coexisting versions, and we need to ensure the inter-operate correctly. Note: although we do not explicitly model this distributed world, our choice of semantics allows us to consider many of the issues that arise.

The key issue we concentrate on is the use of multiple versions of the same class. Java does not directly support this. We can play tricks using class loaders, but this does not provide static guarantees. Moreover, the programmer can not specify one method that takes version 1 of a class C , and another method that takes version 2. Clearly we need add versions directly to the language.

We therefore have taken a small fragment of Java and extended with version information, to form a language called Obverse. We extend the syntax of Java with explicit version numbers, e.g.

```
x = new C<v>();
```

This means construct an object of class C and version v . These versions allow the programmer to specify which version of a class they require.

Figure 1 gives a road map to the development of Obverse. The first diagram, (a), expresses what currently exists in Java: each class only has a single version. Code that requires two versions of a particular class is not supported. We address this by allowing multiple versions to exist, illustrated in (b). Each version is really a different class with its own name, e.g. $C<1>$ is distinct from $C<2>$. Programmers can specify the versions of a class their methods take as arguments and return. We prove the language is type sound.

So far the different versions are completely distinct, we extend the language to allow the programmer to specify forward and backward compatibility between versions, illustrated in (c). This compatibility allows us to use a different version than initially intended. Rather than providing explicit syntax and semantics for upgrading modules on the fly, we provide a non-deterministic semantics for constructing classes that provides a compatible version of the class. This non-determinism allows us to consider the possible upgrade schemes without additional semantic machinery.

The final extension to Obverse is to consider replacing a class’s parent with a new version, shown in (d).

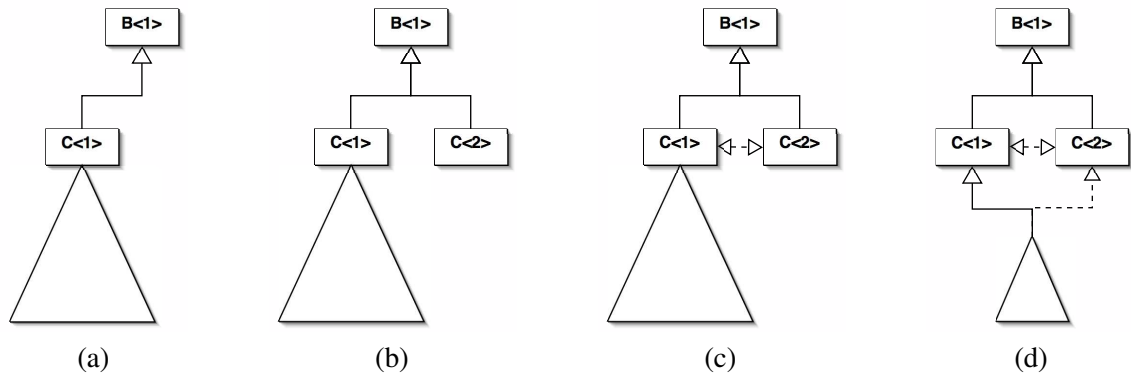


Figure 1: Adding a new version

3.3 Relation to the state of the art and future directions

The .NET architecture has addressed some of the versioning issues by allowing assemblies to contain version information [1]. They allow multiple versions to be stored on a client and let the versioning policy select the correct version. However, it is unclear that this work can deal with the different versions interacting. Their goal is to allow a single application to function correctly, and each application can only require one version of the code. It is unclear how one could use several versions of the same library in one piece of code without having direct language support.

Zenger [2] takes a different approach to the versioning problem. He provides language primitives to allow code to be written in an extensible style. This removes many of the issues of versioning. Our approach, however, requires less changes to the code.

Closely related to versioning is dynamic linking. Dynamic linking allows late updates to code to occur. Drossopoulou, Eisenbach, et al. have study dynamic linking in detail [3]. They provide a detail semantics of when linking errors will occur under changes of class. They play close attention to when different phases of the compilation occur, such as field layout. Our approach remains at a level close to the source code to avoid the problems they highlight.

The version names in the code earlier are programmer supplied. In most cases these versions could be added by the compiler, if only one version of a library existed on the system. However inferring the version if there are multiple versions is harder. The system would require a metric for which version is considered better.

Directly compiling the version name into the code might be too restrictive. One might like to consider a polymorphic versioned bytecode. Polymorphic bytecode of Ancona, Damiani, Drossopoulou and Zucca [4] is an extension to Java bytecode that allows more flexible linking at run-time. They output constraints when they compile a class, so at link-time these constraints can be resolved, and linking can occur against different classes safely. We believe it should be possible to use this work to provide polymorphic version names. Rather than attempting to fill in all the version names statically, we simply construct constraints. These constraints specify that the version must have certain methods and fields. When it comes to link-time the class loader can attempt to solve the constraints. This would allow a great flexibility in which versions can be used.

4 Detailed description of Dynamic Rebinding for Marshalling and Update with Destruct-time λ

4.1 Overview

Most programming languages adopt static binding, but for distributed programming an exclusive reliance on static binding is too restrictive: dynamic binding is required in various guises, for example when a marshalled value is received from the network, containing identifiers that must be rebound to local resources. Typically it is provided only by ad-hoc mechanisms that lack clean semantics.

Here we adopt a foundational approach, developing core dynamic rebinding mechanisms as extensions to the simply-typed call-by-value λ -calculus. To do so we must first explore refinements of the call-by-value reduction strategy that delay instantiation, to ensure computations make use of the most recent versions of rebound definitions. We introduce *redex-time* and *destruct-time* strategies. The latter forms the basis for a λ_{marsh} calculus that supports dynamic rebinding of marshalled values, while remaining as far as possible statically-typed.

We have designed an extension of λ_{marsh} with concurrency and communication, giving examples showing how wrappers for encapsulating untrusted code can be expressed. Finally, we have given a high-level semantics for dynamic updating can also be based on the destruct-time strategy, defining a λ_{update} calculus with simple primitives to provide type-safe updating of running code.

We thereby establish primitives and a common semantic foundation for a variety of real-world dynamic rebinding requirements.

4.2 Technical details

Most programming languages employ *static binding*, with the meaning of identifiers determined by their compile-time context. In general, this gives more comprehensible code than *dynamic binding* alternatives, where the meanings of identifiers depend in some sense on their ‘use-time’ contexts; static binding is also a requirement for conventional static type systems.

Modern software, though, is becoming increasingly dynamic, as it becomes ever more modular, extensible, and distributed. Exclusive use of static binding is too limiting in many ways:

- When values or computations are marshalled from a running system and moved elsewhere, either by network communication or via a persistent store, some of their identifiers may need to be *dynamically rebound*. These may be both ‘external’ identifiers of system-calls or language run-time library functions, and, more interestingly, ‘internal’ identifiers from application libraries which exist in the new context. Such libraries should not be automatically copied with values that use them, both for performance reasons and as they may have location-dependent behaviour (e.g., routing functions).

Moreover, a value may be moved repeatedly, and the set of identifiers to be rebound may change as it moves. For example, it may be desirable to acquire an organisation-specific library that, once resolved, should be fixed and carried with code moved within that organisation.

- Flexible control of dynamic rebinding can support *secure encapsulation* of untrusted code, by allowing access only to sandboxed resources. For example, when loading an untrusted applet, we may bind its `open` identifier to a `safe_open` function that only opens files in the `/tmp` directory. On the other hand, we want the flexibility to link trusted code with the unconstrained `open` function.

