PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

Towards Secure Distributed Computations

Felipe Zipitría

Supervisor: PhD. Gustavo Betarte

Montevideo, Uruguay

2008

Towards Secure Distributed Computations
Zipitría, Felipe
ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 08-22
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, noviembre de 2008

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Master thesis

Towards Secure Distributed Computations

Author: Felipe Zipitría
fzipi@fing.edu.uy

November 17th, 2008

Supervisor: PhD. Gustavo Betarte
gustun@fing.edu.uy

Advisors: PhD. Gustavo Betarte, PhD. Tamara Rezk
gustun@fing.edu.uy,Tamara.Rezk@sophia.inria.fr

Readers: PhD. Alfredo Viola, PhD. Daniel Fridlender, PhD. Javier Baliosián

To Patricia and Melina

Many thanks to: Gilles Barthe, Tamara Rezk and all the team at INRIA Sophie-Antipolis, Daniel Fridlender, Gustavo Betarte and Eduardo Grampín for their major contributions to this work; Alfredo Viola (Tuba) and Javier Baliosián for accepting to be part of this; Jorge Sotuyo, Alejandro Blanco, Marcelo Rodríguez, Julián Adib, Horacio Pérez and the URI teamwork. Last, but not least, PEDECIBA Informática and all my colleagues at the Instituto de Computación.

Contents

List of Figures

Listings

1. Introduction

"Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.
A: The definitions are almost always wrong." — Pierce 2002

1.1. Motivation

Since the beginning, computers have helped humans do lots of different computations. We always had mathematical problems — each one bigger than the other — that we found how to represent in such a way that computers could help us solve. With the need to solve larger problems, arises the dependency on other parties to do complementary computations. Communications with other computers developed to form networks of computers. Proportionally with this increase in communications, computing power has been upgraded, by using remote computers to solve complex problems cooperatively.

Nowadays, there are plenty of networks that work in a cooperative way and form what we know as *grids* of computers. These grids serve a lot of purposes, and they are used with good results for intensive calculation, because the joined computing power aids in solving any kind of complex functions. To cope with these new requirements and facilities, programming languages had to evolve into new paradigms, including facilities to do distributed computing in a straightforward way.

Andrews (And82) gives a definition for *distributed programming language*: it is a "concurrent language in which process interaction is based on message passing rather than shared variables". Using this approach, the underlying network is made transparent to the programmer. How the network is accessed and where processes are located should be avoided as a concern for the programmer.

Two key components are contained in a distributed programming language: processes and channels. A process is a sequential program that contains local variables and statements. A channel is a communication path that connects processes, and processes interact by sending messages to and receiving messages from the channels that are accessible to them. These channels can also be typed for communications.

Most programming languages provide the programmer with a way of classifying their variables by using a type system. A type system is a component of a programming language which keeps track of the types of variables and all other expressions in a program. These type systems are used to determine if programs are well behaved with respect to some properties.

For example, type-safety is a property concerned with *how much* a programming language protects its own abstractions. Type-safety is usually viewed as two properties of the semantics of the programming language: *progress* and (type-) *preservation*. Intuitively, this means that terms approved by the type system should never lead to an incorrect state, where computation cannot continue. Not every programming language has this property, and in the ones that are distributed it is more difficult to conceive.

This happens because on a single computer type safety is usually guaranteed by the compiler. Nevertheless, in the context of distributed programming, the problem of type-safety has additional difficulties. One particular defined type, defined in a host, may have a different implementation in another, different invariants, or the same implementation but another name. Another important property to take into account is abstraction-safety.

Normally, a program is decomposed into a series of modules, each of these modules is used to represent an abstraction of the problem, once identified. Again, types have a fundamental role in this procedure. To make these modules more reusable in other conditions, the introduction of Abstract Data Types (ADTs) allows programmers to create and export the particular functions they want. These data types are used as an interface between an exported implementation and a real implementation. They are also created to preserve some properties. The defined operations on this type are the only means of interacting with values of this particular type. Only the operations defined inside this type have access to the information.

If the programming language semantics and the type system guarantee that the encapsulation provided by type abstraction can never be breached, then the language is called abstraction safe. Note that abstraction safety is a stronger property than mere type safety (but generally it cannot exist without the latter). On a single computer it is usually guaranteed by the compiler. Finally, if we want to have a distributed programming language, it will be desirable to preserve abstraction safety throughout the entire distributed system.

However, this changes radically when we begin to transmit data over the network. Data can be modified, lost, or attacked. After the creation and utilisation of such languages, an aspect remaining to be introduced is the *security* properties of these computations. The security properties of languages that execute on a single host are hard to maintain. We must take increased precautions when dealing with lots of hosts and complex networks. First of all, distributed programming languages must achieve correctness in their own abstractions: they must satisfy type-safety and abstraction safety, among other properties.

There have been efforts towards securing the execution of remote code. There is a technique named Proof Carrying Code (NL96), where code sent to another process has a proof that complies with some specification. This proof is a formal safety proof, called certificate, and shows that this code complies with certain specification of safety rules. This proof can be used, with the help of a proof checker, to verify compliance of the certificate to the safety rules.

How can values of abstract types be secured, in the context of a distributed programming language? We propose the use of a novel technique, called Proof Carrying Results (BP06). This technique is based on Necula's proof carrying code. Basically, the result of some computation comes equipped with a certificate, or witness, that can be used with abstract types.

Let's suppose that we want to have secure computation involving abstract data types. One host asks for some computation to a remote host. This computation is performed using abstract types. After that, the value is returned to the caller. Is it possible for the caller to know if the value complies with the invariants of the abstract type? Instead, if the value comes with a witness that the computation was performed *correctly*, the caller can verify this witness and *know* that the value was generated in a good way.

There are many ways for a host to check the correctness of this value, by using the associated witness. On one hand, it can be verified using a custom made checker for the particular abstract type. This way is prone to errors, in the creation of a correct checker for this values. On the other hand, a proof checker can be used as a reliable tool for checking correctness. It is more general than the other particular solution, because it can be used for any type (abstract or not).

Throughout this thesis work, we will show how to add the PCR technique to a distributed programming language. The supporting infrastructure for the technique is introduced along with it. For checking the values and associated witnesses produced by some host, we use a proof checker for a precise and reliable verification. In the following description we continue with more detailed aspects of the work done.

