Abstraction Preservation and Subtyping in Distributed Languages

Pierre-Malo Deniélou   James J. Leifer

INRIA Rocquencourt

{Pierre-Malo.Denielou,James.Leifer}@inria.fr

Abstract

In most programming languages, type abstraction is guaranteed by syntactic scoping in a single program, but is not preserved by marshalling during distributed communication. A solution is to generate *hash types* at compile time that consist of a fingerprint of the source code implementing the data type. These hash types can be tupled with a marshalled value and compared efficiently at unmarshall time to guarantee abstraction safety. In this paper, we extend a core calculus of ML-like modules, functions, distributed communication, and hash types, to integrate structural subtyping, user-declared subtyping between abstract types, and bounded existential types. Our semantics makes two contributions: (1) the explicit tracking of the interaction between abstraction boundaries and subtyping; (2) support for user-declared module upgrades with propagation of the resulting subhashing relation throughout the network during communication. We prove type preservation, progress, determinacy, and erasure for our system.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords programming language semantics, serialisation and distributed computation, module systems, type theory, subtyping

1. Introduction

1.1 Background and motivation

Abstract types are a powerful feature of modern programming languages. They arise when the implementation of a collection of types and accompanying functions, often called a *module*, is partly hidden by an interface. The creation and manipulation of an abstract data type are then constrained by the functions declared in its interface.

This abstraction mechanism helps the programmer to build data types that support two properties. The first is *invariant preservation*, which states that the interface, by the limits it imposes on data manipulation, ensures the preservation of the internal invariants. The second is *secrecy* (also known as encapsulation) which states that the underlying implementation of all abstract types and of values of those types is hidden and cannot be observed by external code outside the abstraction boundaries. The programming language semantics and the type system then ensure that these properties, which we collectively call *abstraction preservation*, hold throughout all executions.

In the ML [9] family of languages, it is sufficient to check statically at compile time the scoping and typing information required to ensure abstraction preservation. Thus they may be forgotten at run time. A completely untyped run time is, however, inadequate in a distributed environment where a value marshalled on one machine may be transmitted and unmarshalled on another.

Even if one exchanges *concrete* type information to prevent unsafe unmarshalling of a value at the wrong concrete type, this is not sufficient for ensuring either of abstraction preservation's two properties. Nor can we trivially solve this by relying on abstract type *names*, since they are only meaningful when they are unambiguously linked to their original module declaration — which is not possible to check on a distant machine.

We proposed a solution in [7], which was later developed in the Acute programming language [12]. A global name for an abstract type is constructed at compile time by hashing the source of the module that exports that type. We thereby identify, between distributed programs, abstract types that share the same implementation (representation type and code), and thus the same abstract properties. We used the mechanism of coloured brackets introduced by Zdancewic et al. [5] to track the evolution of abstract values through the operational semantics and thus show the abstraction preservation property.

1.2 Problem and solution

Distributed systems involve code that is not only executed on different machines, but also asynchronously modified and deployed across a network. We therefore regard the distributed communication of values between *differing versions* of code to be an expected, rather than exceptional, situation.

In our previous work on Acute, we considered two sorts of version change. The first was concerned with dynamic rebinding. When marshalling a value, the user could choose which code pointers in the value should have their targets included in the marshalled package and which should be cut. The cut pointers would then be rebound at the receiver side to locally available modules, provided the locally available module versions satisfied the right version constraints. This aspect of rebinding is completely outside the scope of the present paper.

The second sort was concerned with marshalling a value at one abstract type and unmarshalling it according to another, *upgraded*, one. The present paper directly addresses this issue. In Acute, the only way to support compatibility between abstract types is via the strong coercion `with!`, which makes the hash of the new module's abstract type equal the hash of the old module's one. This strategy

ICFP06 September 18–20, 2006, Portland, Oregon, USA

has a highly efficient implementation: the compiler checks that the underlying implementation types are the same in the new and old module and then gives the new module the same hash as the old. The drawback to this approach is that it only supports bidirectional compatibility: unmarshalling can convert values from the old type to the new one as well as from the new to the old, even if the user only asserts abstraction safety for one of those directions.

We thus propose to use subtyping to model a more refined unmarshalling test and to introduce subtyping throughout the language so that the user may declare unidirectional compatibility.

Subtyping will come from three different sources. First, we add immutable records, almost the simplest possible data structure that supports structural subtyping. Second, to model upgrade compatibility between two modules, we introduce a constrained form of subtyping between abstract types. Third, we allow the declaration of partially abstract types (also known as *bounded existentials*) which enable the user to partially reveal information about an abstract type.

In order to define an operational semantics that encompasses these features, we needed to make substantive changes to [7]. To handle the second feature, namely subtyping between abstract types, we introduce into the semantics a subhash relation that is calculated at compile time and propagated throughout the network at run time. For the third feature, namely partial abstract types, we add explicit subtyping operations to the syntax (instead of implicit subsumption) and change the typing and reduction rules for coloured brackets to accommodate *additive* brackets, that allow code inside them to see through the ambient abstractions but not vice versa. Our erasure theorem shows that the explicit subtyping and coloured bracket operations can be safely eliminated in an implementation.

1.3 Outline

Our aim in this paper is to provide a type-safe, abstraction-preserving core calculus featuring a simple module system with abstract types, networking and marshalling primitives, and hashes and coloured brackets. Our presentation is divided in three stages.

In the first stage, section 2, we define our core calculus, which closely follows our previous work in [7], with two main changes: a subtype check replaces the equality check at unmarshall time and the system is extended to include simple records and structural subtyping. The details presented in this section will be used as setup for later developments.

In the second, section 3, we introduce user declarations of module upgrades which induce compatibility between abstract types based on subtyping. This leads to changes to the operational semantics for propagating the enriched subtype relation throughout the network during communication.

Finally, in the third, section 4, we extend the module signature syntax to allow partial abstract types. These changes force us to replace implicit subtyping subsumption by explicit subtyping nodes and to make coloured brackets partially transparent.

We present in section 5 our theorems for type preservation, progress, and determinacy. We discuss in section 6 implementation issues for our calculus and state a type erasure theorem. We conclude with the future work in section 7.

2. Base system

In this section we recall the core distributed λ-calculus with abstract types of [7]. Our presentation differs from the original by the introduction of records and structural subtyping. This calculus is the initial setup upon which all further modifications in this paper are made, thus we treat in detail the unusual aspect of this calculus and detail the significant typing and semantics rules.

We give in Figure 1 the initial, "temporary", syntax. In the technical report [4] the reader can find the complete final definition that encompasses all sections in this paper as well as all the theorems and proofs.

2.1 Typed λ-calculus and records

The heart of our system is a simply-typed call-by-value λ-calculus with tuples and records. Concretely, our expression grammar includes expression variables x, functions $\lambda x.e$ and applications $e\,e'$, tuples $(e_1, ..., e_j)$ and projection $\mathbf{proj}_i\, e$, records $\{l_1 = e_1; ...; l_j = e_j\}$ and access to a field $e.l_i$. These elements have the usual typing and reduction rules. Records are not the focus of our paper, but are a convenient way of obtaining structural subtyping; as a result, we take them in almost their simplest form, namely ordered, immutable, and non extensible. The base expressions are unit $()$ and integers $\underline{n}$.

We write *values* as v^c, where c is a colour annotation as explained later in section 2.9.

2.2 Types, type equivalence, and subtyping

These expressions are typed using a standard ML type grammar: the base types are UNIT and INT, and the constructors are arrows $T \rightarrow T'$, tuples $T_1 * ... * T_j$ and records $\{l_1{:}T; ...; l_j{:}T\}$. There is also a special base type corresponding to data in raw form (i.e. byte strings) that circulate on the network: BYTES.

There are two important relations for comparing types. The first, an equivalence relation, written $==$, is used to relate the abstract types with their implementations under certain precise conditions that we will detail later in section 2.7. The second, a subtyping relation, denoted $<:$, is induced by record typing, and is lifted to the other constructors in the usual way. Subtyping includes type equivalene. As we want to keep the subtyping relation between records as simple as possible, we consider only width subtyping between records, i.e. two conditions hold for one record type to be a subtype of another: (1) the supertype's labels are included in the subtype's; (2) types under a common label are equivalent. Then, for example we have $\{l_1{:}T_1, ..., l_j{:}T_j, ..., l_k{:}T_k\} <: \{l_1{:}T_1, ..., l_j{:}T_j\}$. (We treat depth subtyping in our handling of tuples.)

These two relations, equivalence and subtyping, are mirrored by two different kinds that we will use to describe sets of types: the kind $\mathbf{Eq}(T)$ represents the types that are equivalent (w.r.t. $==$) to T; the kind $\mathbf{Le}(T)$ represents the subtypes (w.r.t. $<:$) of T. Type equivalence is included in subtyping.

Finally we add to the type grammar a greatest type, written $\top$, so that the correctness of a type T can be easily expressed by the statement that $T <: \top$ or, equivalently, $T{:}\mathbf{Le}(\top)$.

2.3 Modules

For simplicity, our module syntax is just rich enough for each module to define one abstract type. Concretely, a *module*, referred to by the metavariable U, is the association of a structure M and a signature S: the full declaration is written $\mathbf{module}\, N_U = M{:}S$. We here distinguish, as in [6], the internal name U (which binds and is thus alpha convertible) from the fixed external name N.