- Systems that must provide uninterrupted service (e.g., telephone switches) must be *dynamically updated* to fix bugs and add new functionality – essentially by loading new code into the program and then dynamically rebinding some of the existing identifiers to the new definitions.

While dynamic rebinding is clearly useful in practice, most modern programming languages provide only rather limited and ad-hoc mechanisms. Moreover, no adequate semantic understanding of rebinding currently exists. Our goal in this paper is to identify core mechanisms for dynamic rebinding, as a step towards the design of improved languages for distributed computation.

We are focussing on distributed ML-like languages: with higher-order functions, for expressiveness; with call-by-value (CBV) reduction, for a simple evaluation order (desirable in the presence of either communication effects or dynamic updates); and where possible with static typing, as early detection of errors is particularly important in both distributed and long-running systems.

The motivations for dynamic rebinding arise from distribution, but it turns out that the essential problems come from the interaction between rebinding and sequential computation. We therefore begin with the simply-typed CBV lambda-calculus and develop calculi that support rebinding for marshalling and update. To demonstrate feasibility we sketch an extension of the former with inter-machine communication, and discuss a possible implementation.

We express the semantics of these calculi with direct operational semantics, defining reductions over the calculus syntax. This approach provides clarity, and should scale well to full language designs; it avoids commitment to any particular implementation strategy. We find this preferable to the lower-level alternatives of expressing semantics using abstract machines or encodings (into languages with references), which we believe would lead to rather complex definitions.

Revisiting CBV λ -Calculus Consider the CBV λ -calculus, a model fragment of ML, and in particular the way in which identifiers are instantiated. The usual operational semantics substitutes out binders – the standard *construct-time* (app) and (let) rules

$$\begin{array}{ll} (app) & (\lambda z:T.e)v \quad \longrightarrow \quad \{v/z\}e \\ (let) & \mathbf{let} \ z:T = v \ \mathbf{in} \ e \quad \longrightarrow \quad \{v/z\}e \end{array}$$

instantiate all instances of z as soon as the value v that it has been bound to has been constructed.

This semantics is not compatible with dynamic rebinding, as it loses too much information. To see this, suppose that e in $\mathbf{let} \ z = v \ \mathbf{in} \ e$ transmits a function containing z to some other machine, and we have indicated somehow that z should be dynamically rebound to the local definition when it arrives. With the (let) rule this would be futile, as the z is substituted away before the communication occurs. Similarly, a dynamic update of z after a (let) would be vacuous.

We therefore need a more refined semantics that preserves information about the binding structure of terms, allowing us to delay ‘looking up’ the value associated with an identifier as long as possible so as to obtain the most relevant/recent version of its definition. This should maintain the essentially call-by-value nature of the calculus, however (we elaborate below on exactly what this means).

We present two reduction strategies with delayed instantiation in at the end of this section. The *redex-time* (λ_r) semantics resolves identifiers when in redex position. While this is clean and simple, it is still unnecessarily eager, and so we formulate the *destruct-time* (λ_d) semantics to delay resolving identifiers until their values must be destructed.

Dynamic Rebinding: the λ_{marsh} Calculus With λ_d in place we can consider dynamic rebinding of marshalled values. The key question is this: when a value is moved between scopes, how can the user specify which identifiers should be rebound and which should be fixed? Our answer is embodied in the λ_{marsh} calculus which contains primitives for packaging a value such that some of its identifiers are fixed to bindings in the current context, while others will be rebound when unpackaged in a new scope (e.g., when the value is moved). Which bindings will be fixed is dynamically determined with respect to a *mark*. Marking is done with an expression form

$$e ::= \dots \mid \mathbf{mark} \ M \ \mathbf{in} \ e$$

Here the mark name M is taken from a new syntactic class (not subject to binding); it names the surrounding declaration context. Packaging and unpackaging is done by expressions

$$e ::= \dots \mid \mathbf{grab} \ M \ e \mid \mathbf{ungrab} \ M \ e$$

which are both with respect to a mark. An expression $\mathbf{grab} \ M \ e$ will first reduce e to a value u , and copy all bindings within the nearest enclosing $\mathbf{mark} \ M$; these bindings are essentially static. Identifiers of u not bound within the mark are recorded in a type environment within the packaged value, which has form $\mathbf{grabbed} \ \Gamma \ u$, and can be rebound. For example:

$$\begin{array}{l} \mathbf{let} \ x_1:\text{int} = 5 \ \mathbf{in} \\ \mathbf{mark} \ M \ \mathbf{in} \\ \mathbf{let} \ y_1:\text{int} = 6 \ \mathbf{in} \\ \mathbf{grab} \ M \ (x_1, y_1) \end{array} \longrightarrow \begin{array}{l} \mathbf{let} \ x_1:\text{int} = 5 \ \mathbf{in} \\ \mathbf{mark} \ M \ \mathbf{in} \\ \mathbf{let} \ y_1:\text{int} = 6 \ \mathbf{in} \\ \mathbf{grabbed} \ (x_1:\text{int})(\\ \quad \mathbf{let} \ y_1:\text{int} = 6 \ \mathbf{in} \ (x_1, y_1)) \end{array}$$

Because y_1 is defined within the mark M , its definition is copied into the package, while x_1 is defined outside of M , so it is simply noted in the captured type environment. When this package is unmarshalled using \mathbf{ungrab} with respect to some mark M' , x_1 will be rebound to a definition outside M' , subject to a dynamic type environment check.

To indicate more concretely how λ_{marsh} can form the basis for a distributed programming language that supports mobile code, we sketch an extension with concurrency, communication and external library functions, giving examples showing how wrappers for encapsulating untrusted code can be expressed. We also sketch an implementation strategy.

Dynamic Update: the λ_{update} Calculus Dynamic updating also requires dynamic rebinding and delayed variable instantiation.

We again extend λ_d , here with a simple \mathbf{update} primitive that allows a program variable to be rebound to a new expression. As an example, consider the expression on the left below:

$$\begin{array}{l} \mathbf{let} \ x_1 = 5 \ \mathbf{in} \\ \mathbf{let} \ y_1 = (4, 6) \ \mathbf{in} \\ \mathbf{let} \ z_1 = \mathbf{update} \ \mathbf{in} \\ \pi_1 \ y_1 \end{array} \xrightarrow{\{y \leftarrow (x_1, 6)\}} \begin{array}{l} \mathbf{let} \ x_1 = 5 \ \mathbf{in} \\ \mathbf{let} \ y_1 = (x_1, 6) \ \mathbf{in} \\ \mathbf{let} \ z_1 = () \ \mathbf{in} \\ \pi_1 \ y_1 \end{array}$$

The \mathbf{update} expression indicates that an update is possible at the point during evaluation when \mathbf{update} appears in redex position. At that run-time point the user can supply an update of the form $\{w \leftarrow e\}$, indicating that w should be rebound to expression e . In the example this update is $\{y \leftarrow (x_1, 6)\}$; the let-binder for y_1

is modified accordingly yielding the expression on the right above, and thence a final result of 5. Here any identifier in scope at the update point can be rebound, to an expression that may mention identifiers in scope at its binding point. We define what it means for an update to be well-typed with respect to a program; applying well-typed updates preserves typing.

The use of λ_d enables us to deal simply and cleanly with higher-order functions, largely ignored in past work. We imagine λ_{update} will form the core of future calculi that include other desirable features, such as state transformation, abstract types, changing the types of variables, multi-threading, etc. As a first step, in [5] we develop a model of updating in the style of Erlang [6].

4.3 Comparison with the state of the art

4.3.1 Lambda Calculi

Our approach in λ_r and λ_d of using **lets** to record the arguments of functions has some similarities to prior work on explicit substitutions [7] and on sharing in call-by-need languages [8].