1.2. Description

Functional programming is a paradigm that treats computation as the evaluation of mathematical functions. Functional programming languages implement the concepts introduced by this paradigm. Usually, they are modelled using λ calculus, but other variants exist. Along this line we have languages like ML, Haskell and (*Pure*)Lisp. This work will have its focus on ML-like languages.

As part of the evolution in grid computing, some functional programming languages were adapted to handle these new grid requirements. To be used in distributed contexts, the calculi had to be extended with new paradigms. Theoretic support for concurrent and distributed programming was conceived. For concurrent programming the π calculus was created, and this formalism was extended for mobility on the *ambient* calculus. From these approaches, new functional languages were created. Examples of concurrent programming languages are Pict (PT97), occam-pi (INM84) and Concurrent Haskell (JGF96). In the case of distributed programming languages, we can mention Nomadic Pict (Woj00; SW99), Alice (Kor01)and Acute (SLW^{+}04b).

This work is concerned with correctness and safety in distributed languages, with focus on ML-like languages and the properties they have. To this aim, we will use a language called Acute. This language was born for doing research in distributed programming, and was created as a joint effort of the University of Cambridge and INRIA Rocquencourt.

In Acute we have modern primitives for interaction between cooperating programs. Its main features involve: type-safe marshalling of arbitrary values by using two primitives, *marshal* and *unmarshal*. As names are important in a distributed context, type names are hashed to uniquely identify types across the entire distributed system. In addition to this, type names are generated (freshly and by hashing) to ensure that type equality tests suffice to protect the invariants of abstract types. To ensure that name equality tests suffice for type-safety of associated values, expression-level names generated, e.g., values carried on named channels. When some value has been sent to another host using the marshal primitive, it is rebounded to local resources in a controlled dynamic way. Finally, there is a technique that packs threads and mutexes (called *thunkification*) that can be used to support computation mobility.

These features are a large part of what is needed to produce a typeful distributed programming. This language has been used to enable sophisticated infrastructure development. For example, there are implementations of Nomadic Pict and JoCaml as simple Acute libraries.

Acute has powerful properties: type and abstraction safety are guaranteed along the distributed system. Among the main design choices in Acute, was the focus on typing and naming issues. But there are problems, clearly documented, when unmarshaling values of abstract types. If the value marshalled is of an abstract type, the representation could not be available at unmarshal time. Therefore, in this case: how can it be assured that this value was generated by a well-behaved Acute run-time?

What happens when there are entities that can tamper with data transmitted between hosts? The presence of active adversaries in the networks changes the scenario. If this situation occurs, safety can be no longer guaranteed. In this case, an additional check is that the marshalled value is a well-formed representation of something of that type. The implementation of that type could be used for checking types at unmarshal time. But this implementation is not available, and especially in the case of abstract types. This limits the language to a simple decision to handle this case: to work only in a trusted scenario, or to marshal only values of concrete types.

Throughout this work we will try to overcome this particular limitation of the Acute language. We will extend the language to support marshalling of abstract types in a non-trusted scenario. For this purpose, we will introduce the PCR technique into the language, and a supporting infrastructure for it that will end increasing the properties of safety in a distributed context. The next section summarises the contributions of this work.

1.3. Contribution

This thesis has contributed in three main aspects:

- an infrastructure has been defined and implemented for supporting the technique of proof carrying results,
- the Acute distributed programming language has been extended, with a mechanism that permits the exchange of abstract values in a certified way, and
- for doing the verification of the results, this infrastructure has been connected with the COQ proof checker.

An important aspect of the definition of the infrastructure, is that it can be used in other situations where there is a need for using the proof carrying results technique and not only in this particular case. It is independent from the language, and also from the proof checker used.

For the Acute language, a solution is proposed for a problem previously detected. This solution uses the infrastructure defined here, and its implementation is available for further testing (Zip08). This was achieved by increasing abstraction safety when exchanging values of abstract types, extending the *marshal/unmarshal* primitives of the language itself. In order to achieve this, we used the technique of proof carrying results, where values transmitted to other hosts carry a witness that proves the computation has been done in a correct way. For the process of verifying that the witness certifies these transmitted values, we introduced a proof checker.

The proof checker is used as a reliable tool to prove that the certificates can be verified correctly. In this step we used the COQ proof assistant, but without interacting with it; only as a verification tool. Any other trustworthy proof checker can be considered. The proofs are made over these values of abstract types and their properties, carried in a certificate.

1.4. Methodology

The Acute language was initially conceived for trust, when transmitting values over the network. We wanted to prove if abstraction safety in the Acute distributed language could be broken by tampering with data. Therefore, having a proof of concept and discovering how easy or difficult it was seemed a good starting point.

In due course, we constructed a simple example that effectively showed us that it was easy to circumvent abstraction-safety and change the data to break abstractions (Appendix D). Understanding all the subtleties of the marshalling process including its semantics and implementation was a must for this step. The marshalling process is an important feature, and includes concepts for maintaining abstractions. The technique relies on the concept of Coloured Brackets (LPS$^+$03). This is technically delicate (and not needed for implementations, which can erase all brackets) but provides useful clarity in a setting where abstraction boundaries may be complex, with abstract types shared between programs. The semantics also preserves the internal structure of hashes. The relevant parts and dependencies in the implementation are distributed in various files.

In addition to this, we identified the precise place where we would plug-in our infrastructure, in order to have a more secure language. After finding this spot, we created the necessary binding for connecting the language to our verification infrastructure. The binding with the COQ (dt08a) proof assistant required the study of the different ways of communication with it. Now we proceed to describe the organisation of this document.

1.5. Organisation

Chapter 2 will begin by explaining what type systems are, and what properties they have. We will focus on two of these properties: type-safety and abstraction-safety, and how they are viewed in a distributed setting. In addition to this, we will present some notions about distributed computations and primitives in this context, namely, marshalling and naming. Finally, we will introduce some security concerns in a distributed environment, and discuss how they affect the presented concepts of type systems.

In Chapter 3 we will introduce proof carrying results. The chapter begins with a description of this technique, and continues with its supporting algorithmic base: certification algorithms. There are many examples shown there of algorithms that can be used with certification in mind, the lack of such algorithms for other cases is also mentioned.