A *structure* M defines one type abbreviation and one value, written $[T, v^\bullet]$; of course the value itself may be a record of several fields, for example. Since each structure declares only one type and one value, they do not need individual identifiers. The declared type of a module U will then be referred as U.TYPE and the value as $U.\mathrm{term}$. In Ocaml, we would have to name the type and value explicitly, for example writing our structure $[T, v^\bullet]$ as:

```
struct type t = T let x = v• end
```

In order to avoid consideration of side effects in module initialisation, we only consider expression fields that are values. The reader

Figure 1. Initial syntax of the calculus

The elements that are below the dotted lines are not in user source programs but are introduced by compilation and reduction steps.

Summary of variables:

x	expressions
X	types
U	modules

Expressions:

$e ::= () \mid \underline{n}$	unit, integer
$\mid (e, ..., e) \mid \mathbf{proj}_i\, e$	tuple
$\mid \{l_1 = e, ..., l_j = e\} \mid e.l_i$	records
$\mid x \mid \lambda x{:}T.e \mid e_1\, e_2$	λ-calculus (x binds in e)
$\mid U.\mathrm{term}$	term declared by module U
$\mid \mathbf{mar}\,(e{:}T)$	marshalling
$\mid \mathbf{unmar}\; e{:}T$	unmarshalling
$\mid\ !\,e \mid\ ?$	send and receive
....	
$\mid \mathbf{marshalled}_{\,c}(e{:}T)$	marshalled value
$\mid \mathbf{Unmarfailure}^T$	unmarshalling error
$\mid [e]_c^T$	coloured brackets

Values:

$v^c ::= () \mid \underline{n}$	unit, integer
$\mid (v_1^c, ..., v_j^c)$	tuple
$\mid \{l_1 = v_1^c, ..., l_j = v_j^c\}$	record
$\mid \lambda x{:}T.e$	function
$\mid \mathbf{marshalled}_{\,c}(e{:}T)$	marshalled value
$\mid [v^{c'}]_{c'}^{h.\mathrm{TYPE}}$	coloured brackets ($h \notin c \cup c'$)

Machines:

$m ::= e$	expression
$\mid \mathbf{module}\, N_U\ = M{:}S\ \mathbf{in}\ m$	module declaration (U binds in m)

Modules (structures and signatures):

$M ::= [T, v^\bullet]$	structure
$S ::= [X{:}K, T]$	signature (X binds in T)

Kinds:

$K ::= \mathbf{Le}(T)$	kind of all subtypes of T
$\mid \mathbf{Eq}(T)$	singleton kind of T

Types:

$T ::= \mathrm{UNIT} \mid \mathrm{INT}$	base types
$\mid X \mid T \to T$	variable, function
$\mid \{l_1{:}T; ...; l_j{:}T\}$	record ($j > 0$)
$\mid T * ... * T$	tuple
$\mid U.\mathrm{TYPE}$	type declared in the module U
$\mid \mathrm{BYTES}$	byte string
$\mid \top$	top
....	
$\mid h.\mathrm{TYPE}$	hash type

Networks:

$n ::= \mathbf{0}$	empty
$\mid m$	single machine
$\mid (n\|n)$	parallel composition

Colours and hashes:

$h ::= \mathbf{hash}(N, M{:}[X{:}\mathbf{Le}(T_1), T_2])$	hash (X binds in T_2)
$c ::= \bullet$	empty colour
$\mid \{h\}$	colour with one hash
$\mid c \cup c$	union

may wish to look at our analysis of module initialisation issues in the work on Acute [12].

A *signature* S is written $[X{:}K, T]$. It is a dependant pair of two elements. First $X{:}K$ corresponds to the type definition from the structure, where K is a kind specifying if the type is manifest (using $K = \mathbf{Eq}(T_0)$) or abstract (using $K = \mathbf{Le}(\top)$). The type variable X refers to this type and binds to the right in T, the type of the value field. For example, the Ocaml-like syntax of a manifest signature $[X{:}\mathbf{Eq}(\mathrm{INT}), \mathrm{INT}{\to}X]$ would be

```
sig type t=int val x : int->t end;
```

for an abstract signature $[X{:}\mathbf{Le}(\top), \mathrm{INT}{\to}X]$ the Ocaml translation would be

```
sig type t val x : int->t end.
```

Later, in section 4, we will accept kind declaration of the form $\mathbf{Le}(T')$ for any type T' (not just $\top$), in order to describe partial abstract types. The typing judgement for modules is as follows:

$$\frac{\begin{array}{c} E \vdash_\bullet T{:}\mathbf{Le}(\top) \\ E \vdash_\bullet [T_0, v^\bullet]{:}S \\ E, U{:}S \vdash_\bullet m{:}T \end{array}}{E \vdash_\bullet (\mathbf{module}\, N_U\ = [T_0, v^\bullet]{:}S\ \mathbf{in}\ m){:}T} \quad (1)$$

The premises check that T does not have U as a free variable, that the signature corresponds to the structure, and that the rest of the code may be typed in an environment enriched by the present declaration.

2.4 Networking

Our calculus features simple communication of byte strings between *machines*, denoted m. A machine consist of a succession of module definitions, followed by an expression. Every machine (modules and main expression) is compiled separately. A network n is simply modelled by a parallel composition of machines $m_1 \mid m_2 \mid ...$, linked by an anonymous, unique, shared, untyped channel only capable of transmitting byte strings. Any machine may send a message on this channel, which may be removed and received once by any machine. Note that we only consider a reliable and safe network: no message gets altered; no attacker is present to listen or send any message.

While it might be possible to do away with the network via a continuation-style encoding into λ-calculus, the explicit representation of machines and the network carries little overhead in our semantics and allows us to have a precise view of the differing runtime information available at each machine (such as the subhash relation which is propagated during communication, as we see in Section 3.3).

The communication expressions are ! of type $\mathrm{BYTES}{\to}\mathrm{UNIT}$ for sending and ? of type BYTES for receiving. The communication rule is the following ($CC_c^\bullet$ and $CC_{c'}^\bullet$ are contexts, they are described more precisely in section 2.9).

$$CC_c^\bullet.!\, v^c \mid CC_{c'}^\bullet.? \quad \longrightarrow \quad CC_c^\bullet.() \mid CC_{c'}^\bullet.v^c$$

Finally, we provide the two operations of marshalling and unmarshalling, written $\mathbf{mar}\,(e{:}T)$ and $\mathbf{unmar}\; e{:}T$, which are statically annotated with types. The first converts any expression to a byte string and the second reads a byte string to recreate a value. We do not describe the internals of these two operations but do give their operational semantics later. The type annotation on $\mathbf{mar}$ gives the type of the value to be marshalled; the type annotation on $\mathbf{unmar}$ gives the expected type of the value resulting from unmarshalling. At unmarshall time, we compare the sent type with the expected type by a subtype check. This is sufficient to ensure the safety of the unmarshalling operation. However, when the types are abstract, a simple equality check (we do not have subtyping between abstract types yet) is not sufficient for abstraction safety, as we show in the following example.

Example 2.1 (Abstract type preservation failure) Here is a simple module providing an abstract counter, which, for simplicity, we write in Ocaml-like syntax:

```
module Counter =
struct                      sig
 type t = int                type t
 let init = 0          :     val init : t
 let incr x = x + 1          val incr : t -> t
 let value x = x             val value : t -> int
end                         end
```

Suppose that we want to send a counter over the network, which can be done with an expression such as:

`!(mar(Counter.incr(Counter.init):Counter.t))`

On the receiver side, however, suppose that we have a different module implementing a counter that always stays even, but with the same name.

```
module Counter =
struct                      sig
 type t = int                type t
 let init = 0          :     val init : t
 let incr x = x + 2          val incr : t -> t
 let value x = x             val value : t -> int
end                         end
```

To import a counter and see its value: we write

`Counter.value(unmar(?): Counter.t).`

For safety, we have to check that the representation types are equal (here they are both integers), but if that is all we check, we risk importing a counter which has an odd value, thus breaking the expected invariants on the receiver's side. □

Therefore, we need a way to unambiguously identify each different abstract type, independently from the originating machine. This can be achieved by using hashes.

2.5 Hashes and compilation

We will now take as run-time identifiers for abstract types a fingerprint of their original module source code. A hash type, written $h.\text{TYPE}$, will then represent the abstract type of the module whose source code has hash h. The hashes are computed from the external name, the structure and the signature of a given module: $h = \mathbf{hash}(N, ([T, v^\bullet]{:}[X{:}\mathbf{Le}(T''), T']))$. In practice, we use a normalized form of the syntax tree and a usual (cryptographic) fingerprint algorithm; for simplicity, we ignore possible collisions.

To show exactly where hash types come in the compilation process of machines, we have to first describe the compilation of manifest modules. The compilation rule in this case is the following. Braces denote a substitution.

$$\begin{aligned}&\mathbf{module}\, N_U = [T, v^\bullet]{:}[X{:}\mathbf{Eq}(T''), T']\ \mathbf{in}\ m\\ &\quad\longrightarrow_{\mathsf{m}}\ \{U.\text{TYPE}{\leftarrow}T'', U.\text{term}{\leftarrow}v^\bullet\}m\end{aligned}$$

The references to the module are all substituted at compile time. $U.\text{TYPE}$ is replaced by the type present in the signature, while $U.\text{term}$ is replaced by its value as defined in the structure.