In work on the compilation of extended recursion (particularly for mixin modules) Hirschowitz, Leroy, and Wells have (independently) used a semantics which is similar to λ_d save that (a) the language allows more general recursive definitions, and (b) the semantics collapses multiple **lets** [9, 10]. It draws on work of Ariola and Blom [11] which also collapses **let** blocks. For rebinding, we need to preserve this structure.

There are also similarities with Felleisen and Hieb’s syntactic theory of state [12]. Their Λ_S models late (redex-time) resolution of state variables in a substitution-based system by labelling the substituted-in values with the name of the variable; assignment to a variable triggers a global replacement of all values labelled with that variable throughout the program with the new value. This is then revised to an equivalent store-based model. As in our system, there is a notion of a “final answer”, which may require further clean-up to yield the value that is the result of the computation in the usual calculus (our $\llbracket \cdot \rrbracket$).

4.3.2 Dynamic Rebinding and λ_{marsh}

Dynamic Binding Work on dynamic binding can be roughly classified along three dimensions. First, one can have either *dynamic scoping*, in which variable occurrences are resolved with respect to their dynamic environment, or *static scoping with explicit rebinding*, where variables are resolved with respect to their static environment, but additional primitives allow explicit modification of these environments. Second, one can work either with one class of variables or split into two: one treated statically and one dynamically. Third, for explicit rebinding the variables to be rebound can be specified either individually, per name, or as all those bound by a certain term context. We identify some points in this space below, and refer the reader to the surveys of Moreau and Vivas [13, 14] for further discussion.

Dynamic scoping first appeared in McCarthy’s Lisp 1.0 as a bug, and has survived in most modern Lisp dialects in some form. It is there usually referred to as “dynamic binding.” Lisp 1.0 had one class of variables. MIT Scheme’s [15] `fluid-let` form and Perl’s `local` declaration similarly perform dynamically-scoped rebinding of variables. Modern Lisp distinguishes at declaration time between dynamically and statically scoped variables, as formalised in the λ_d -calculus of Moreau [13]. Lewis et al propose to add syntactically-distinct, dynamically-scoped *implicit parameters* [16] to statically-scoped Haskell. While flexible, dynamic scoping can result in unpredictable behaviour, since variables can be inadvertently captured; this was referred to as the

downward funarg problem in the Lisp community (to avoid this in a typed setting Lewis et al forbid arguments of higher-order functions from using dynamically scoped variables).

Turning to static scoping with explicit rebinding, the *quasi-static scoping* Scheme extension of Lee and Friedman [17] and the λN -calculus of Dami [18] both have two classes of variable with a rebinding primitive that specifies new bindings for individual variables. Jagannathan’s *Rascal* language [19] maintains both a static environment and a *public* environment, corresponding again to two variable classes. The *barrier*, *reify*, and *reflect* operations allow explicit manipulation of the variables bound by an entire term context.

Outside the above classification, MIT Scheme also permits explicit manipulation of *top-level* environments. Hashimoto and Ohori introduce a typed context calculus [20] for expressing first-class evaluation contexts within the lambda calculus. Context holes can be ‘filled in’ with terms having free variables which are captured by the surrounding context. This allows binding at context-application time, but does not support rebinding. It is developed in the *MobileML* language [21]. Garrigue [22] presents a calculus based on streams that can be used to encode dynamic binding for particular, *scope-free* variables.

Locating our λ_{marsh} calculus in this space, it adopts static scoping with explicit rebinding, has a single class of variables, and supports rebinding with respect to named contexts (not of individual variables). Use of the destruct-time strategy delays variable resolution until the last possible moment to give the most useful semantics, e.g., for repeatedly-mobile code. We believe these choices will lead to code that is easier to write and maintain, particularly for large systems.

We conjecture that λ_{marsh} could be encoded in *Rascal*, and also that it could be given semantics either in an environment-passing style or using an abstract machine with concrete environments. We believe, however, that our reduction semantics, with small-step reductions over the source syntax, is more perspicuous.

Partial Continuations The context-marking operator **mark** is reminiscent of Felleisen and Friedman’s [23] prompt operator **#**, and **grab/ungrab** of their control operator \mathcal{F} . Their operators capture partial *continuations*, whereas our operators may be seen as capturing partial *environments*: whereas **mark** marks a *binding* context, **#** marks an *evaluation* context. In fact, λ_{marsh} filters the captured context to retain only the binding structure (E_2), whereas Felleisen et al’s semantics exhibits the behaviour of our λ_c , eagerly substituting out bindings and leaving only the control structure (E_1) to be captured.

Another interesting connection is between abstract continuations [24], as used by Queinnec [25], and the reduction contexts E_3 used in our operational semantics. Each A_1 or A_2 corresponds to a frame of the continuation, except that the semantics of ACPS substitutes the A_2 binding frames away.

Gunter et al [26] have studied **#** and \mathcal{F} in a typed setting. It is interesting to note that although they state a type safety result, this does not exclude the possibility that a well-typed program can get ‘stuck’ if an appropriate prompt does not exist.

In the λ_{marsh} calculus, marks are named (not anonymous), are not bound, and are preserved by marshal/unmarshal operations. Some other choices have been investigated in the context of partial continuations by Moreau and Queinnec [27, 25].

Dynamic Linking Dynamic linking is a ubiquitous simple form of dynamic binding, allowing program bindings to be resolved either at load-time or run-time, rather than statically. Conventional executables will, when run, dynamically link shared libraries for standard library functions (e.g., `read`, `write`, etc.). Which libraries are loaded depends upon the context; for example, a machine might have a library compiled with profiling

enabled and one without. However, once dynamically bound, a variable's definition is fixed, precluding rebinding for marshalling or update. Modern languages often provide an interface to the dynamic linker so that programs can load new code at run-time [28, 29, 30, 31, 6]. Dynamic linking has been formally modelled for low-level machine code [32, 33, 34], and high-level languages like Java [28]. Several authors have considered customised linking for security, performance, or debugging purposes [31, 35, 33, 36].

Rebinding in Distributed Calculi A number of distributed process calculi provide implicit rebinding of names, adopting interaction primitives with meanings that depend on where they are used in a location structure [37, 36, 38, 39, 40, 41]. This allows a form of rebinding to application libraries, but these works do not address the problem of integrating this rebinding with local functional computation.

The JoCaml and Nomadic Pict languages for mobile computation [42, 40] provide rebinding to external functions, but the details are matters of implementation, not semantically specified – though a more principled proposal for JoCaml has been made by Schmitt in a Join-calculus setting [39].

4.3.3 Dynamic Update

There are a number of implemented systems for dynamic updating surveyed in [43], notably including Erlang [6]. There is very little rigorous semantics, however. Duggan [44] has a formal framework for updating types, but updating code is considered only informally, based on arguments around reference types. Gilmore et al [45, 46] have a formal description of updating, but it is centred on abstract types, and is tied to their particular abstract machine. Neither of these systems properly handles updating first-class functions. Gilmore et al require that a function not be *active* when it is updated; closures in activation records are active, and cannot thus be updated. Reference-based indirections require that the types of function arguments change in a way that interacts poorly with polymorphism [43].