After presenting the context of this work in detail, in Chapter 4 we present our take on attempt to improve the safety of the Acute programming language, by means of using the technique presented and introducing an infrastructure for it.

There is a full case study in Chapter 5, where the infrastructure presented is used to solve a problem: how to certify prime number generation.

The conclusions and future work can be viewed in Chapter 6. The source code of the extensions programmed into the Acute language is mentioned in Appendix A.

2. Type-safe Distributed Computations

This chapter begins introducing type systems in programming languages, and two major properties that we will be referring to throughout this document: (type-) safety and (type-) abstraction. There are many points of view regarding type systems in general, and the literature on them and their related properties is extensive. We will be following two authors, Luca Cardelli (Tuc97) and Benjamin C. Pierce (Pie02), throughout the definitions and terms introduced here. We will continue with concepts of distributed computations, with focus on marshalling. We shall also analyse the preservation of these properties in the context of a distributed programming language, and in the presence of active adversaries.

2.1. A review of type and abstraction safety

This review begins with a basic explanation of type systems. Before further analysis, we must talk about errors in programming languages, and how these errors are handled by them.

We define an execution error as an error that appears at run-time, and was unexpected by the programmer. Not surprisingly, to prove that we will not have execution errors is a difficult task. If we treat the lack of execution errors in a program as a property, then we say that a language is *type sound* when that property holds for all the possible runs of a program that can be expressed within this language. To have a proof that a language is type sound requires a tremendous amount of work and a lot of careful analysis. This is why a discipline emerged to carry out formal analysis, classification, and study of type systems. The formalisation of type systems needs to elaborate a common language with precise notations and definitions, and additional formal proof that can give, at last, confidence in these definitions.

A type system is a component of a typed language which keeps track of the types of variables and all other expressions in a program. These type systems are used to determine if programs are well behaved. We will introduce types in 2.1.1, and the notion of program behaviour in section 2.1.3.

Besides, we must note that not all the programming languages have a type system. These languages are called *untyped languages*, and the ones that have a type system are called *typed* languages. In the next section we present a definition for *type.*

Ultimately, one of the major aims of having a type system in a programming language is to prevent the occurrence of execution errors (Car96). This informal statement motivated a lot of study of type systems, but it needs to be extended and made precise. The next sections will introduce terminology and define commonly used terms in this context.

2.1.1. Types

It is common practise for a programmer to introduce variables when programming. The *type* of a variable can be viewed as an upper bound for the range of values this variable can have during the execution of a program. For instance, if we have a variable v of type *int* it is supposed to have only integer values during the execution of a program. Also, if v has type *int*, then the integer expression *nonzero(v)* has a sensible meaning in every run of the program. We call languages where we can give variables nontrivial types *typed languages.*

In addition to this, a language that has a universal type that contains all values is called *untyped* language. Now we proceed to discuss type safety.

2.1.2. Type-Safety

Resuming the topic of errors, there are errors that unexpectedly abort the execution of a program. These are called trapped errors. Furthermore, there are errors which are not detected until some time later, and are responsible for unpredictable program behaviour. These are called untrapped errors. There are numerous examples of both cases: a division by zero is a common example of a trapped error; accessing a valid memory address in an invalid way by exceeding the bounds of an array is a common example of an untrapped error.

When we have a program that does not cause untrapped errors to occur we call it *safe.* Languages are called safe when all of its programs are safe. Consequently, these safe languages are good because they leave out the problems of errors that can cause unpredictable behaviour without being noticed. Untyped languages must perform checks at run-time to enforce safety. Typed languages can check the programs statically, and if a program which could be unsafe is found, it can be rejected. These languages can also perform run-time checks to reinforce safety.

	Typed	Untyped
Safe	OCaml,C#,Java	Perl,Python,LISP
Unsafe	C	Assembler, BCPL, Forth

Table 2.1.: Safety and typing

This table shows some languages and their stated safety. For example, the C language is typed but unsafe, and Perl language is untyped and safe. Most languages have more or less programmers who like each of their properties; because they write programs without too much work, or because they are used to them.

Summing up, type-safety is concerned with *how much* a programming language protects its own abstractions. Intuitively, this means that well-typed terms should never be in an incorrect state, where computation cannot continue. Type-safety is usually viewed as two properties of the semantics of the programming language: *progress* and (type-) *preservation.*

We now introduce program behaviour, which is usually related to type safety.

2.1.3. Behaviour

There is a subset of the execution errors from section 2.1, that we will call forbidden errors. These errors include untrapped errors, in addition to a subset of the trapped errors. It can be said that a program has a good behaviour if it does not cause any forbidden errors to occur. In particular, a well behaved part is safe.

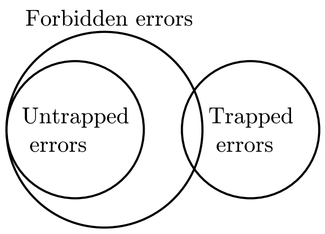

Figure 2.1.: Execution error sets

A language where all programs have good behaviour is called strongly checked. Therefore, when talking about a given type system, this holds for a strongly checked language:

- “*no untrapped errors occur: this is the safety guarantee*”
- *“no single trapped error in the forbidden subset occurs*”
- “*the programmer should avoid other trapped errors*”

Good behaviour can be enforced by typed languages by performing static checks at compile time to prevent unsafe programs from running. The checking process is called type-checking, and the algorithm that performs these checks is called type-checker. If some program happens to pass the type-checker without errors, is said to be well typed. If this program does not pass this stage is called ill-typed, which may stand for an ill-behaved program. Examples of languages which are statically checked are OCaml, C# and Java.

Untyped languages can enforce good behaviour in a different way, by checking all array access or all divisions by zero. This checking process is known as dynamic checking. Examples of this kind of languages are Perl, Python and LISP.

If a language happens to be statically checked, this does not imply that at run-time it can execute all programs without minimal checking. Consider, for example, the classical problem of array bounds, which may be not known at compile time. The problems that arise at run-time should be checked to have a more robust language.