However our compilation rule for abstract types has to be different since we want abstract types to be present at run-time and different from their concrete representation. Then for a module described by $\mathbf{module}\, N_U = [T, v^\bullet]{:}[X{:}\mathbf{Le}(T''), T']$, compilation will replace all occurrences of $U.\text{TYPE}$ by the corresponding hash-type $h.\text{TYPE}$ with $h = \mathbf{hash}(N, ([T, v^\bullet]{:}[X{:}\mathbf{Le}(T''), T'])$.

Naively introducing hashes via compilation, however, breaks type preservation, as we see next.

Example 2.2 (Introduction of hashes) Let us take the first implementation of an abstract `Counter` seen above. If we have as an expression the following well-typed term:

`(fun x:Counter.t -> ()) (Counter.init)`

Then, if we write h for the hash of the module `Counter`, the compilation will lead to:

`(fun x:h.t -> ()) 0`

However, this expression is not typable since `h.t` is an abstract type and its real implementation cannot be revealed outside the `Counter` module: we cannot prove that `0` of type `int` has type `h.t` as supertype. □

2.6 Coloured brackets

A solution to the previous problem is to use *coloured brackets* to wrap abstract terms and separate the revelation areas of abstract types. We write these terms as $[e]_c^T$, where T is the outside type and c the *colour*, i.e. a set of hashes representing the abstract types that are allowed to be revealed inside. We write $\bullet$ for the empty colour.

Example 2.3 (Coloured brackets) If we reuse the situation of the example 2.2, the result of the compilation will be:

`(fun x:`$h.t$` -> ())` $[0]_h^{h.t}$

where h is the hash of the `Counter` module. Then $[0]_h^{h.t}$ can be typed with type $h.t$ and, inside the brackets, using the colour h, we can associate 0 and its natural type `int` with type $h.t$. This typing proof is detailed below in example 2.4. □

This example shows how coloured brackets can split an expression into a subterm where an abstract type, i.e. a hash type, can be revealed, and a context where the abstract type remains opaque. Our type system has to take this behaviour into account and be able to prove different typing judgements depending on the set of abstract types that can be revealed, i.e. the colour. That is the reason why we annotate our typing judgements by a colour: a typing judgement of expression e with type T in a colour c, with an environment of type, module and expression variables, denoted E, will be written $E \vdash_c e{:}T$.

Of course, other sorts of judgements will use the same colour annotation, such as kinding judgements, correction, equivalence or subtyping judgements.

Since a current colour represents the state of knowledge we have about abstract types, and brackets are the borders between these states, the typing rule for bracketed expression can be written in the following way.

$$\frac{E \vdash_{c'} e{:}T \qquad E \vdash_c \text{ok}}{E \vdash_c [e]_{c'}^T{:}T}$$

The coloured brackets are typed in colour c, while the internal expression is typed in colour c'. The second premise only checks the correctness of the colour c.

2.7 Revelation and equivalence.

Coloured brackets define, in a term, a limit between a subterm where some abstract types are revealed and the rest of the expression. The revelation mechanism relies on the equivalence relation $==$, which will relate a hash type with its implementation when the colour allows it, i.e. when the hash belongs to the current colour. The following rule reflects this behaviour.

$$\frac{E \vdash_c \text{ok}}{\begin{array}{l}E \vdash_c h.\text{TYPE} == T\\ \text{where } h = \mathbf{hash}(N, [T, v^\bullet]{:}S) \in c\end{array}}$$

We see that we can prove the equivalence between $h.\text{TYPE}$ and its implementation when $h \in c$. We finally need a way to implicitly use equivalences in typing proofs.

$$\frac{E \vdash_c e{:}T \qquad E \vdash_c T == T'}{E \vdash_c e{:}T'}$$

This last rule was the last missing step to prove the typing judgement of coloured brackets, as we show in the following example.

Example 2.4 (Typing proof) Following example 2.3, we show here how we can prove that the expression `Counter.init`, compiled to $[0]_h^{h.t}$, can be typed in the empty colour.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\vdash_h \text{ ok}}}{\vdash_h 0{:}\text{INT}} \qquad \dfrac{\dfrac{\dots}{\vdash_h \text{ ok}}}{\vdash_h h.t == \text{INT}}}{\vdash_h 0 : h.t} \qquad \dfrac{}{\vdash_\bullet \text{ ok}}}{\vdash_\bullet [0]_h^{h.t} : h.t}$$

To prove that $[0]_h^{h.t}$ has type $h.t$ in the empty colour $\bullet$, we use the fact that, in colour $\{h\}$, 0 can be typed with type $h.t$; this is achieved by using the usual integer typing rule and the equivalence between INT and $h.t$ that is coming from the presence of h in the colour $\{h\}$. The proof above is truncated to omit the leaves corresponding to colour and hash correctness.

2.8 Compilation

With the help of coloured brackets, we can finally write a correct module reduction rule in the abstract case.

$$\begin{aligned}&\textbf{module}\, N_U \;= [T, v^\bullet]{:}[X{:}\mathbf{Le}(T''),\, T'] \;\textbf{in}\; m\\ &\longrightarrow_{\mathsf{m}} \{U.\text{TYPE}{\leftarrow}h.\text{TYPE},\, U.\text{term}{\leftarrow}[v^\bullet]_{\{h\}}^{\{X\leftarrow h.\text{TYPE}\}T'}\}\; m\\ &\quad \text{where } \; h = \mathbf{hash}(N, ([T, v^\bullet]{:}[X{:}\mathbf{Le}(T''),\, T']))\end{aligned}$$

We remark that besides the substitution of U.TYPE by h.TYPE which was explained in example 2.2, we need to wrap the value $v^\bullet$ by some coloured brackets, as shown in example 2.3. The brackets have a type annotation T' where all uses of the abstract type (i.e. X) are replaced by the corresponding hash type.

Now that we have completed our compilation rules by the introduction of bracketed expressions, we need a corresponding expression reduction semantics.

2.9 Expression reduction and bracket pushing rules

The concept of current colour is extended to the operational semantics so that abstract values that are surrounded by coloured brackets can be opened and reduced when they enter, by substitution, an authorized part of the term. An expression reduction from e to e' in colour c is then written $e \longrightarrow_c e'$.

As in the type system, the coloured brackets are the limits between the different levels of knowledge. We first present the context rule that expresses the fact that in a coloured bracket of colour c, expressions are evaluated with colour c.

$$\frac{e \longrightarrow_c e'}{C_c^{c'}.e \longrightarrow_{c'} C_c^{c'}.e'}$$

The single-level contexts designated by the variable $C_c^{c'}$ are the usual call-by-value contexts for which $c = c'$, except for the coloured bracket case where the inner colour c can be different from the outer colour c'. The grammar for single-level evaluation contexts is thus:

$C_{c_2}^{c_1} \;::= [\_]_{c_2}^{T}$	coloured bracket
All other single-level contexts impose $c_1 = c_2$.	
$\mid\; \_.l_i \;\mid\; \mathbf{proj}_i\, \_$	field access, projection
$\mid\; \_\, e \;\mid\; v^{c_1}\, \_$	applications
$\mid\; (v_1^{c_1}, .., v_{i-1}^{c_1}, \_, e_{i+1}, .., e_j)$	tuple
$\mid\; \{l_1 = v_1^{c_1}, ..., l_{i-1} = v_{i-1}^{c_1},$ $l_i = \_, l_{i+1} = e_{i+1},$ $..., l_j = e_j\}$	record ($1 \leqslant i \leqslant j$)
$\mid\; \mathbf{mar}\,(\_{:}T) \;\mid\; \mathbf{unmar}\, \_{:}T$	marshall and unmarshall
$\mid\; !\, \_$	send

Coloured evaluation contexts, written $CC_{c_2}^{c_1}$, are just a composed chain of zero or more single-level evaluation contexts:

$CC_{c_2}^{c_1} ::=$	coloured evaluation context
$\mid\; CC_{c'}^{c_1}.C_{c_2}^{c'}$	extra level
$\mid\; \_$	identity, if $c_1 = c_2$

The current colour of the reduction should then be able to open and reveal the abstract expressions that are wrapped by coloured brackets. This has a first consequence: the outcome of the reduction of an expression depends on the current colour. This is the reason why the values, i.e. the subset of expressions that are in normal form, are annotated by colours, as the $v^\bullet$ part from the structure in the module reduction rule. In fact, values are mostly used in the definitions of the reduction rules in order to have a deterministic semantics.

If we now consider the way brackets can be opened, we have first to remark that other expressions than abstract values are wrapped in coloured brackets: the abstract module reduction rule that was presented above always add coloured brackets to the declared value. Our first concern is then to push the brackets through the term until we reach an abstract value: we realize it by a few bracket pushing rules.

$$[()]_{c'}^{\text{UNIT}} \longrightarrow_c ()$$

$$[\underline{n}]_{c'}^{\text{INT}} \longrightarrow_c \underline{n}$$

When we reach a base type which is not abstracted (the coloured brackets are annotated with a concrete type) we just delete the brackets since they are no longer useful.