Construct-time λ_c

<i>Values</i>	$v ::= n \mid () \mid (v, v') \mid \lambda z: T.e$	
<i>Atomic evaluation contexts</i>	$A ::= (-, e) \mid (v, -) \mid \pi_r _ \mid - e \mid v _ \mid \mathbf{let} \ z = _ \mathbf{in} \ e$	
<i>Evaluation contexts</i>	$E ::= _ \mid E.A$	
<i>(proj)</i>	$\pi_r (v_1, v_2) \longrightarrow v_r$	
<i>(app)</i>	$(\lambda z: T.e)v \longrightarrow \{v/z\}e$	$\frac{e \longrightarrow e'}{E.e \longrightarrow E.e'}$
<i>(let)</i>	$\mathbf{let} \ z = v \mathbf{in} \ e \longrightarrow \{v/z\}e$	
<i>(letrec)</i>	$\mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ e' \longrightarrow \{\lambda x: T.\mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ e/z\}e'$	if $z \neq x$

Redex-time λ_r

<i>Values</i>	$u ::= n \mid () \mid (u, u') \mid \lambda z: T.e \mid \mathbf{let} \ z = u \mathbf{in} \ u' \mid \mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ u$	
<i>Atomic evaluation contexts</i>	$A_1 ::= (-, e) \mid (u, -) \mid \pi_r _ \mid - e \mid u _ \mid \mathbf{let} \ z = _ \mathbf{in} \ e$	
<i>Atomic bind contexts</i>	$A_2 ::= \mathbf{let} \ z = u \mathbf{in} \ _ \mid \mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ _$	
<i>Evaluation contexts</i>	$E_1 ::= _ \mid E_1.A_1$	
<i>Bind contexts</i>	$E_2 ::= _ \mid E_2.A_2$	
<i>Reduction contexts</i>	$E_3 ::= _ \mid E_3.A_1 \mid E_3.A_2$	$\frac{e \longrightarrow e'}{E_3.e \longrightarrow E_3.e'}$
<i>(proj)</i>	$\pi_r (E_2.(u_1, u_2)) \longrightarrow E_2.u_r$	
<i>(app)</i>	$(E_2.(\lambda z: T.e))u \longrightarrow E_2.\mathbf{let} \ z = u \mathbf{in} \ e$	if $\text{fv}(u) \notin \text{hb}(E_2)$
<i>(inst)</i>	$\mathbf{let} \ z = u \mathbf{in} \ E_3.z$	$\longrightarrow \mathbf{let} \ z = u \mathbf{in} \ E_3.u$
	if $z \notin \text{hb}(E_3)$ and $\text{fv}(u) \notin z, \text{hb}(E_3)$	
<i>(instrec)</i>	$\mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ E_3.z$	$\longrightarrow \mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ E_3.\lambda x: T.e$
	if $z \notin \text{hb}(E_3)$ and $\text{fv}(\lambda x: T.e) \notin \text{hb}(E_3)$	

Destruct-time λ_d

<i>Values</i>	$u ::= n \mid () \mid (u, u') \mid \lambda z: T.e \mid \mathbf{let} \ z = u \mathbf{in} \ u' \mid \mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ u \mid z$	
<i>Atomic evaluation contexts</i>	$A_1 ::= (-, e) \mid (u, -) \mid \pi_r _ \mid - e \mid u _ \mid \mathbf{let} \ z = _ \mathbf{in} \ e$	
<i>Atomic bind contexts</i>	$A_2 ::= \mathbf{let} \ z = u \mathbf{in} \ _ \mid \mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ _$	
<i>Evaluation contexts</i>	$E_1 ::= _ \mid E_1.A_1$	
<i>Bind contexts</i>	$E_2 ::= _ \mid E_2.A_2$	
<i>Reduction contexts</i>	$E_3 ::= _ \mid E_3.A_1 \mid E_3.A_2$	
<i>Destruct contexts</i>	$R ::= \pi_r _ \mid - u$	$\frac{e \longrightarrow e'}{E_3.e \longrightarrow E_3.e'}$
<i>(proj)</i>	$\pi_r (E_2.(u_1, u_2)) \longrightarrow E_2.u_r$	
<i>(app)</i>	$(E_2.(\lambda z: T.e))u \longrightarrow E_2.\mathbf{let} \ z = u \mathbf{in} \ e$	if $\text{fv}(u) \notin \text{hb}(E_2)$
<i>(inst - 1)</i>	$\mathbf{let} \ z = u \mathbf{in} \ E_3.R.E_2.z$	$\longrightarrow \mathbf{let} \ z = u \mathbf{in} \ E_3.R.E_2.u$
	if $z \notin \text{hb}(E_3, E_2)$ and $\text{fv}(u) \notin z, \text{hb}(E_3, E_2)$	
<i>(inst - 2)</i>	$R.E_2.\mathbf{let} \ z = u \mathbf{in} \ E'_2.z$	$\longrightarrow R.E_2.\mathbf{let} \ z = u \mathbf{in} \ E'_2.u$
	if $z \notin \text{hb}(E'_2)$ and $\text{fv}(u) \notin z, \text{hb}(E'_2)$	
<i>(instrec - 1)</i>	$\mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ E_3.R.E_2.z$	$\longrightarrow \mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ E_3.R.E_2.\lambda x: T.e$
	if $z \notin \text{hb}(E_3, E_2)$ and $\text{fv}(\lambda x: T.e) \notin \text{hb}(E_3, E_2)$	
<i>(instrec - 2)</i>	$R.E_2.\mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ E'_2.z$	$\longrightarrow R.E_2.\mathbf{letrec} \ z = \lambda x: T.e \mathbf{in} \ E'_2.\lambda x: T.e$
	if $z \notin \text{hb}(E'_2)$ and $\text{fv}(\lambda x: T.e) \notin \text{hb}(E'_2)$	

Three Call-by-Value Lambda Calculi

5 Detailed description of Proteus: hot update of code versions

5.1 Summary

Dynamic software updates can be used to fix bugs or add features to a running program without downtime. Essential for some applications and convenient for others, low-level dynamic updating has been used for many years. Perhaps surprisingly, there is little high-level understanding or language support to help programmers write dynamic updates effectively.

To bridge this gap, we present Proteus, a core calculus for dynamic software updating in C-like languages that is flexible, safe, and predictable. Proteus supports dynamic updates to functions (even active ones), to named types and to data, allowing on-line evolution to match source-code evolution as we have observed it in practice. We ensure updates are type-safe by checking for a property we call “con-freeness” for updated types t at the point of update. This means that non-updated code will not use t *concretely* beyond that point (concrete usages are via explicit coercions) and thus t ’s representation can safely change. We show how con-freeness can be enforced dynamically for a particular program state. We additionally define a novel and efficient static *updateability analysis* to establish con-freeness statically, and can thus automatically infer program points at which all future (well-formed) updates will be type-safe. We have implemented our analysis for C and tested it on several well-known programs.

5.2 Technical details

Dynamic software updating (DSU) is a technique by which a running program can be updated with new code and data without interrupting its execution. DSU is critical for non-stop systems such as air-traffic control systems, financial transaction processors, enterprise applications, and networks, which must provide continuous service but nonetheless be updated to fix bugs and add new features. DSU is also useful for avoiding the need to stop and start a non-critical system (e.g., reboot a personal operating system) every time it must be patched.

Providing general-purpose DSU is particularly challenging because of the competing concerns of *flexibility* and *safety*. On the one hand, the form of dynamic updates should be as unrestricted as possible, since the purpose of DSU is to fix bugs or add features not necessarily anticipated in the initial design. On the other hand, supporting completely arbitrary updates (e.g., binary patches to the existing program) makes reasoning about safety impossible, which is unacceptable for mission-critical software.