2.1.4. Abstraction safety

Normally, a program is decomposed into a series of modules, each of these modules is used to represent, once identified, an abstraction of the problem. Again, types have a fundamental role in this procedure. To make these modules more reusable in other conditions, the introduction of Abstract Data Types (ADTs) allows programmers to create and export the particular functions they want. These data types are used as an interface between an exported implementation and a real implementation. The ADTs will also be created to preserve some properties. The defined operations on this type are the only means of interacting with values of this particular type. Only the operations defined inside this type have access to the information. In (JHM73), there are two important properties that type abstraction presents:

Authentication "*It is forbidden for any other except the implementation to construct values of abstract types. This brings the possibility of having enforced invariants that would be too difficult to maintain otherwise*".

Secrecy "*Only the implementation is allowed to inspect values of abstract types. This enforces loose coupling between the possible clients of the abstract type. If something is changed in the implementation, but the interface to the clients remains unchanged, there will be no need for the clients to change their code*".

If the programming language semantics and the type system guarantee that the encapsulation provided by type abstraction can never be breached, then the language is called abstraction safe. Note that abstraction safety is a stronger property than mere type safety (but generally it cannot exist without the latter). On a single computer it is usually guaranteed by the compiler. But this changes radically when we begin to transmit data over the network. Data can be modified, lost, or attacked. Finally, if we want to have a distributed programming language, it will be desirable to preserve abstraction safety throughout the entire distributed system. This topic is covered in the next section.

2.2. Concepts of distributed computations

From this section on, we refer to some properties of distributed computations and how programming languages have been adapted to support these properties in a transparent way. We introduce some problems found in distributed systems, and ways of dealing with those problems. These problems include global naming of types across a distributed context, or how to pass values between different hosts in a network.

To this end, we will introduce a language called Acute, created for doing research in distributed programming languages. We will see the main design points of it, with special focus on some primitives. In addition to this, we will discuss security in a distributed context: more precisely, how to maintain some properties in the context of active adversaries in our network.

2.2.1. Type naming

In a distributed setting, names of types do not suffice for declaring type equality on different hosts. We must deal with abstract types, and that is when problems arise. If we have a non-abstract module type there could be at least some comparison for equality (name or structural equality on types (SH00; BS79)). When working with abstract types, the representation may not be available for performing this check.

For example, in one host A we may have an abstract type called $Counter_A$ which counts integers, and in another host B an abstract type with the same name $Counter_B$ (we put a sub-index for doing the distinction) which is used for counting odd integers only. Despite the bad selection of the type name (it could have been $OddCounter$), we cannot rely on type names only for exchanging values of abstract types.

There are different proposals to solve this, and the more general problem of using abstract data types in distributed contexts (BHII87; PMMR06). The solution we will describe here was introduced by (LPS$^+$03) and it is to use hashes as type names. By hashing module definitions at compile time, global module type names can be created for abstract types, even between different compilations on different hosts. This hash of module definitions consists of the module itself, and all the dependencies the module has. By using this technique, hashes can be literally viewed as types. They are included in the type grammar as well. We will use Listing 2.1 as input for an intuitive description. When a program is compiled, whenever an abstract type Seed.t is found, it will be replaced by the hash of the definition of the module Seed. These type hashes can not be used by a programmer: they are only an abstraction in the type grammar of the language. At compilation time, hashes are inserted when needed. We will show examples of how this is achieved.

In this proposal, we need to modify the standard operational semantics of a language. Standard abstract types operational semantics substitute types and operations, as computation goes on. For this case, we need to maintain additional information for type preservation theorems that are added to this semantics, and to prove that static and dynamic type equality match. We will see a construct to help preserve this additional type information, called *coloured brackets* (GMZ00), in the next section.

2.2.2. Coloured brackets

By using hashes as global names, we are in fact introducing a problem we did not have before. When compiling source code, we must take into account that we cannot substitute types of values belonging to abstract data types with their hash representation. This happens because we lose information to use for type-checking, and we can be exposing concrete type representation details of this abstract type. This small example shows how this happens:

We have a module, called Seed, which provides an abstract seed to be used as input for cryptographic algorithms. We call the signature of the module *SeedSig*, and the structure is called *SeedStruct*. Now we proceed to give a brief description of the compilation process, because it is important for understanding when and how hashes are introduced.

Throughout this process we will use $\longrightarrow_c$ as the symbol for indicating that a compilation was done. This is because compilation is like a relation between source code and compiled code, and because the process could be non-deterministic. First, the source code is type-checked for errors. After that, in places where we have a $TYPE$, there will be reductions that substitute t with corresponding hashes $h.t$. This process is made in

```
module Seed = struct
   type t
   val init: unit -> t
   val rand: int -> t
   val get: t -> int
  end;;

(fun s:Seed.t -> ()) (Seed.init)
```

Listing 2.1: Example Seed module

the signature, in the structure of the module, and also in all modules referenced by it. Furthermore, all hashes are generated in a deterministic way. This ensures that on every machine the same code gets the same hash value h.

Having the compilation process clear, we proceed to compile the example module in Listing 2.2.

```
⟶c (* compilation *)

(fun s:h.t -> ()) 43

where
h = hash(Seed, (module Seed=SeedSig:SeedStruct),t)
```

Listing 2.2: Example Seed module, compiled

The expression obtained after traditional compilation is not typable, because h.t is an abstract type and its implementation should be opaque outside the Seed module. This happens because we have lost information due to compilation. With the knowledge we have, we cannot prove that 43 of type *int* has type h.t as super-type.

To solve this problem we introduce the concept of coloured brackets (GMZ00). The author's aim was to treat terms accessed from different hosts (*principals*) in a different way, based on the knowledge they have of the implementation of some abstract type. Later, this was extended by (LPS+03) to cope with this case.

Basically, coloured brackets wrap abstract terms. This is used to separate where we can show the internal details of abstract types and where we must show this term as an abstract type. These terms are written $[e]_c^T$, where T is the outside type and c the *colour*. This colour is basically a single hash that tells us which type is equivalent to this abstract type. There is an empty colour, that we denote as $\bullet$.

Back in our example, in listing 2.3 we show the application of coloured brackets.