If we have an explicit type constructor in the annotation on the coloured brackets, we just push the brackets inside and decompose the annotations into each of the subterms.

$$[(v^{c'}, ..., v_j^{c'})]_{c'}^{T_1 * \dots * T_j} \longrightarrow_c ([v_1^{c'}]_{c'}^{T_1}, ..., [v_j^{c'}]_{c'}^{T_j})$$

$$\begin{aligned}&[\{l_1 = v_1^{c'}, .., l_j = v_j^{c'}\}]_{c'}^{\{l_1:T_1,\dots,l_j:T_j\}}\\ &\qquad\qquad \longrightarrow_c \{l_1 = [v_1^{c'}]_{c'}^{T_1}, .., l_j = [v_j^{c'}]_{c'}^{T_j}\}\end{aligned}$$

Pushing the coloured brackets through a λ-expression is more complex because of substitutions, as discussed in Section 2.12.

Finally, our expression reduction rules push all the brackets towards the leaves of the term and either the brackets disappear or they meet an abstract type. The revelation rule then carefully checks if the brackets are useful, i.e. if the hash type is really abstract or can be revealed.

$$[v^{c'}]_{c'}^{h.\text{TYPE}} \longrightarrow_c [v^{c'}]_{c'}^{T} \quad \text{when } h \in c \cap c' \text{ and impl}(h){=}T$$

The side condition checks if the abstract type can be revealed inside the coloured brackets ($h \in c'$) and outside the coloured brackets ($h \in c$). We use then the abbreviation $\text{impl}\,(h)$ to denote the implementation type of h: $\text{impl}\,(h)$ is defined by $\text{impl}\,(\mathbf{hash}(N, [T, v^\bullet]{:}S)) = T$.

Finally, our bracket pushing subsystem needs to deal with sequences of several brackets. In this case, only the inner coloured brackets and the outer type annotation matter.

$$[[v^{c_0}]_{c_0}^{h_0.\text{TYPE}}]_{c_1}^{h_0.\text{TYPE}} \longrightarrow_c [v^{c_0}]_{c_0}^{h_0.\text{TYPE}} \quad (2)$$
$$\text{when } h_0 \notin c \cap c_1 \text{ and } h_0 \notin c_0 \cap c_1$$

These two last rules are slightly different from their counterparts of our previous work in [7]. The first difference lies in the bracket revelation mechanism that was previously directly removing the brackets. The second one is that colours are now sets of hashes instead of sets of at most one element. Our new rules, while they are still equivalent to the old ones, will make the introduction of our main contributions smoother.

Since we just presented the coloured brackets and their corresponding reduction rules, we can switch to the main reduction rule of our system, i.e. the unmarshalling rule. This first requires more explanations and details about the $\mathbf{mar}$ and $\mathbf{unmar}$ constructors.

2.10 Marshalling and unmarshalling.

The typing rules for $\mathbf{mar}$ and $\mathbf{unmar}$ are straighforward. For the former, we check that the expression e has the declared type; for the latter, we check that the received expression is a byte string and the expected type is correct:

$$\frac{E \vdash_c e{:}T}{E \vdash_c \mathbf{mar}\,(e{:}T){:}\text{BYTES}}$$

$$\frac{E \vdash_c T{:}\mathbf{Le}(\top) \quad E \vdash_c e{:}\text{BYTES}}{E \vdash_c (\mathbf{unmar}\; e{:}T){:}T}$$

As we intend it to be possible to send $\mathbf{mar}\,(e{:}T)$ to a different machine, using a different current colour, we let marshalling produce an expression that can be typed in any colour. We call this expression $\mathbf{marshalled}$ and annotate it with the colour in which is was created:

$$\frac{E \vdash_c \text{ok} \qquad \mathbf{nil} \vdash_{c'} e{:}T}{E \vdash_c \mathbf{marshalled}_{\,c'}(e{:}T){:}\text{BYTES}}$$

The premises check if the environment is correct with respect to the current colour and if the expression e has the declared type T in the declared colour.

In some way, $\mathbf{marshalled}_{\,c}$ works as a coloured bracket with the difference that $\mathbf{marshalled}_{\,c}(e{:}T)$ is a value and no reduction can happen in such a context. However, after unmarshalling, the colour annotation on $\mathbf{marshalled}$ turns into real colour brackets to allow the sent expression to be well-typed.

The production rule of $\mathbf{marshalled}$ is just the addition of the colour annotation to a $\mathbf{mar}$ expression.

$$\mathbf{mar}\,(v^c{:}T) \longrightarrow_c \mathbf{marshalled}_{\,c}(v^c{:}T)$$

Now we focus on the unmarshall rule that recreates a value from its marshalled form. By contrast with Leifer et al. [7] who use a type equivalence check to compare the expected and actual types, we choose to use a subtyping check. Also, as described above, the unmarshalled expression is wrapped by coloured brackets in order to be well-typed on this distant machine.

$$\mathbf{unmar}\,(\mathbf{marshalled}_{\,c'}(e{:}T){:}T')$$
$$\longrightarrow_c \begin{cases} [e]_{c'}^{T} & \text{if } \mathbf{nil} \vdash_{\bullet} T <: T' \\ \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases}$$

The subtyping check is done in the empty colour for implementation reasons. We will give more details in section 6. Finally, we solve the case when the subtyping check fails by introducing a failure expression that will stop the evaluation.

2.11 Subtyping

Expressions are typed using implicit subtyping *subsumption*: a well typed expression $e{:}T$ can be typed with any supertype of T:

$$\frac{E \vdash_c e{:}T \qquad E \vdash_c T <: T'}{E \vdash_c e{:}T'}$$

The subtyping relation in itself comes from the records that only support width subtyping. We present the rule here.

$$\frac{E \vdash_c T_i{:}\mathbf{Le}(\top) \quad 1 \leqslant i \leqslant k}{E \vdash_c \{l_1{:}T_1, ..., l_j{:}T_j, ..., l_k{:}T_k\} <: \{l_1{:}T_1, ..., l_j{:}T_j\}}$$

The subtyping relation is lifted in the usual way to the other constructors.

2.12 Coloured brackets and substitutions

Finally, typing for coloured brackets can potentially be broken due to substitutions, as we show in the following example.

Example 2.5 (Substitutions) Let us consider $(\lambda x{:}T.[x]_{c_1}^{T'})v^{c_0}$, a well-typed redex in the colour c_0. Then if we β-reduce it, applying the substitution, we will get $[v^{c_0}]_{c_1}^{T'}$ which is not necessarily typable: though v^{c_0} is typable in c_0, it may not be in c_1. □

Therefore, our rules should not allow arbitrary values to move from one colour to another. We then need to add additional coloured brackets to the only rule creating substitutions at run time, the β-reduction rule:

$$(\lambda x{:}T.e)\, v^c \longrightarrow_c \{x{\leftarrow}[v^c]_c^T\}e$$

where the brackets wrap the substituted value in the current colour, so that it can be safely typed.

The same kind of protection is necessary when we want to push the coloured brackets inside a function.

$$[\lambda x{:}T.e]_{c'}^{T' \to T''} \longrightarrow_c (\lambda x{:}T'.[\{x{\leftarrow}[x]_{c'}^{T}\}e]_{c'}^{T''})$$

Since on the left-hand side the brackets only wrap the function, its argument is expected to have type T and to be typable in the c' colour. However on the right-hand side, the function lives in the outside colour and expects an argument of type T' from the c colour: we need to wrap the variable in the function so that the new argument can be inserted in a safe place where extra brackets are providing a safe interface with the inside implementation.

We have now achieved the presentation of our base system which smoothly extends our previous with structural subtyping and subtyping at unmarshall time.This review and the issues it raised will be useful in the following sections where we introduce our main contributions.

3. User declared subtyping

During the life of a distributed program, parts of it are often corrected or rewritten to fix bugs or add features. If the deployment is decentralised, the updates may not necessarily be rolled out on all hosts at the same time. As a result, a program trying to unmarshall a value of an abstract type may be using a different version of the module from the one that created the value. Should the run-time allow this unmarshall to succeed? More precisely, under what conditions should the subtype comparison of two hash types succeed?

The calculus presented in the previous section prohibits interoperability after version change: any modification of a module's code results in a different hash incomparable to the original. Thus, unmarshalling always throws an exception in the face of such a change. This discipline is prudent: the only way for the compiler to allow such updates and still guarantee abstraction preservation by unmarshalling would be for it to mechanically prove that the changes affected no invariants of an abstract type, an unrealistic task given current theorem proving technology.

While the compiler cannot guarantee abstraction safety after code change, the user can. In this section we extend the calculus to allow user declarations of compatibility between a module exporting an abstract type and an earlier one. The compiler checks that such a declaration is *safe*, i.e. that the representation types are compatible, but relies on the user's good judgement about abstraction preservation.

The addition of this new declaration has many ramifications to compilation, typing, and reduction, which we detail below.

3.1 Motivating example

Before treating the new declaration formally, we present a motivating example: two modules with different structures and signatures for which invariant preservation holds despite their differences. In this example we see how this compatibility is asserted by the user through the new keyword **restricts**.