Proteus is a general-purpose DSU formalism for C-like languages that carefully balances these two concerns, and adds assurances of predictability. Proteus programs consist of function and data definitions, and definitions of *named types*. In the scope of a named type declaration $t = \tau$ the programmer can use the name t and representation type τ interchangeably but the distinction lets us control updates. Dynamic updates can add new types and new definitions, and also provide replacements for existing ones, with replacement definitions possibly at changed types. Functions can be updated even while they are on the call-stack: the current version will continue (or be returned to), and the new version is activated on the next call. Permitting the update of active functions is important for making programs more available to dynamic updates [47, 48, 49]. We also support updating function pointers. Based on our experience [48] and a preliminary study on the evolution of C programs, we believe Proteus is flexible enough to support a wide variety of dynamic updates.

When updating a named type t from its old representation τ to a new one τ' , the user provides a *type transformer function* c with type $\tau \rightarrow \tau'$. This is used to convert existing t values in the program to the new representation.

To ensure an intuitive semantics, we require that at no time can different parts of the program expect different representations of a type t ; a concept we call *representation consistency*. The alternative would be to allow new and old definitions of a type t be valid simultaneously. Then, we could *copy* values when transforming them, where only new code sees the copies [50, 48]. While this approach would be type safe, old and new code could manipulate different copies of the same data, which is likely to be disastrous in a language with side-effects.

To ensure type safety and representation consistency, we must guarantee the following property: after a dynamic update to some type t , no updated values v' of type t will ever be manipulated *concretely* by code that relies on the old representation. We call this property “con- t -freeness” (or simply “con-freeness” when not referring to a particular type). The fact that we are only concerned about subsequent *concrete* uses is important: if code simply passes data around without relying on its representation, then updating that data poses no problem. Indeed, for our purposes the notion of con-freeness generalizes notions of encapsulation and type abstraction in object-oriented and functional languages. This is because concrete versus abstract uses of data are not confined to a single linguistic construct, like a module or object, but could occur at arbitrary points in the program. Moreover, con-freeness is a flow-sensitive property, since a function might manipulate a t value concretely at its outset, but not for the remainder of its execution.

To enforce con-freeness, Proteus programs are automatically annotated with explicit *type coercions*: $\mathbf{abs}_t e$ converts e to type t (assuming e has the proper type τ), and $\mathbf{con}_t e$ does the reverse at points where t is used concretely. Thus, when some type t is updated, we can dynamically analyze the active program to check for the presence of coercions \mathbf{con}_t , taking into account that subsequent calls to updated functions will always be to their new versions. If any \mathbf{con}_t occurrences are discovered, then the update is rejected.

Unfortunately, the unpredictability of a dynamic con-free check could make it hard to tell whether an update failure is transient or permanent, since the dynamic check is for a particular program state. Rather, we would prefer to reason about update behavior statically, to (among other things) assess whether there are enough update points. Therefore, we have developed a novel *static updateability analysis*. We introduce an **update** expression to label program points at which updates could be applied. For each of these, we estimate those types t for which the program may not be con- t -free. We annotate the **update** with those types, and at run time ensure that any dynamic update at that point does not change them. This is simpler than the con-free dynamic check, and more predictable. In particular, we can automatically infer those points at which the program is con-free for all types t , precluding dynamic failure.

In summary, our contributions consist of the following points:

- ▷ We present Proteus, a simple and flexible calculus for reasoning about representation-consistent dynamic software updating in C-like languages. We have motivated our DSU support in Proteus with a study of the changes over time to some large C programs, taking these as indicative of dynamic updates that we have to support.
- ▷ We formally define the notion of con-freeness, and prove that it is sufficient to establish type safety in updated programs.
- ▷ We have designed a novel updateability analysis that statically infers the types for which a given **update** point is not con-free.

To enable on-line evolution we must support software changes unanticipated during the initial design. The kind of changes that must be supported on-line are, we believe, similar to those that can be observed off-line in the program source tree. Therefore, to motivate our approach to DSU we describe the results of a small study we did on the source code evolution of some long-running services.

Using a custom tool, we compared increasing versions of a few large C programs. These include Linux, version 2.4.17 (Dec. 2001) to 2.4.21 (Jun. 2003); BIND, versions 9.2.1 (May 2002) to 9.2.3 (Oct. 2003); Apache, version 1.2.6 (Feb. 1998) to 1.3.29 (Oct. 2003); and OpenSSH, version 1.2 (Oct. 1999) to 3.8 (Feb. 2004). For these programs, the changes followed a few key trends:

- ▷ The overwhelming majority of a version change consists of added functions, or changes to existing functions which do not involve a changed type signature. Few, if any, functions are deleted. For example, BIND 9.2.2—9.2.3 resulted in 30 new functions added and 890 changed (starting from 3214 total functions). Of the changed functions, only 28 had a change of signature, of which about two-thirds were to add or remove arguments with the rest changing the type of an argument. As another example, Apache 1.3.0–1.3.6 resulted in 51 functions added, 10 deleted, and 290 changed, of which 4 changed their type signature (starting from 836 total functions).
- ▷ Global variables tend to be fairly static, adding a few and deleting a few, but growing over time. For example, OpenSSH grew from 106 to 251 total global variables from version 1.2.2 to 3.7, adding from 3 to 30 new variables per release, deleting up to 10. One change in Apache added 88 variables and deleted 37, but most added or deleted fewer than 10. It is extremely rare for a global variable to change type.
- ▷ Data representations, which is to say *type definitions*, change between versions, though rarely. In C, types are defined with `struct` and `union` declarations (aggregates), `typedefs`, and `enums`. Very often, the changes are to aggregates and involve adding or removing a field. For example, moving from Linux 2.4.20—2.4.21 resulted in 36 changes to `struct` definitions (out of 1214 total), of which 21 were the addition or removal of fields, while the remaining 15 were changes to the types of some fields. Type definitions are rarely deleted.

In short, to match these kinds of changes DSU must readily support the addition of new definitions (functions, data, or types), and the replacement of existing definitions (data or functions) at the same type. It must also allow changes to function types and data representations, but not necessarily to the types of global variables. Proteus supports these kinds of changes.

5.3 Comparison with the state of the art

Dynamic software updating has been used in industry for many years and is well-studied in academia. To our knowledge, approaches taken in industry are often application-specific, or rely on redundant hardware, limiting their applicability. Academic approaches range from being quite flexible but type-unsafe, to type-safe but quite inflexible.

The systems of which we are aware are either less safe or less flexible than our approach. Many systems are either not type safe at all [51, 52, 50, 49], or could admit dynamic type errors [47]. Some systems are type-safe but not representation consistent [48, 53]. For example, Hicks [48] ensures type-safety by copying and transforming values from their old representation to the new; existing code will continue to use the old, stale values unless the programmer manually ensures otherwise. Other systems are too restrictive. For example, updates may only be permitted to individual class instances whose type cannot change [54, 55, 52, 56], or else representation changes are only permitted for abstract types or encapsulated objects [57, 56, 45]. In many cases, updates to active code are disallowed [45, 58, 51, 50, 56], and data stored in local variables may not be transformed [48, 50, 51, 52].

A number of systems use techniques that bear some resemblance to our approach. Dynamic ML [45] supports updating modules defining *abstract* types `t`. Since by definition clients of such a module must use values of

type t abstractly, the module can be updated if none of its functions are on the call-stack (i.e. it is *inactive*). Our use of abs_t and con_t coercions generalizes this idea to non-abstract named types, and permits more fine-grained determination of safe update points. In particular, we could discover points *within* an abstract module at which it could be safely updated. This allows our $\text{conFree}[-]$ check to be more precise than Dynamic ML’s “activeness” check. Dynamic ML has no static notion of proper update timing, as we do with our updatability analysis.