With this additional information, now $[43]_h^{h.t}$ can be typed with type $h.t$ by the type-checker. Also, inside the brackets, 43 can be associated with its natural type *int* and with h.t, by using the *colour* h. This happens because the hash h contains *all*

```
(fun s:h.t -> ()) [43]_h^{h.t}


where
h = hash(Seed, (module Seed=SeedSig:SeedStruct),t)

and
SeedStruct =
  type t=int
  ...
```

Listing 2.3: Colored brackets

the information for its calculation: in particular, it contains what is needed to do the association of t to *int*, in the *structure* of the module Seed. This is a sort of revelation of the information carried with each hash. Naturally, this behaviour must be taken into account by the type system to prove different equalities based on the set of hashes that can be revealed in each particular term.

In particular this should be applied whenever we have to transmit a value of an abstract type to other hosts or processes in our network. Coloured brackets are used to carry the information of the equivalent types that this abstract type has. We will introduce marshalling in the next section.

2.2.3. Marshalling

The name has its roots in terminology used by the designers of Modula-3 (CDJ$^+$89a; CDJ$^+$89b) (amongst others), who introduced the term *marshal* for shipping data to other places in a self-contained form. We will use the following definition:

Marshalling is the process of gathering data and transforming it into a standard format. In this way, the data can transcend network boundaries or be used from a persistent storage in a later time. In order for a value of some type to be moved around a network, it must be converted into a data stream that corresponds with the packet structure of the network transfer protocol. When the data is transmitted, the receiving computer converts the marshalled data back to its original form.

As an example, this process must be done when passing the output parameters of a program written in one language as input to a program written in another language.

Many programming languages have support for this process. In Table 2.2 we summarise some of these languages, where this procedure can be referred to in different ways: “pickling”, “serialise” or “marshal”. There are some authors that make slight distinctions between these terms, but we will use their intended meaning as if they were interchangeable.

Language	Primitive for packing	Primitive for unpacking
MSH	export-clixml	import-clixml
PHP	serialize	unserialize
OCaml	Marshal.to_string	Marshal.from_string
Ruby	Marshal.dump	Marshal.load
Haskell	Data.Binary.encode	Data.Binary.decode
F#	BinaryFormatter.Serialize	BinaryFormatter.Deserialize
Smalltalk	storeBinaryOn	readBinaryFrom
Python	pickle.dump	pickle.load
Common Lisp	(with-standard-io-syntax (write obj stream))	(with-standard-io-syntax (read obj stream))

Table 2.2.: Languages and representations

2.2.4. Exchanging data between hosts: to trust or not to trust

It is common for hosts to send and receive data in a network. An example is shown in figure 2.2. There are variations in the trust that we have between hosts. A priori, it is not the same to connect to a server in your local area network, or to connect to an unknown host on the Internet.

There are hosts (or entities) that are malicious, and can act as adversaries preventing us from achieving a particular goal. Specifically, *active adversaries* are the ones that tamper the messages transmitted between hosts. This may be intercepting, replying or modifying these messages.

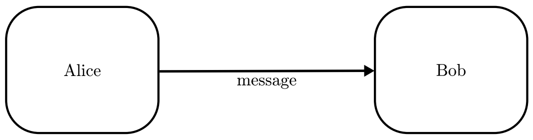

Figure 2.2.: Transmission

Figure 2.3 shows an interception, made by host named *Trudy*.

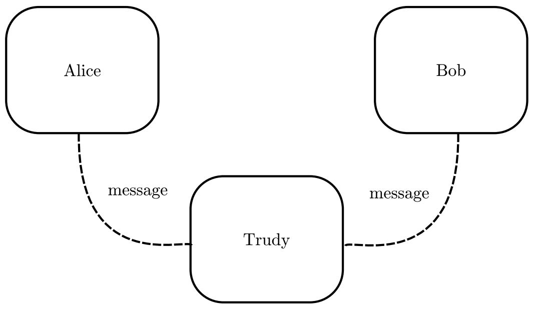

Figure 2.3.: An adversary intercepting a message

This simple example show that adversaries could modify messages in a network. We will not show *how* this is achieved, because it is not within the scope of this work.

2.2.5. Type safety with active adversaries

In a distributed setting, it is more difficult to maintain type safety properties: hosts on the network may not have the same representations for the same types, or there can be adversaries on the network that are actively introducing errors or altering security properties that affect type safety directly. Figure 2.4 shows an example of this problem:

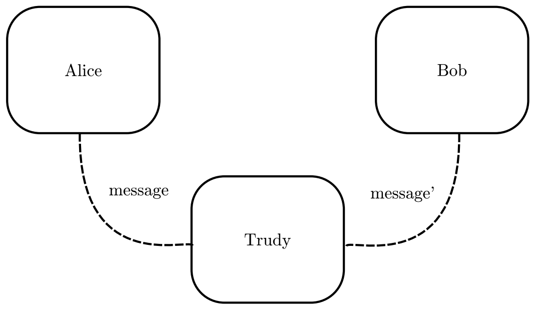

Figure 2.4.: Altering message

The adversary modifies types and/or values that are travelling along the network between two hosts so type safety can not be guaranteed anymore.

2.3. A distributed language: Acute

The Acute language is based on an ML-like core, with extensions to support distributed development in high-level languages. The focus was on what has to be added to a higher-order typed language, with call-by-value semantics traditional in ML languages, to support distributed development and interaction. The language is the result of a collaborative work between the University of Cambridge and INRIA Rocquencourt. The topics in this research included some interesting and key properties of distributed languages, and some results were added as features of the language. It is important to note that the implementation of this language is a prototype, made to remain as close as possible to the semantics of the language. The features added focus on naming and identity in a distributed setting, and for the sake of this, includes among others properties:

- Type-safe distributed interaction, permitting marshalling of values of any type. This key feature will be extended on 2.3.2.1.
- Type names generated by hashing the definition of this type, or in a fresh manner, by generating a unique id for this type. This generation can be done at compile time, for example.

- Expression-level names generated to be certain that name equality test suffices for type safety of associated values.
- Thunkification of threads and mutexes to support computation mobility (migration of threads).
- Rebinding of marshalled values to local resources where values are to be unmarshalled. This feature allows the association of the values to local types, or to be evaluated in a different context.

We will broaden some of these features in the next sections, emphasising on the ones that affect type and abstraction safety directly. Next we continue with a brief description of the syntax and semantics of the language.