Example 3.1 (Module upgrade and subtyping) First we define an abstract module of counters:

```
module CounterA =
struct type t = int
       let v = { init = 0 ;
                 incr = fun (x:int)-> x+1;
                 get  = fun (x:int)-> x }
end :
sig type t
    val v : { init : t ;
              incr : t -> t ;
              get : t -> int }
end
```

A new version of this module can then be defined by adding a `decr` function:

```
module CounterB restricts CounterA =
struct type t = int
       let v  = { init = 0;
                  incr = fun (x:int)-> x+1;
                  decr = fun (x:int)-> x-1;
                  get  = fun (x:int)-> x }
end :
sig type t
    val v : { init : t ;
              incr : t -> t ;
              decr : t -> t ;
              get : t -> int }
end
```

The old invariant that any value of `CounterA.t` was a non-negative integer implies the new invariant that any value of `CounterB.t` is an integer. Moreover, the inclusion of the `decr` function gives no more discriminating power to distinguish between values than there was without it.

As shown in the above example, the programmer asserts this compatibility by writing `restricts CounterA`, resulting in `CounterA.t` being treated as a subtype of `CounterB.t`.

Consider now the converse: should a marshalled value of type `CounterB.t` be unmarshallable as `CounterA.t`? No, since the invariant ensured by `CounterA`'s implementation, namely that the counter is always non-negative, would be broken. □

By design, **restricts** introduces only a subtype relationship, not a type equality, so is well suited to this kind of example where the programmer does not want symmetric compatibility. (If two-way compatibility *is* required, that is easily catered for by also using **extends**, which is the symmetric analogue of **restricts**. For the sake of brevity, we limit formal treatment of **extends** to the technical report [4], where we also allow both constructs to accept multiple arguments.)

This precision is in contrast to the blunter `with!` mechanism of Acute which always introduces a run-time type equality, thus forcing the programmer to choose between no compatibility or bidirectional compatibility with no middle ground. Because of this limitation, `with!` is remarkably easy to implement: when a new abstract type is declared to be compatible with a previous one, the compiler just reuses the old hash rather than computing a new one. Our present work pays a much higher implementation price for its flexibility, namely the necessity to propagate the user-declared subtype relation throughout the run-time, as we describe below.

3.2 Syntax, typing, and reduction

Formally, we extend the syntax to include the new declaration as follows (κ is a metavariable referring to a previously defined module, or a hash).

$$\begin{array}{lll} m ::= e & & \text{expression} \\ \quad \mid \ \textbf{module}\, N_U \ \textbf{restricts}\, \kappa = & & \\ \qquad M{:}S \ \textbf{in}\ m & & \text{module declaration} \end{array}$$

The effect of this declarations is to enrich the subtyping relation with new pairs, such as $U.\text{TYPE} <: V.\text{TYPE}$ or $h.\text{TYPE} <: U.\text{TYPE}$.

In order for the judgements to make use of the larger subtyping relation, we introduce a subhashing relation H containing pairs such as $\kappa <: \kappa'$, which decorates the judgements. For example, the old typing judgement $E \vdash_c e{:}T$ now will be written $E \vdash_c^H e{:}T$; the subhash relation intervenes in the type system through the following rule:

$$\frac{E \vdash_c^H \text{ok} \qquad \kappa <: \kappa' \in H}{E \vdash_c^H \kappa.\text{TYPE} <: \kappa'.\text{TYPE}}$$

Most of the old typing rules are transparently lifted with this new annotation: the premises and the conclusion are annotated with a same metavariable H. We consider now the interesting changes.

The first rule that has to be modified is the rule to type module declarations.

$$\frac{\begin{array}{l} E \vdash_\bullet^H T{:}\mathbf{Le}(\top) \\ E \vdash_\bullet^H [T_0, v^\bullet]{:}S \\ E, U(T_0){:}S \vdash_\bullet^{H\cup\{\kappa <: U\}} m{:}T \end{array}}{E \vdash_\bullet^H (\textbf{module}\, N_U \ \textbf{restricts}\, \kappa = [T_0, v^\bullet]{:}S \ \textbf{in}\ m){:}T}$$

Compared to the first version, rule 1 in Section 2.3, we see two main changes. First, the subhash annotation appears in all judgements and is enriched, thanks to **restricts**, in the rest, m, of the program. Second, in order to check the present subhash declaration $\kappa <: U$, as well as subsequent ones in m, we store the representation type T_0 of U in the environment.

Note that the modules in m may now rely on the enriched subhash relation in order to be well typed. As a result, hashes now also rely on a subhash relation for their correctness. Our current hash correctness rule assumes the existence of such a subhash relation without providing it in a computable way. Another solution would be to annotate all hashes with their original subhash relation, but this would risk polluting hash equality and other set operations involving hashes and colours. In any event, in a type-erased implementation, is unnecessary to check hash correctness at run time (section 6).

Since the judgements take a new parameter H, the reduction rules have to follow suit. By analogy with compilation, $H, m \longrightarrow_{\mathsf{m}} H', m'$, expression reduction, $H, e \longrightarrow_c H', e'$, is now similarly annotated. The module reduction rule (when the declared type is abstract) is thus adapted to enrich the current subhash relation with its new **restricts** declaration.

$$H, \mathbf{module}\ N_U\ \mathbf{restricts}\ h_0 = [T, v^\bullet]{:}[X{:}\mathbf{Le}(T''), T']\ \mathbf{in}\ m \longrightarrow_{\mathsf{m}} H', \{U \leftarrow h, U.\text{TYPE} \leftarrow h.\text{TYPE}, U.\text{term} \leftarrow [v^\bullet]_{\{h\}}^{\{X \leftarrow h.\text{TYPE}\}T'}\}\ m$$

$$\text{with} \quad \begin{cases} h &= \mathbf{hash}(N, [T, v^\bullet]{:}[X{:}\mathbf{Le}(T''), T']) \\ H' &= H \cup \{h_0 <: h\} \end{cases}$$

3.3 Propagation of H over the network

In a similar way as the presence of H in the hashes, marshalled values may need the original H in order to be well-typed. Thus we need to annotate the **marshalled** constructor with H so that it can be properly carried over the network.

$$e ::= \ldots \mid \mathbf{marshalled}_{\ c,H}(e{:}T) \qquad \text{marshalled value}$$

The corresponding typing rule will then become:

$$\frac{E \vdash_c^{H_0} \text{ok} \qquad \mathbf{nil} \vdash_{c'}^{H_1} e{:}T}{E \vdash_c^{H_0} \mathbf{marshalled}_{\ c',H_1}(e{:}T){:}\text{BYTES}}$$

and the corresponding reduction rule will be:

$$H, \mathbf{mar}\,(v^c{:}T) \longrightarrow_c H, \mathbf{marshalled}_{\ c,H}([v^c]_c^T{:}T)$$

The logic would be that now, at unmarshall time, we incorporate the received H with the local subhashing relation, so that the received term could be typed. However, this opens a potential security hole: a programmer would be able, intentionally or not, to spread an unsafe abstraction compatibility declaration. To counter this problem each machine on the network could have its own trust policy and refuse some or all incoming H'. A side effect would be that the subtype check in unmarshalling would fail more often with an $\mathbf{Unmarfailure}^{T'}$.

Trust questions are mostly orthogonal to our main concerns, so for simplicity we accept all incoming H', leaving a discussion of the range of potential policies to Section 7.3. We present here the final rule for unmarshalling:

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{\ c',H'}(v^\bullet{:}T){:}T') \longrightarrow_c \begin{cases} H \cup H', [v^\bullet]_{c'}^T & \text{if } \mathbf{nil} \vdash_\bullet^{H \cup H'} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases}$$

Note that the condition $\mathbf{nil} \vdash_\bullet^{H \cup H'} T <: T'$ not only checks the subtyping relation between T and T', but also checks the correctness of $H \cup H'$ and thus of H', ensuring at least the type safety of the result.

4. Partial abstract types

In this section we add expressive power to the calculus by allowing abstract types to be partly revealed without fully exposing their representation types. This is done by enriching signatures to allow the declaration of a supertype, i.e. an upper bound, for each abstract type. This notion, called *partial abstract types* or *bounded existentials*, was first described by Cardelli et al. [3, 10], and implemented in Oberon [14] and Modula-3 [2].

This is our final addition to the calculus, which is now sufficiently rich to have nontrivial instances of subtyping between two abstract types (through `restricts`) and between an abstract type and a concrete one (through partial abstract types). As a consequence, the interplay between subtyping and abstraction boundaries becomes complex and entails two significant changes to the calculus in order to maintain type preservation. First, we make coloured brackets additive: rather than them masking the colour outside, they now simply add to it. Second, we remove the implicit subtyping rule and track explicitly the flow of subtyping coercions as they commute with coloured brackets. (The explicit subtype coercions may be deleted in a type-erased semantics, as we show in the Theorem 6.1 (Correspondence with the erased semantics).)

4.1 Relaxed signatures for partial abstractions

A partial abstract type arises when, instead of giving a completely opaque signature such as `sig type t ... end`, the programmer specifies a known supertype in the following way: `sig type t <: T ... end`. This can be seen as a way to hide parts of a data-type. For example, some of the record fields of an abstract type may be exported and some hidden.