Duggan [53] supports dynamic updates to named types, which use constructs *fold* and *unfold* to create and de-struct values of named type (similar to our abs_t and con_t). However, updated programs are not representation-consistent. Rather, programmers must provide transformer functions that go both ways: from the old to the new representation and from the new version back to the old. Occurrences of *unfold* will dynamically compare the expected version of the t value with its actual version and apply some composition of forward or backward transformers to convert the value. This approach ensures well-formed updates are always well-timed. However, programs are harder to reason about. We might wonder: will the program still behave properly when converting a t value forward for new code, backward for old code, and then forward again? Moreover, it may not always be possible to write backward transformers, since updated types often contain more information than their older versions.

Boyapati et al. [57] and the K42 operating system [56] ensure well-timed updates to objects. Both systems rely on object encapsulation to guarantee that no active code depends on an object’s representation when the object is updated. In Boyapati et al., proper timing is enforced by programmer-defined database-style transactions: if an update occurs at an inopportune time, they abort the current transaction, perform the update, and then restart the transaction. In K42, an object to be updated is made *quiescent* by blocking new threads from using it, and waiting until all current threads that could be using it have terminated. Our approach uses the more general notion of con-freeness, rather than encapsulation. Transactions are approximated by automatically- or programmer-inserted **update** points, but without the benefit of rollback. To mimic this approach in our setting, we could force **update** points to synchronize in different threads; an update could proceed only when all threads have reached safe update points. We intend to flesh out this idea in future work.

While our updatability analysis is new, its general formulation is similar to other capability type systems [59, 60, 61]. For example, capabilities in the Calculus of Capabilities [59] statically prevent a runtime dereference of a dangling pointer by approximating the runtime heap. Our capabilities prevent runtime access to a value whose representation might have changed by approximating the current set of legal types.

6 Detailed description of Acute: high-level language design for safe, distributed communication

6.1 Overview

This work addresses the design of distributed languages. Our focus is on the higher-order, typed, call-by-value programming of the ML tradition: we concentrate on what must be added to ML-like languages to support typed distributed programming. We explore the design space and define and implement a programming language, Acute.

This builds on previous work done by James J. Leifer, Gilles Peskine (INRIA), Peter Sewell, Keith Wansbrough (U. of Cambridge), published in ICFP 2003. The present work extends the theory to contend with the challenge of integrating with an ML like programming language. This requires a synthesis of novel and existing features.

Type-safe marshalling Type-safe marshalling demands a notion of type identity that makes sense across multiple versions of differing programs. For concrete types this is conceptually straightforward, but with abstract types more care is necessary. We generate globally-meaningful type names either by hashing module definitions, taking their dependencies into account; freshly at compile-time; or freshly at run-time. The first two enable different builds or different programs to share abstract type names, by sharing their module source code or object code respectively; the last is needed to protect the invariants of modules with effect-full initialisation.

Dynamic linking and rebinding Dynamic linking and rebinding to local resources in the setting of a language with an ML-like second-class module system raises many questions: of how to specify which resources should be shipped with a marshalled value and which dynamically rebound; what evaluation strategy to use; when rebinding takes effect; and what to rebind to. For Acute we make interim choices, reasonably simple and sufficient to bring out the typing and versioning issues involved in rebinding, which here is at the granularity of module identifiers. A running Acute program consists roughly of a sequence of module definitions (of ML structures), imports of modules with specified signatures, which may or may not be linked, and marks which indicate where rebinding can take effect; together with running processes and a shared store.

Global expression names Globally-meaningful expression-level names are needed for type-safe interaction, e.g. for communication channel names or RPC handles. They can also be constructed as hashes or created fresh at compile time or run time; we show how these support several important idioms. The polytypic support and swap operations of Shinwell, Pitts and Gabbay's FreshOCaml are included to support renaming of local names occurring inside of values during communication.

Versioning In a single-program development process one ensures the executable is built from a coherent set of versions of its modules by controlling static linking often by building from a single source tree. With dynamic linking and rebinding more support is required: we add versions and version constraints to modules and imports respectively. Allowing these to refer to module names gives flexibility over whether code consumers or producers have control.

Local concurrency Local concurrency is important for distributed programming. Acute provides a minimal level of support, with threads, mutexes and condition variables. Local messaging libraries can be coded up using these, though in a production implementation they might be built-in for performance. We also provide thunkification, allowing a collection of threads (and mutexes and condition variables) to be captured as a thunk that can then be marshalled and communicated (or stored); this enables various constructs for mobility to be coded up.

We deal with the interplay among these features and the core, in particular with the subtle interplay between versions, modules, imports, and type identity, requiring additional structure in modules and imports. We develop a semantic definition that tracks abstraction boundaries, global names, and hashes throughout compilation and execution, but which still admits an efficient implementation strategy.

The definition is too large to make proofs of the properties feasible with the available resources and tools. To increase confidence in both semantics and implementation, therefore, our implementation can optionally type-check the entire configuration after each reduction step. Our strategy has been to synchronise the development of the formal specification of the Acute language with that of its implementation. As a result we have been able to quickly discover and fix errors in the type-system and in the run-time semantics.

The specification of Acute, together with a discussion of the design rationale, is available as an INRIA Research Report [62].

6.2 Versions and version constraints

In this subsection we present in further detail the versioning support of Acute.

In a single-executable development process, one ensures the executable is built from a coherent set of versions of its component modules by choosing what to link together — in simple cases, by working with a single code directory tree. In the distributed world, one could do the same: take sufficient care about which modules one links and/or rebinds to. Without any additional support, however, this is an error-prone approach, liable to end up with semantically-incoherent sets of versions of components interoperating. Typechecking can provide some basic sanity guarantees, but cannot capture these semantic differences.

One alternative is to permit rebinding only to identical copies of modules that the code was initially linked to. Usually, though, more flexibility will be required — to permit rebinding to modules with “small” or “backwards-compatible” changes to their semantics, and to pick up intentionally location-dependent modules. It is impractical to specify the semantics that one depends upon in interfaces (in general, theorem proving would be required at link time, though there are intermediate behavioural type systems). We therefore we introduce *versions* as crude approximations to semantic module specifications. We need a language of versions, which will be attached to modules, a language of version constraints, which will be attached to imports, a satisfaction relation, checked at static and dynamic link times, and an implication relation between constraints, for chains of imports.

Now, how expressive should these languages be? Analogously to the situation for *resolvespecs*, there is a tension between allowing arbitrary computation in defining the relations and supporting compile-time analysis. Ultimately, it seems desirable to make the basic module primitives parametric on abstract types of versions and constraints — in a particular distributed code environment, one may want a particular local choice for the languages. For Acute once again we choose not the most general alternative, but instead one which should be expressive enough for many examples, and which exposes some key design points.

1 It is common to use version numbers which are supplied by the programmer, with no checked relationship to the code. As an arbitrary starting point, we take version numbers to be nonempty lists of natural numbers, and version constraints to be similar lists possibly ending in a wildcard `*` or an interval; satisfaction is what one would expect, with a `*` matching any (possibly empty) suffix.

Many minor enhancements are possible and straightforward. Arbitrarily, we enhance version constraints with closed, left-open and right-open intervals, e.g. `1.5-7`, `1.8.-7`, and `2.4.7-`. These are certainly not exactly what one wants (they cannot express, for example, the set of all versions greater than `2.3.1`) but are indicative.

The *meanings* of these numbers and constraints is dependent on some social process: within a single distributed development environment one needs a shared understanding that new versions of a module will be given new version numbers commensurate with their semantic changes.