2.3.1. Syntax and Semantics

The complete type grammar of the language is in Figure 2.5. There is shown the (non-standard) principal expressions, and the form of modules. In this case, h is a module name, hash or freshly generated, t is a field of extern type identifier, M_M is a pair of internal/external module identifiers, and MK is a constant string. Types which are a subindex are inferred in general. Moreover, the last three type expressions are only present in the semantics, not in source code.

$$\begin{aligned}
T ::= {} & \textbf{int} \,|\, \textbf{bool} \,|\, \textbf{string} \,|\, \textbf{unit} \,|\, \textbf{char} \,|\, \textbf{void} \,|\, T_1 * \cdots * T_n \,|\, T_1 + \cdots + T_n \,|\, T \to T' \,| \\
& T\,\textbf{list} \,|\, T\,\textbf{option} \,|\, T\,\textbf{ref} \,|\, \textsf{exn} \,|\, M_M.t \,|\, t \,|\, \forall t.T \,|\, \exists t.T \,|\, T\,\textbf{name} \,|\,|\, T\,\textbf{tie} \,| \\
& \textbf{thread} \,|\, \textbf{mutex} \,|\, \textbf{cvar} \,|\, \textbf{thunkifymode} \,|\, \textbf{thunkkey} \,|\, \textit{thunklet} \,|\, h.t \,|\, n
\end{aligned}$$

$$\begin{aligned}
e ::= {} & \ldots \,|\, \textbf{marshal}\, e_1\, e_2 : T \,|\, \textbf{unmarshal}\, e\, \textbf{as}\, T \,| && \textsf{marshalling} \\
& \textbf{fresh}_{\mathbf{T}} \,|\, \textbf{cfresh}_T \,|\, \textbf{hash}(T, e_2)_{T'} \,|\, \textbf{hash}(T, e_1, e_2)_{T'} \,|\, n_T \,|\, h.x \,| && \textsf{naming} \\
& \textbf{namecase}\, e_1\, \textbf{with}\, \{t, (x_1, x_2)\}\, \textbf{when}\, x_1 = e \to e_2\, \textbf{otherwise} \to e_3 \,| \\
& \textbf{create\_thread} \,|\, \ldots \,|\, \textbf{thunkify} \,|\, [e]^T_{eqs} && \textsf{threads}
\end{aligned}$$

$$\begin{aligned}
& sourcedefinition ::= \\
& \quad \textbf{module}\, \textit{mode}\, M_M : \textit{Sig}\, \textbf{version}\, \textit{vne} = \textit{Str.withspec} \,| && \textsf{modules} \\
& \quad \textbf{import}\, \textit{mode}\, M_M : \textit{Sig}\, \textbf{version}\, \textit{vce}\, \textit{likespec}\, \textbf{by}\, \textit{resolvespec} = Mo \,| && \textsf{import} \\
& \quad \textbf{mark}\, MK && \textsf{marks}
\end{aligned}$$

Figure 2.5.: Acute syntax

The Acute static type system for source programs is based on an OCaml core and a second-class module system, with singleton kinds for expressing abstract and manifest type fields in modules. Module initialisation can involve arbitrary computation. The definition of compilation describes how global type- and expression-level names are constructed, including the details of hash bodies. This is technically delicate but provides useful clarity in a setting where abstraction boundaries may be complex, with abstract types shared between programs. In the implementations, all brackets can be erased. The semantics preserves also the internal structure of hashes. This too can be erased in implementations, and hashes and fresh names can be implemented with literal bit-strings (e.g. 160-bit SHA1 hashes and pseudo-random numbers), but is needed to state type preservation and progress properties. The abstraction-preserving semantics makes these rather stronger than usual.

In the next subsection we continue extending the language features.

2.3.2. Features of Acute

Here we discuss some language features, focusing on the properties relevant for this work. There are other properties that we do not extend here, but can be revisited in the main technical report (SLW+04b).

2.3.2.1. Type-safe distributed interaction

For doing type-safe (un)marshalling, the language brings two primitives, marshal and unmarshal. Below there is an example of their syntax:

$$\text{e} ::= \ \ \ldots \mid \text{marshal } e_1\ e_2 : \text{T} \mid \text{unmarshal } e : \text{T} \mid \ldots$$

These expressions are used to send values along untyped communication channels. The marshal/unmarshal primitives can be used in many ways, for example, between cooperating programs or to perform remote computations.

The marshal primitive has two parameters: the first one, e_1 is an expression of type string used as a mark. This mark, which will be explained next, is used for dynamic rebinding. The second parameter, e_2 is an expression of type T.

2.3.2.2. Dynamic rebinding

A particular value can be sent over the network, or saved for using in a later time by using the marshal primitives. After this procedure, different modules can be available when we want to use this value at unmarshal time. The language introduces an interesting feature which is that this value can be rebounded to local resources or shipped along.

This is different from other approaches that convert local resources to distributed references for referencing this resource remotely. In that case there is more traffic associated to each of the indirect attempts to access to this reference. This language does not have explicit support for distributed references. Instead, that option is left to be programmed with the basic primitives of communication.

The programmer can select which values are packed and sent over the network, and which ones should be rebounded remotely to modules available at the other end of the communication. This is achieved simply by putting a special primitive named **mark** in the source code of our programs, that determines which module definitions are shipped with any marshal operation, and which are rebounded to local resources (BHS+03).

Every module in Acute is a sequence of definitions, plus marks interspaced between the modules.

An example source code showing the usage of *mark* can be viewed in listing 2.4.

```
module M1: sig val y:int end = struct let y=6 end
mark "MK"
module M2: sig val z:int end = struct let z=3 end
IO.send( marshal "MK" (fun ()-> (M1.y,M2.z))
     : unit->int*int)
--
module M1: sig val y:int end = struct let y=6 end
module M2: sig val z:int end = struct let z=4 end
((unmarshal(IO.receive()) as unit->int*int)() ,M2.z)
```

Listing 2.4: Dynamic rebinding of modules

In this example we can see two programs. In the first program, there are two modules $M1$ and $M2$. After the definition of $M1$ there is a *mark* "MK" which divides the part that will be sent ($M2$) from the one that will be rebound remotely ($M1$). When the other program is executed, there is already a definition for module $M1$ and for module $M2$. This makes the receiver program use its module $M2$. The result of the execution of the second program will be $((6, 3), 4)$.