In section 2, the grammar for signatures $[X{:}K, T]$ includes the possibility to define partial abstract types through the kind production $K = \mathbf{Le}(T'')$, though we excluded the "partial" possibility by confining our attention to the case $T'' = \top$. We now lift the restriction on K, allowing $\mathbf{Le}(T'')$ to be used with any type T'' (including $\top$). Consequently, the module reduction rule presented in section 2 gains immediate power by generalising T'' to be an arbitrary type.

As a result, all hashes, whether they are in the current colour or not, may be compared to their declared supertype:

$$\frac{E \vdash_c^H \text{ok} \qquad \vdash h\,\text{ok}}{\begin{array}{l} E \vdash_c^H h.\text{TYPE} <: T'_0 \\ \text{where } h = \mathbf{hash}(N, [T_0, v^\bullet{:}T_1]{:}[X{:}\mathbf{Le}(T'_0), T'_1]) \end{array}} \tag{3}$$

In the following subsection, we examine the potential difficulties that arise in our system because of this addition.

4.2 Semantic consequences

Consider an expression built as a destructor applied to an abstract value. Before the changes introduced in this section, the abstract value could not be completely opaque in the current colour (otherwise the application would be badly typed) so the only possibility was for it to be completely transparent. In this case, the value's coloured brackets could be pushed inwards to allow the destructor to proceed. Now, there is another possibility, namely the abstract type is not transparent in the current colour but is a subtype of a concrete type. We make this precise in the following example.

Example 4.1 (Partial abstract types and coloured brackets) If a partial abstract type $h.\text{TYPE}$ mentions in the signature that it is bounded above by a tuple $T_1 * T_2$ (e.g. by writing $\mathbf{Le}(T_1 * T_2)$ as its kind in the signature), then a projection applied to such an abstract expression should succeed, i.e. the coloured brackets around the abstract value have to be *partially* opened, leading to several difficulties as we now illustrate.

Let $h.\text{TYPE}$ be an abstract type of which we know under the colour c_0 has as a supertype the pair $T_1 * T_2$. We might be tempted to give the following reduction rule:

$$H, \mathbf{proj}_1\,[v^c]_c^{h.\text{TYPE}} \longrightarrow_{c_0} H, [\mathbf{proj}_1\,v^c]_c^{T_1}$$

But this rule would break for two reasons.

First, even though on the left-hand side we can prove under the colour c_0 that $h.\text{TYPE}$ is actually a pair, it is not necessarily so under the colour c which is the current colour *inside* the brackets.

Second, T_1 is a candidate for the annotation on the right-hand side, but it is not the only choice, and may not be the best or even a safe choice. Suppose that from H we learn that $h.\text{TYPE} <: h'.\text{TYPE}$ and that h' happens to belong to $c \cap c_1$, meaning that we can prove that $h.\text{TYPE} <: T'_1 * T'_2$ where $T'_1 * T'_2$ is the concrete representation of $h'.\text{TYPE}$. Then we can safely write T'_1

on the brackets instead of T_1. As T_1' may be more precise than T_1, choosing T_1 may lead to a stuck expression.

□

The two problems shown in the above example lead to two interdependent major changes, which we present in the following subsections, 4.3 and 4.4.

4.3 Additive brackets

Coloured brackets completely mask the expression inside from the type equivalences valid outside them. Moving a destructor inside coloured brackets can thereby lead to an ill-typed term, as we have seen in the example 4.1. However, this situation can only arise when the value inside the coloured brackets is itself an abstract value, i.e. surrounded by another pair of coloured brackets.

One solution is to introduce a bracket merging rule that supersedes the rule (2) presented in section 2.9 to eliminate consecutive coloured brackets in expressions and thus prevent their presence in the value grammar.

$$H, [[v^{c_0}]_{c_0}^{h_0.\text{TYPE}}]_{c_1}^{h_1.\text{TYPE}} \longrightarrow_c H, [v^{c_0}]_{c_0 \cup c_1}^{h_1.\text{TYPE}} \quad \text{if } h_0 \notin c_1 \cap c_0 \text{ and } h_1 \notin c_1 \cap c \tag{4}$$

However, this rule does not fully solve our original problem, which was that type equivalences from outside coloured brackets are forgotten inside them. The subtyping annotations that we introduce in the next subsection face this difficulty. We thus consider a more radical solution.

Our first major change will be to make coloured brackets additive, meaning that the outside equalities are accessible from the inside, as shown in the following typing rule:

$$\frac{E \vdash_c^H T{:}\mathbf{Le}(\top) \qquad E \vdash_{c\cup c'}^H e{:}T}{E \vdash_c^H [e]_{c'}^T{:}T}$$

The inside expression is now typed in the colour $c \cup c'$ instead of c' only: the brackets are now only opaque in one direction.

Some rules are changed in the semantics by this new additivity of the coloured brackets. For example, the brackets are no longer necessary in the β-rule since all the sites where x appears in e have *at least* the type equalities available in c:

$$(\lambda x{:}T.e)\, v^c \longrightarrow_c \{x \leftarrow v^h\} e$$

The rules governing coloured brackets are also smoothly adapted as we see in the following bracket revelation rule:

$$H, [\widehat{v}^{c'\cup c}]_{c'}^{h.\text{TYPE}} \longrightarrow_c H, [\widehat{v}^{c'\cup c}]_{c'}^{T}$$
$$\text{if } h \in c \text{ and impl}(h){=}T$$

Note that values and their colour annotations are changed to reflect the new additivity of coloured brackets. Values that are written $\widehat{v}$ represent values that do not have any coloured brackets as the outermost construct. The side conditions have also to be rewritten, as is the case in the previous rule as well as in the bracket merging rule, for which we now show the definitive versions:

$$H, [[\widehat{v}^{c\cup c_0\cup c_1}]_{c_0}^{h_0.\text{TYPE}}]_{c_1}^{h_1.\text{TYPE}} \longrightarrow_c H, [\widehat{v}^{c\cup c_0\cup c_1}]_{c_0\cup c_1}^{h_1.\text{TYPE}}$$
$$\text{if } h_0 \notin c_1 \cup c \text{ and } h_1 \notin c$$

4.4 Explicit subtyping

Example 4.1 illustrated some of the difficulties caused by the complexity of the subtyping relation: it was unclear *which* type to choose for the brackets on the right-hand side of a reduction pushing a tuple projection inwards.

A solution in that specific case could be to compute a notion of principal tuple supertype. However, the result of this calculation would depend on the current colour and subhashing relation, and is thus quite complicated.

In general, we could use an oracle to non-deterministically choose an upper bound. This would have transferred the obligation of discovering an appropriate bound to the proof of type preservation. However, our presentation to pursue a syntax-directed reduction relation permits us to experiment with executable prototypes of the fully typed semantics (à la Acute).

Thus we adopt the strategy to make the type on the right-hand directed by types already present on the left. This is achieved by switching to a system with explicit subtyping, which is our second major change to address the issues raised in example 4.1.

We can expect that these annotations can be inferred at compile time so the user is not burdened by inserting them manually. Others are added automatically by the run-time time semantics (e.g. at unmarshall time).

For now we add a new syntactic constructor to the syntax of expressions.

$e ::= \ \ldots \ \mid \ {}_{(T_1 <: T_2)}e$ explicit subtyping

The old implicit subtyping subsumption rule is replaced by a new explicit one:

$$\frac{E \vdash_c^H e{:}T \qquad E \vdash_c^H T <: T'}{E \vdash_c^H {}_{(T<:T')}e{:}T'}$$

Most typing and reduction rules are not changed since explicit subtyping only aims at syntactically witnessing the implicit subtyping that was already present.

However in a few situations some type annotations have to be inserted to match the presence of implicit subtyping. This is the case of the **unmar** rule where we expect the two types to be related by subtyping, so we introduce the corresponding annotation on the right-hand side:

$$H, \mathbf{unmar}\,(\mathbf{marshalled}_{\,c',H'}(v^\bullet{:}T){:}T')$$
$$\longrightarrow_c \begin{cases} H \cup H',\ {}_{(T<:T')}[v^\bullet]_{c'}^{T} & \text{if } \mathbf{nil} \vdash_c^{H\cup H'} T <: T' \\ H, \mathbf{Unmarfailure}^{T'} & \text{otherwise} \end{cases}$$

This is also the case in the module reduction rule. There is implicit subtyping between the (implicit) type of the value field in the structure and its declared type in the signature. Thus we need to make the type explicit in structures so we can construct the subtyping node at compile time, as follows:

$M ::= \ [T, v^\bullet{:}T']$ structure

The module reduction rules now explicitly insert the subtyping node:

$$H, \mathbf{module}\, N_U = [T_0, v^\bullet{:}T_1]{:}[X{:}\mathbf{Eq}(T_0'), T_1'] \ \mathbf{in}\ m$$
$$\longrightarrow_{\mathsf{m}} H, \{U.\text{TYPE} \leftarrow T_0', U.\text{term} \leftarrow {}_{(T_1<:\{X\leftarrow T_0'\}T_1')}v^\bullet\}m$$

$$H, \mathbf{module}\, N_U \ \mathbf{restricts}\, h_1 = [T_0, v^\bullet{:}T_1]{:}[X{:}\mathbf{Le}(T_0'), T_1'] \ \mathbf{in}\ m$$
$$\longrightarrow_{\mathsf{m}} H \cup \{h_1 <: h\}, \sigma\, m$$

where

$$h = \mathbf{hash}(N, [T_0, v^\bullet{:}T_1]{:}[X{:}\mathbf{Le}(T_0'), T_1'])$$
$$\sigma = \{U \leftarrow h,\ U.\text{TYPE} \leftarrow h.\text{TYPE},\ U.\text{term} \leftarrow [{}_{(T_1<:\{X\leftarrow h.\text{TYPE}\}T_1')}v^\bullet]_{\{h\}}^{\{X\leftarrow h.\text{TYPE}\}T_1'}\}$$

In these two rules we can observe the introduction of the two explicit subtyping annotations that give the value $v^\bullet$ the declared type of the signature.