2 To support tighter version control than this, we can make use of the global module names (hash- or freshly-generated) introduced in: equality testing of these names is an implementable check for module semantic identity. We let version numbers include `myname` and version constraints include module identifiers `M` (those in scope, obviously). In each case the compiler or runtime writes in the appropriate module name. This supports a useful idiom in which code producers declare their exact identity as the least-significant component of their version number, and consumers can choose whether or not to be that particular. For example, a module `M` might

specify it is version `2.3.myname`, compiled to `2.3.0xA564C8F3`; an import in that scope might require `2.3.M`, compiled to `2.3.0xA564C8F3`, or simply `2.3.*`; both would match it.

A key point is the balance of power between code producers and code consumers. The above leaves the code producer in control, who can “lie” about which version a module is — instead of writing `myname` they might write a name from a previous build. This is desirable if they know there are clients out there with an exact-name constraint but also know that their semantic change from that previous build will not break any of the clients.

3 Finally, to give the code consumer more control, we allow constraints not only on the version field of a module but also on its actual name (which is unforgeable within the language). Typically one would have a definition of the desired version available in the filesystem (in Acute bringing it into scope as `M` with an `include`) and write `name=M`. (These exact-name constraints are also used to construct default imports when marshalling). One could also cut-and-paste a name in explicitly: `name=0xA564C8F3`. To guarantee that only mutually-tested collections of modules will be executed together, e.g. when writing safety-critical software, this would be the desired idiom everywhere, perhaps with development-environment support.

The current Acute version numbers and constraints, including all the above, are as follows.

```

avne ::=          Atomic version number expression
  n          natural number literal
  N          numeric hash literal
  myname     name of this module

vne  ::=          Version number expression
  avne | avne.vne

avce ::=          Atomic version constraint expression
  n          natural number literal
  N          numeric hash literal
  M          name of module M

dvce ::=          Dotted version constraint
  avce | n-n' | -n | n- | * | avce.dvce

vce  ::=          Version constraint
  dvce       dotted version constraint
  name = M   exact-name version constraint

```

Version number and constraint expressions appear in modules and imports as below.

```

definition ::= ...
  module M:Sig version vne = Str ...
  | import M:Sig version vce ...
  by resolvespec = Mo

```

In constructing hashes for modules we also take into account their version expressions, to prevent any accidental equalities. That version expression can mention `myname`, and, as we do not wish to introduce recursive hashes,

the hash must be calculated before compilation replaces `myname` with the hash.

It turns out that one needs exact-name version constraints not just for user-specified tight version constraints, as in the idiom above, but also during marshalling, when one may have to generate imports for module bindings that cross a mark. Exact-name constraints seem to be the only reasonable default to use there.

One might wish to extend the version language further with conjunctive version number expressions and disjunctive constraints. One might also wish to support cryptographic signatures on version numbers. Both would affect the balance of power between code producer and consumer, and further experience is needed to discover what is most usable.

Finally, we have had to choose whether version numbers are hereditary or not. A hereditary version number for a module `M` would include the version numbers of all the modules it depends on (and the version constraints of all the imports it uses), whereas a non-hereditary version number is just a single entity, as in the grammar above. The hereditary option clearly provides more data to users of `M`, but, concomitantly, requires those users to understand the dependency structure — which usually one would like a module system to insulate them from. If one really needs hereditary numbers, perhaps the best solution would be to support version number expressions that can calculate a number for `M` in terms of the numbers of its immediate dependencies, e.g. adding tuples and `version(M)` expressions to the *avne* grammar.

Just as for *withspecs* one might need rich development-environment support. Local specifications of version constraints, spread over the imports in the source files of a large software system, could be very inconvenient. One might want to refer to the version numbers of a source-control system such as CVS, for example.

6.3 Comparison with the state of the art

Acute builds on our earlier work: compile-time fresh generation of abstract type names and channel names [63]; hash-generation of effect-free abstract type names [64]; and dynamic rebinding [65]. There is extensive related work on module systems, dynamic binding, dynamic type tests, and distributed process calculi. For most of this we refer the reader to the discussion in those papers, confining our attention here to some of the most relevant distributed programming language developments.

Early work on adding local concurrency to ML resulted in Concurrent ML [66] and the initial *Facile*, both based on the SML/NJ implementation. *Facile* was later extended with rich support for distributed execution, including a notion of *location* and computation mobility [67]. dML [68] was another distributed extension of ML, implementable by translation into remote procedure calls without requiring communication at higher types. Erlang [6] supports concurrency, messaging and distribution, but without static typing.

The Pict experiment [69] investigated how one could base a usable programming language purely on local concurrency, with a π -calculus core instead of primitive functions or objects. The Distributed Join Calculus [42] and subsequent JoCaml implementation [70] modified the π primitives with a view to distribution, and added location hierarchies and location migration. The runtime involved a complex forwarding-pointer distributed infrastructure to ensure that, in the absence of failure, communication was location-independent. (Polyphonic C[#] [71] adds the Join Calculus local concurrency primitives to a class-based language.) Other work in the 1990s was also aimed at providing distribution transparency, notably *Obliq* [72], with network-transparent remote object references above Modula3's network objects.

Distribution transparency, while perhaps desirable in tightly-coupled reliable networks, cannot be provided in systems that are unreliable or span administrative boundaries. Work on *Nomadic Pict* [40, 73] adopted a lower

level of abstraction, showing how a wide variety of distributed infrastructure algorithms, including one similar to that of the JoCaml implementation, could be expressed in a high-level language; one was proved correct. The low level of abstraction means the core language can have a clean and easily-understood failure semantics.

A distinct line of work has focussed on typing the entire distributed system to prevent resource access failures, for $D\pi$ [74] and with modal types [75]. Even where this is possible, however, programmers must still deal with low-level network failure.

Work on Alice [76, 77] is perhaps closest to ours, with ML modules, support for marshalling (‘pickling’) arbitrary values, and run-time fresh generation of abstract type names.

Many of the language designs cited above address distributed *execution*, with type-safe interaction within a single program that forks across the network, but there has been little work on distributed *development*, on typed interaction *between* programs¹, or on version change.

Both Java and .NET have some versioning support, though neither is integrated with the type system. Java serialisation, used in RMI, includes *serialVersionUID*s for classes of any serialised objects. These default to (roughly) hashes of the method names and types, not including the implementation. Class authors can override them with hashes of previous versions. Linking for Java, and in particular binary compatibility, has been studied by Drossopoulou et al [78]. The .NET framework supports versioning of *assemblies* [79]. Sharable assemblies must have *strong names*, which include a public key, file hashes, and a *major.minor.build.revision* version. Compile-time assembly references can be modified before use by XML policy files of the application, code publisher, and machine administrator; the semantics is complex.

Explicit versioning is common in package management, however. For example, both RedHat and Debian packages can contain version constraints on their dependencies, with numeric inequalities and capability-set membership. ELF shared objects express certain version constraints using pathname and symlink conventions. Vesta [80] provides a rich configuration language.

References

- [1] S. Pratschner. Simplifying deployment and solving DLL hell with the .NET framework. msdn.microsoft.com/, 2001.
- [2] Matthias Zenger. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, EPFL, Switzerland, 2004.
- [3] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, April 2003.
- [4] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for java-like languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, Long Beach, CA, USA, January 2005.
- [5] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoye. Formalizing dynamic software updating. In *Proc. 2nd International Workshop on Unanticipated Software Evolution (USE 2003)*, April 2003.

¹Several, including JoCaml and Nomadic Pict, have ad-hoc ‘traders’ for establishing initial connections between programs.