This provides an additional strong feature to distributed programming. For instance, if we are constructing a program to do routing in a network, when a value is sent to another router, it can be rebound to the local resources, and be used in a completely different instance of the same program (which measures distance to neighbours in a different way).

2.3.2.3. Concurrency

The work on concurrency is very mature, using mutexes and condition variables. Also, there is a procedure for the migration of groups of threads (called *thunkification*) and its associated mutexes and condition variables on these mutexes to be carried to another host or to be saved on magnetic media for later use.

2.3.2.4. Versions

In this language, versions can be grasped as rough approximations to semantic module specifications. Versions have a specific language within Acute. This language will be attached to modules. Furthermore, we need some way of restricting which version we are willing to import in a module: there is a language of version constraints. In addition to this, it should be possible to specify chains of imports, each one with its own version. We also need a relation that gives us what versions satisfy our import specification.

This is a feature which permits, among other things, to do partial migration of modules in a distributed environment, by allowing to specify versions of software modules. The code in listing 2.5 shows a simple example of a version used in a module definition. This permits the restriction of versions with a specification of the desired version included in the definition of the module. This specification is named *withspec.*

```
module Show: sig val v end = version 2.0
——
import Show version 2.*
```

Listing 2.5: Version example

Then it should be easy to define which version code we are willing to use in our program by using an import statement.

2.3.3. Acute and abstraction safety

Our main motivation to use this language is because it is supposed to be type-safe in a trusted setting.

When there are no adversaries in our system, it is explicitly detailed on (SLW$^+$04a) that unmarshal is not only type-safe, but also abstraction safe. This happens because of the coloured brackets technique presented in section 2.2.2, and because type primitives like $marshal(e : T)$ and $unmarshal(e : T')$ are closed. This means that they do not have free module identifiers or type variables. This step aids to have a simple representation of marshalled values as byte-strings, which can be carried over the network in an easy way. Finally, type checking at unmarshal-time will be simple with string equality.

By using this technique we must observe that standard operational semantics for abstract types must be changed, because it is common to forget about abstraction, by doing substitution of types and operations.

To be more robust against active adversaries, we have to add something to prevent the values being marshalled to be tampered with. In this case it should be mandatory to check that the marshalled value is a well-formed representation of the type declared. The authors of Acute propose to use ASN.1 or XML instead of strings for exchanging data and to have a PKI infrastructure or to use PCC to secure the information sent over the wire.

Our proposal will be around a similar idea of this, but using a novel technique, introduced in the next chapter. The objective is to have a more robust language against active adversaries.

3. Proof Carrying Results

In this chapter we explain a novel technique called Proof Carrying Results. We begin with an introduction showing its basis. We continue with its motivation, description and formalisation. Furthermore, we add the description of a special kind of algorithms called *certifying algorithms*, which are a fundamental support for this technique.

3.1. Introduction

The technique introduced here has its base on previous work made by (NL96) in proof carrying code (PCC). In that work, Necula put formal mechanisms to solve the problem of remote code execution in a behaviour-controlled way.

We have a scenario in which two parts exchange information (see Figure 3.1): one consumer of code, and one producer who sends code to it. The code sent to a remote consumer has a certificate, which is a formal safety proof, showing that this code complies with certain specification of safety rules. The consumer must have a way of validating this proof, with the help of a proof checker, to verify compliance of the certificate to the safety rules.

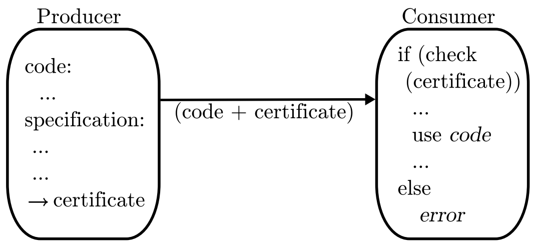

Figure 3.1.: Proof carrying code example

There are many areas where this technique can be applied. For example, mobile code, in the form of applets, for Internet browsers or mobile phones. Moreover, it would be interesting to use it for updating software remotely. The code to be updated carries evidence that proves it is safe to execute it on the local machine. Then the consumer could certify that the code complies with some specification.

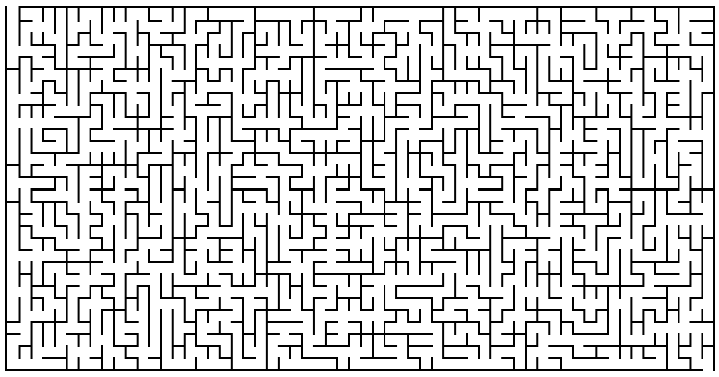

Figure 3.2.: A maze

A simple example that represents how this technique works can be shown using a maze. In Figure 3.2 there is a simple maze. Can the reader tell if there is an exit to this maze, in brief time?. Without additional information, this can be a difficult task. But if we show a path or trace that can be followed until the maze is solved, it is easy and very fast to determine the exit point. Figure 3.3 shows the maze solved, using a trace of square dots. The example is known as Necula's labyrinth, and is commonly used by Necula to show this PCC property.