Finally, we consider the interaction between explicit subtyping and coloured brackets. When the former meets the latter, we swap them around in the following way:

$$H, {}_{(T'<:T'')}([\widehat{v}^{c'\cup c}]_{c'}^{h.\mathrm{TYPE}}) \longrightarrow_c H, [{}_{(T'<:T'')}\widehat{v}^{c'\cup c}]_{c'}^{T''} \quad \text{where } h \notin c$$

Note that moving the subtyping annotation inside the brackets depends on brackets being additive: without additivity, the subtyping node which is valid in colour c would not be valid anymore in colour c'; with additivity, it is valid since we have the colour $c \cup c'$ on the inside. We also remark the introduction of new elements in the value grammar. The grammar is now divided in three parts: v^c represent all values in colour c; $\widehat{v}^c$ represents the values that do not have coloured brackets as their outermost construct; $\widehat{\widehat{v}}^c$ are the subclass of $\widehat{v}^c$ values that do not have a subtyping annotation either as their outermost construct.

When explicit subtyping annotations move inside coloured brackets, they become exposed to the additional colours annotating the brackets. As a result, the types in the subtyping annotation may be rewritten accordingly with hashes being replaced by more concrete types. We omit these bookkeeping rules here and refer the reader to the technical report [4].

Let us return to example 4.1 in the light of the changes made in this and the previous subsections. Now, when a projection meets coloured brackets we know that the latter's type is a product since there is no more implicit subtyping subsumption. Of course, the situation in the example of a projection applied to an abstract expression could arise, but only if there is an intervening explicit subtyping node between the projection and the brackets. Thanks to the rules introduced, the subtyping node can be pushed inside and propogated as far as necessary to produce an explicit tuple, as desired.

5. Theorems

In this section we summarise the formal results about the global safety of our static and dynamic semantics. All the following results with their proofs can be found in [4].

Our calculus satisfies the properties of type preservation for machine reduction (i.e. compilation), expression and network reductions.

Theorem 5.1 (Type preservation for machine, expression and network reduction)
If $\vdash_\bullet^H m\!:\!T$ and $H, m \longrightarrow_{\mathsf{m}} H', m'$ then $\vdash_\bullet^{H'} m'\!:\!T$.
If $\vdash_c^H e\!:\!T$ and $H, e \longrightarrow_c H', e'$ then $\vdash_c^{H'} e'\!:\!T$.
If $\vdash n$ ok and $n \longrightarrow n'$ then $\vdash n'$ ok.

The progress property also holds for machine, expression and network reduction.

Theorem 5.2 (Progress for compilation) If $\vdash_\bullet^H m\!:\!T$ then either m is an expression or it reduces under $\longrightarrow_{\mathsf{m}}$. Moreover, compile time reduction is terminating.

Theorem 5.3 (Progress for expressions) If $\vdash_c^H e\!:\!T$ then one of the following holds :

- e is a value, i.e. there exists a v^c such that $e = v^c$
- e can reduce, i.e. there exist e' and H' such that $H, e \longrightarrow_c H', e'$
- e is waiting for I/O, i.e. there exist $CC_{c_2}^c$ and v^{c_2} such that $e = CC_{c_2}^c.!\,v^{c_2}$ or $e = CC_{c_2}^c.?$
- e has thrown an exception, i.e. there exist $CC_{c_2}^c$ and T' such that $e = CC_{c_2}^c.\mathbf{Unmarfailure}^{T'}$

Concerning networks, our theorem has to include the different possible outcomes from the communication.

Theorem 5.4 (Progress of networks) If $\vdash n$ ok then one of the following cases holds:

- n is stopped, i.e. there exists $n_{()}$ and n_{fail} such that $n \equiv n_{()} | n_{\mathrm{fail}}$.
- n is waiting to input, i.e. there exists $n_{()}$ and n_{fail} and $n_?$ such that $n \equiv n_{()} \mid n_{\mathrm{fail}} \mid n_?$
- n is waiting to output, i.e. there exists $n_{()}$ and n_{fail} and $n_!$ such that $n \equiv n_{()} \mid n_{\mathrm{fail}} \mid n_!$
- n can reduce, i.e. there exists n' such that $n \longrightarrow n'$

where $\equiv$ is a standard network structural congruence and

$n_{()} ::= \mathbf{0}$	null
$\quad n_{()} \mid n_{()}$	parallel composition
$\quad ()$	unit
$n_{\mathrm{fail}} ::= \mathbf{0}$	null
$\quad n_{\mathrm{fail}} \mid n_{\mathrm{fail}}$	parallel composition
$\quad CC_c^\bullet.\mathbf{Unmarfailure}^T$	dead
$n_? ::= n_? \mid n_?$	parallel composition
$\quad CC_c^\bullet.?$	waiting to input
$n_! ::= n_! \mid n_!$	parallel composition
$\quad CC_c^\bullet.!\,v$	waiting to output

Both compile-time machine reduction and run-time expression reduction are deterministic (network reduction is not, of course):

Theorem 5.5 (Determinism of machine reduction) Reduction of machines is deterministic, i.e. if $m \longrightarrow_{\mathsf{m}} m_1$ and $m \longrightarrow_{\mathsf{m}} m_2$ then $m_1 = m_2$ and both reductions use the same rule on the same redex.

Theorem 5.6 (Determinism of expression reduction) Expression reduction is deterministic, i.e. if $H, e \longrightarrow_c H', e'$ and $H, e \longrightarrow_c H'', e''$ then $e' = e''$ and $H' = H''$ and both reductions use the same rule on the same redex.

6. Implementation challenges

Now that we have presented our calculus and its properties, we discuss the challenges to integrating it into a programming language.

6.1 Erasure

In most ML-languages, all type annotations are erased at run-time thanks to a theorem stating that the behaviour of the erased version of a correctly typed program is similar to the original. Our system satisfies a similar result.

First, let us describe a proposed erasure operation, which turns out to be almost, but not quite, what we need. An *erased term* is a term where all type and subtyping annotations and all coloured brackets have been forgotten, with the notable exception of the type annotations on **mar**, **marshalled** and **unmar**. The reductions in the erased world are formed by applying erasure to both sides of each reduction rule in the typed world, throwing out any rules whose RHS becomes identical to the LHS after erasure.

One reduction rule, however, does not behave well in the erased world, namely the garbage-collection of the unused fields of a record:

$$H, {}_{(\{l_1:T_1;\ldots;l_j:T_j\}<:\{l_1:T_1';\ldots;l_i:T_i'\})}\{l_1 = v_1^c;\ldots;l_j = v_j^c\} \longrightarrow_c H, \{l_1 = v_1^c;\ldots;l_i = v_i^c\}$$

The difficulty in applying erasure to this rule is that the explicit subtype disappears, thus we have no way of knowing locally *which* fields should be garbage collected.

One solution could be to find a way to safely add some explicit garbage collection nodes to the erased-world syntax. However, this is difficult to do. The problem is that explicit subtyping nodes in the typed world evolve during reduction: it is possible for the source and target types to start as completely abstract, e.g. ${}_{(h.\mathbf{Type}<:h'.\mathbf{Type})}\, e$, and only later become manifestly record types when the expression is used in a different colour context (as explained at the end of Section 4.4). Thus it is unclear how erasure should translate abstract subtype nodes so that they later evolve into the correct garbage collection — a problem we leave for future work.

Therefore, we chose a different strategy here. We weaken our erasure function to an erasure relation $\mathrm{erase}(e, \underline{e})$ which relates any term e to its erased version $\underline{e}$, in which every record $\{l_1 = v_1^c; ...; l_j = v_j^c\}$ in e is related to one in $\underline{e}$ that may contain additional junk value fields: $\{l_1 = \underline{v}_1; ...; l_j = \underline{v}_j; ...; l_k = \underline{v}_k\}$. We write $\underset{\mathrm{nb}}{\longrightarrow}$ for the expression reduction relation in the erased world.

The theorem then asserts that reduction paths in both worlds follow a strict correspondence, namely each reduction in the typed world is mapped to at most one reduction in the erased world; and each reduction in the erased world is reflected by a series of one or more reductions in the typed world.

Theorem 6.1 (Correspondence with the erased semantics) If $\vdash_c^H\ e{:}T$ and $H, e\ \longrightarrow_c\ H', e'$ and the erased-world expression $\underline{e}$ is such that $\mathrm{erase}(e, \underline{e})$, then there exists a $\underline{e}'$ such that $H, \underline{e} \underset{\mathrm{nb}}{\longrightarrow}{}^{?} H', \underline{e}'$ and $\mathrm{erase}(e', \underline{e}')$.