- [6] J. Armstrong, R. Viriding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996. 2nd ed.
- [7] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Proc. 17th POPL*, pages 31–46, 1990.
- [8] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proc. 22nd POPL*, pages 233–246, 1995.
- [9] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Principles and Practice of Declarative Programming*, pages 160–171. ACM Press, 2003.
- [10] Tom Hirschowitz. *Modules mixins, modules et récursion étendue en appel par valeur*. Thèse de doctorat, Université Paris 7, 2003.
- [11] Z. M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1–3):97–170, 2002.
- [12] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, September 1992.
- [13] Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, December 1998.
- [14] José Luis Vivas Frontana. *Dynamic Binding of Names in Calculi for Mobile Processes*. PhD thesis, KTH, Stockholm, March 2001.
- [15] MIT Scheme. <http://www.swiss.ai.mit.edu/projects/scheme/>.
- [16] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In *Proc. 27th POPL*, pages 108–118, 2000.
- [17] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proc. 20th POPL*, pages 479–492, 1993.
- [18] Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2):201–231, 1998.
- [19] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM TOPLAS*, 16(3):456–492, May 1994.
- [20] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1-2):249–272, 2001.
- [21] Masatomo Hashimoto and Akinori Yonezawa. MobileML: A programming language for mobile computation. In *COORDINATION*, number 1906 in LNCS, page 198 ff., 2000.
- [22] Jacques Garrigue. Dynamic binding and lexical binding in a transformation calculus. In *Workshop on Functional and Logic Programming*, 1995.
- [23] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–219. Elsevier North-Holland, 1987.

- [24] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *ACM Conference on LISP and Functional Programming, Snowbird, Utah*, pages 52–62, July 1988.
- [25] Christian Queinnec. A library of high level control operators. *Lisp Pointers, ACM SIGPLAN Special Interest Publ. on Lisp*, 6(4):11–26, October 1993.
- [26] C.A. Gunter, D. Rémy, and J.G. Riecke. A generalisation of exceptions and control in ML-like languages. In *Proc. FPCA*, pages 12–23, 1995.
- [27] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations: A duumvirate of control operators. In *Proc. PLILP, LNCS 844*, pages 182–197, September 1994.
- [28] Sophia Drossopoulou and Susan Eisenbach. Manifestations of Java dynamic linking. http://www-dse.doc.ic.ac.uk/projects/slurp/dynamic_link/Manifest.pdf.
- [29] POSIX dlopen specification. <http://www.opengroup.org/onlinepubs/007904975/functions/dlopen.html>.
- [30] X. Leroy et al. The Objective Caml system release 3.04 documentation, December 2001.
- [31] François Rouaix. A Web navigator with applets in Caml. In *Proc. 5th World Wide Web Conference*, pages 1365–1371, 1996.
- [32] Dominic Duggan. Sharing in Typed Module Assembly Language. In *Proc. 3rd Workshop on Types in Compilation*, pages 85–116, 2000.
- [33] Michael Hicks, Stephanie Weirich, and Karl Cray. Safe and flexible dynamic linking of native code. In *Proc. 3rd Workshop on Types in Compilation, LNCS 2071*, pages 147–176, 2000.
- [34] Michael Hicks and Stephanie Weirich. A calculus for dynamic loading. Technical Report MS-CIS-00-07, University of Pennsylvania, 2000.
- [35] Albert Serra, Nacho Navarro, and Toni Cortes. DITools: Application-level support for dynamic extension and flexible composition. In *Proc. USENIX Annual Technical Conference*, pages 225–238, 2000.
- [36] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Proc. 13th Computer Security Foundations Workshop*, pages 269–284, 2000.
- [37] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. 1st FoSSaCS, LNCS 1378*, pages 140–155, 1998.
- [38] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proc. 26th POPL*, pages 93–104, 1999.
- [39] Alan Schmitt. Safe dynamic binding in the join calculus. In *Proc. IFIP TCS 2002*, 2002.
- [40] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, 1999.
- [41] Tom Chothia and Ian Stark. A distributed pi-calculus with local areas of communication. In *Proc. 4th HLCL, ENTCS 41.2*, 2000.
- [42] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. 7th CONCUR, LNCS 1119*, 1996.

- [43] Michael Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, August 2001.
- [44] Dominic Duggan. Type-based hot swapping of running modules. In *Proc. 5th ICFP*, pages 62–73, 2001.
- [45] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, December 1997.
- [46] Chris Walton. *Abstract Machines for Dynamic Computation*. PhD thesis, University of Edinburgh, 2001. ECS-LFCS-01-425.
- [47] J. L. Armstrong and R. Virding. Erlang — An Experimental Telephony Switching Language. In *XIII International Switching Symposium*, Stockholm, Sweden, May 27 – June 1, 1991.
- [48] M. W. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, The University of Pennsylvania, August 2001.
- [49] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
- [50] D. Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.
- [51] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 14(2):111–128, September 1991.
- [52] G. Hjalmtýsson and R. Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proc. USENIX*, June 1998.
- [53] D. Duggan. Type-based hot swapping of running modules. In *Proc. ICFP*, 2001.
- [54] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of Java software. In *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2002.
- [55] S. Drossopoulou and S. Eisenbach. Flexible, source level dynamic linking and re-linking. In *Proc. ECOOP 2003 Workshop on Formal Techniques for Java Programs*, 2003.
- [56] C. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. USENIX*, June 2003.
- [57] C. Boyapati, B. Liskov, L. Shrira, C-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Proc. OOPSLA*, 2003.
- [58] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. ECOOP*, 2000.
- [59] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.
- [60] D. Walker. A type system for expressive security policies. In *Proc. POPL*, pages 254–267, January 2000.
- [61] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proc. PLDI*, 2002.

- [62] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, October 2004. Also published as INRIA RR-5329. 193pp. Available from <http://www.cl.cam.ac.uk/users/pes20/acute/UCAM-CL-TR-605.pdf>.
- [63] P. Sewell. Modules, abstract types, and distributed versioning. In *Proc. 28th POPL*, 2001.
- [64] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, 2003.
- [65] G. Bierman, M. Hicks, P. Sewell, G. Stoyale, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ . In *Proc. ICFP*, 2003.
- [66] J. H. Reppy. *Concurrent Programming in ML*. Cambridge Univ Press, 1999.
- [67] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *CONCUR'96, LNCS 1119*, 1996.
- [68] Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. In *Proc. POPL*, pages 99–112, 1993.
- [69] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. 2000.
- [70] JoCaml. <http://pauillac.inria.fr/jocaml/>.
- [71] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C^\sharp . In *Proc. ECOOP, LNCS 2374*, 2002.
- [72] L. Cardelli. A language with distributed scope. In *Proc. 22nd POPL*, pages 286–297, 1995.
- [73] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proc. POPL*, pages 116–127, January 2001.
- [74] M. Hennessy, J. Rathke, and N. Yoshida. Safedpi: A language for controlling mobile code. In *Proc. FOSSACS, LNCS 2987*, 2004.
- [75] T. Murphy, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proc. LICS*, 2004.
- [76] The Alice project, 2003. <http://www.ps.uni-sb.de/alice/>.
- [77] A. Rossberg. Generativity and dynamic opacity for abstract types. In *Proc. 5th PPDP*, August 2003.
- [78] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus towards a model of separate compilation, linking and binary compatibility. In *Proc. LICS*, pages 147–156, 1999.
- [79] Packaging and deploying .net framework applications (.net framework tutorials), 2003. <http://msdn/microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/netdevanchor.asp>.
- [80] Vesta. <http://www.vestasys.org/>.