If $\vdash_c^H\ e{:}T$ and there is a $\underline{e}$ such that $\mathrm{erase}(e, \underline{e})$ and if $H, \underline{e} \underset{\mathrm{nb}}{\longrightarrow} H', \underline{e}'$ then there exists e' such that $\mathrm{erase}(e', \underline{e}')$ and $H, e \longrightarrow_c^+ H', e'$.

Finally, note that the presence of garbage fields in the erased world is not disturbing: by examining the type of a whole program at compile time it is possible to statically infer which fields in the final value are garbage. Even if, as we discuss above, it turns out not to be possible to systematically eliminate garbage fields in all intermediate values of the computation, it seems likely that we can derive record narrowings in enough cases to render the problem moot for real examples. We will investigate this in future work.

6.2 Hashing

Thus far in our presentation we have considered hashes, such as $\mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Le}(T),\, T'])$, to be algebraic constructions from which we can extract all the subparts, such as the structure $[T_0, v^\bullet]$ or the representation type T_0. However, in an implementation, we would like to compile hashes to small fingerprints generated from the application of an uninvertible, pseudo-injective function, such as SHA-1. As a result an implementation can only compare hashes for equality.

We now examine the erased-world semantics to show that this implementing strategy of compiling hashes to fingerprints is safe, modulo a small change.

In the erased semantics, there is only one place in which the parts of a hash are examined, namely the subtype check performed at unmarshall time. Note first that the subtype check potentially engages all of the rest of the type system, *except* the "expression has type" rules. Therefore we have to systematically consider every place in the type system where a hash may need to be deconstructed. We show that each case, except the last, may be avoided; the last case motivates a simple change, leading to our final proposal.

- The subtyping check at unmarshall time is done in the empty colour. None of the premises of any type rule (except the excluded "expression hash type" ones) has a different colour from the conclusion's. Thus we never use type revelation in the erased semantics, obviating the need to know the representation type of a hash.
- In all type derivation trees, any hash h that appears in a judgement is ultimately checked for correctness, i.e. the leaves of the derivation tree consist of subproofs of the form $\vdash\ h\,\mathrm{ok}$. These subproofs look into the structure of h. However, our hypothesis is that all information received from the network is generated by a well-typed program and transmitted by a reliable network, thus we may assume that all hashes are correct. As a result, we can dispense with the hash correctness checks: as far as hash correctness is concerned, hashes may be completely opaque, as desired.
- In all type derivation trees, the subhash annotation H is also checked for correctness. The check consists or each pair of types present in H in verifying the compatibility between the two representation types and the two announced kinds. However, for the same reason as the previous item, i.e. because of our security assumptions, this check is superfluous.
- Finally, hashes for partial abstract types are subtypes of their declared supertypes via the rule 3 from section 4.1. The supertype is not necessarily available locally and we thus need access to the declared supertype in every hash type.

As a consequence, we conclude that we can safely compile a hash $h = \mathbf{hash}(N, M{:}[X{:}\mathbf{Le}(T),\, T'])$ to a pair consisting of a fingerprint of all the components and the declared supertype T, i.e. we compile h to $(\mathit{sha1}(N, M{:}[X{:}\mathbf{Le}(T),\, T']), T)$, the first component being opaque and the second being readable. As a result, hashes may be implemented compactly and, thanks to the fingerprint function, are independent of the size of the source code contained in the structure M.

6.3 Decidability of subtype comparison

Our semantics requires a subtype check at unmarshall time to compare the received type to the expected type. This check is non trivial because of the variety of ways subtyping can be derived, namely as a transitive chain of steps; each step may be an instance of structural subtyping between record types, of subhashing from pairs of hashes in the H relation, or of the passage from a hash to its declared supertype.

We have an algorithm for checking subtyping, which we believe is terminating, sound, and complete, through we have not yet proved this. While we would be delighted to successfully complete this proof, the algorithm would be useful even if it were not complete. Such a situation exists throughout language and proof tools, for example the undecidable higher-order unification algorithm [8] of the Twelf theorem prover, which is highly useful in practice.

A false negative in our subtype check is easy to handle: the unmarshall rule would just raise $\mathbf{Unmarfailure}$. In fairness, however, incompleteness *would* be frustrating for the programmer: unlike in a theorem prover or compiler where a false negative can be overcome by the user tweaking the code or adding hints, it is difficult in a distributed system for a failed unmarshall at one end of the network to be debugged from the other. Proving the completeness of our algorithm is thus important future work.

7. Future work

We now outline some of the principal areas of future research that flow from the ideas presented in this paper.

7.1 Distributed parametricity

Coloured brackets demark abstraction boundaries originating from the modules in user code. In our semantics we track the propagation of coloured brackets throughout the execution of a concurrent composition of separately-compiled programs. We know that their propagation is safe, as shown by the type preservation theorem, thus allowing us to conclude informally that abstraction preservation holds.

Nonetheless, it remains an open problem to formally state an abstraction preservation theorem in a distributed setting that would be an analogue to Reynolds' [11] parametricity result for a single program. Ideally such a theorem would cover the two properties we defined at the beginning of the paper. For the first, *invariant preservation*, we want to state formally that any value successfully unmarshalled at an abstract type satisfies the invariant properties expected of the type. In other words, unmarshalling never circumvents the invariants guaranteed by the receiver's abstractions. For the second, *secrecy*, any two values of an abstract type that are observationally congruent remain observationally congruent after being transmitted and unmarshalled on another machine.

7.2 Integration with existing implementations

Two main implementations supporting abstraction-safe marshalling and hash types currently exist: Acute, a stand-alone interpreter with many experimental features (versioning, rebinding, thread freezing), and Hashcaml, a smooth extension (for hashes and marshalling) of Ocaml. A challenging task will be to integrate this paper's proposals in both.

Hashcaml [1] is a patch to Ocaml and thus supports most of the latter's features. In order to integrate our work with Ocaml, we would need to adapt our subtyping technology to deal with the existing Ocaml structures that already support subtyping, namely objects and polymorphic variants. Both of these have complex typing rules that would demand careful treatment.

The integration task would be easier in Acute [12], where subtyping is presently absent. This would be good setting to experiment with Acute's and our competing mechanism for module upgrade (`with!` vs `restricts`) as well as the secrecy and authentication issues discussed below.

7.3 Security

In section 4.4, we presented our unmarshalling rule. In the rule, the receiver merges the incoming subhash relation with its own. This policy of completely trusting the sender to transmit an invariant-preserving subhash relation represents a potential security hole. In response, our unmarshalling rule could be modified to accept or reject any incoming subhash relation. The question therefore arises: under which conditions should it do so? We could require the sender to provide evidence of the wholesomeness of the subhash relation. This evidence could be lightweight, e.g. a cryptographically signed certificate attesting that the subhash relation's provenance is a trusted peer, or more powerful and heavy, e.g. an oracle allowing the receiver to construct a formal proof of invariant preservation for the subhash relation (à la proof carrying code).

In the same way, our current system assumes that all received hashes are correct, i.e. the declared supertype actually corresponds to the abstract type implementation (which remains hidden). We also assume that a received value has the claimed type declared by the $\mathtt{marshalled}$ construction. However, in these two cases, any intruder can break the system by sending wrong hash-types or ill-typed marshalled values. As in the previous paragraph, we may consider adding various kinds of certificates to ensure we only communicate with trusted peers or require proof witnesses from untrusted ones.

Finally, secrecy as an abstraction property can be strengthened by encrypting abstract values (perhaps using hashes as keys) in the style of Pierce and Sumii [13].

Acknowledgments

We wish to thank our colleagues Andrew Appel, Jean-Jacques Lévy, Gilles Peskine, Peter Sewell, and Francesco Zappa Nardelli for fruitful discussions and useful comments on drafts of this paper. We also thank Frank Pfenning for his detailed help concerning our question about the Twelf literature. We are grateful to the anonymous referees whose close reading led to many corrections and improvements.

References

[1] J. Billings, P. Sewell, M. Shinwell, and R. Strniša. Type-safe distributed programming for OCaml. Submitted for publication. `http://www.cl.cam.ac.uk/users/pes20/hashcaml/`.

[2] L. Cardelli, J. E. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, Austin, Texas, 1989.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[4] P.-M. Deniélou and J. J. Leifer. Abstraction preservation and subtyping in distributed languages. Technical report, INRIA Rocquencourt, 2006. Available from `http://pauillac.inria.fr/~denielou/`.

[5] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6):1037–1080, 2000.

[6] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, pages 123–137, 1994.

[7] 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://pauillac.inria.fr/~leifer/research.html`.

[8] S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[9] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[10] G. Plotkin, M. Abadi, and L. Cardelli. Subtyping and parametricity. In *Proc. of 9th Ann. IEEE Symp. on Logic in Computer Science, LICS'94, Paris, France, 4–7 July 1994*, pages 310–319. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[11] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[12] P. Sewell, J. J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proceedings of ICFP 2005: International Conference on Functional Programming (Tallinn)*, Sept. 2005. To appear.

[13] E. Sumii and B. C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003. Extended abstract appeared in *14th IEEE Computer Security Foundations Workshop*, pp. 256–269, 2001.

[14] N. Wirth. The programming language oberon. *Software Practice and Experience*, 18(7), July 1988. The Language Report.