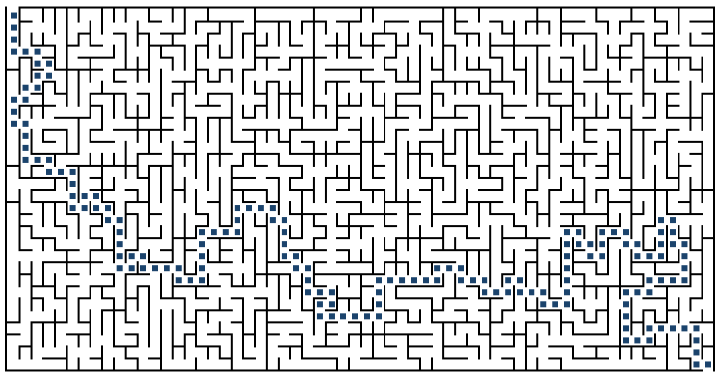

Figure 3.3.: Follow the dots to solve the maze

The trace itself can be viewed in this way: when we have a maze to solve; to try to solve it should be more difficult than to follow the dots to the exit. Therefore, the certificate that is carried along with the code in PCC should be like the dots in the maze. In this way, the consumer only has to use this trace to verify the compliance to the safety rules.

When checking the certificate provided, the checking process must provide a reliable checker. For this purpose, we introduce the trusted computing base (TCB) (U.S85). The TCB is everything in a computing system that provides a secure environment. This includes the operating system and its provided security mechanisms, hardware, physical locations, network hardware and software. It is assumed that the trusted computing base has been or should be tested or verified. The checker used in this process must be contained in the TCB.

In the next section we introduce a technique called proof carrying results, which is based on Necula's work on PCC.

3.2. Proof carrying results

This technique was introduced in (BP06). The authors propose that some of the concepts introduced in PCC be reused. As in PCC, we have two parts, but the exchange of information is applied in a different scenario. Instead of a code consumer, we have a results consumer: some host consumes a remote function, and the host that makes the computation sends its results with additional information that allows the consumer to validate this result.

The approach is that a consumer host sends an untrusted part some computation to be done remotely. The untrusted part then returns the result of this computation, with a *certificate* that the computation has been done in a correct way.

Certificates provide additional data to check the correctness of computations. They can contain *witnesses*, which are generated in the process of computing, and *proofs*, that state properties about the result. These witnesses are like a trace of the computation made, and are closely related to the final result returned. This trace is similar to the one in the previous maze example. Proofs, which can be part of the additional data carried, can be there to establish properties about the result itself, or about the witnesses.

Figure 3.4, introduces a basic interaction between hosts. There, some host sends a value a to an untrusted part. It asks for the computation of a function f with the value sent.

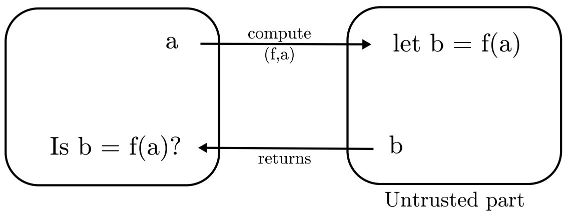

Figure 3.4.: Basic PCR approach

Afterwards, the untrusted part obtains b as the result of applying the function f to a. Then b is returned to the initial host. The question now is: how can this host be sure that b is effectively the result of applying function f to a?.

This technique tries to answer that question by using a verification approach, as in PCC. It also shares some important properties with PCC, which are summarised here (from (BP06)):

Based on verification instead of trust. *Indeed, PCR focuses on mathematical properties of the result rather than on its origins. In particular, it does not require the existence* *of a global trust infrastructure (although as for PCC it can be used in combination with cryptographic based trust infrastructures).*

Resource-aware *Indeed, PCR technology advocates for succinct certificates that can be checked efficiently and aims at avoiding performing costly computations.*

Transparent *While PCR uses certifying algorithms, which may be difficult to program or consuming to run, it requires result consumers only to check certificates, which is fully automatic, and not to build these certificates.*

General, flexible and configurable *Formal frameworks for certificates, are very expressive, and lend themselves to verification, thus it is in principle applicable to a wide range of algorithms. Furthermore, it is possible to specialise certificates and certificate checkers for each particular algorithm.*

Figure 3.5 describes a more complex example of a PCR interaction. This begins when host A wants to obtain a prime number from host U_1, sending a seed for doing calculations. Suppose that U_1 returns, along with number p, a vector $\vec{w}$ of witnesses which can be used to verify that p is effectively a prime number.

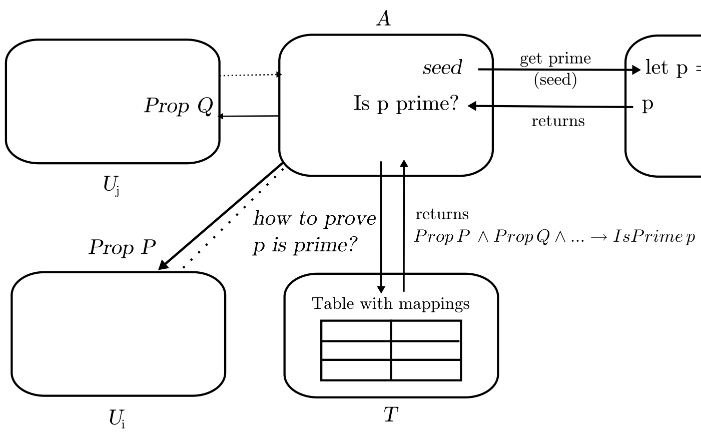

Figure 3.5.: Complex PCR

Suppose that this vector contains a number of integers, which also have the property of being primes. Then host A can use the additional vector $\vec{w}$ as aid to do this verification. An extra step is added here: host A asks another host T (in this case a trusted one) for the conditions to verify the desired property, since T knows *how* to prove whether a number is prime. This information can be in the same host A, but in this example it was taken from a remote source. This T gives A the way to obtain a proof of primality of p. With this information, host A begins to ask other hosts about proofs (these are the computations) of these properties. In this way there is a distribution of the proof that p is prime.

Formalisation

We have a function f and we want to delegate its computation, using a as argument:

- $f(a)$ is delegated to an untrusted party, $f \in A \rightarrow B, a \in A$
- $b \in B$ is the expected value

To verify that this value is the value we sent to compute, we must have some function *check*:

$$check_f \in A \times B \rightarrow bool \mid \forall(a,b) \in A \times B, check_f(a,b) = true \Rightarrow b = f(a)$$

In general, PCR allows the untrusted part to provide additional data H intended to ease the checking process. Thus, one may have a checker function $check_R \in A \times B \times H \rightarrow bool$ such that $check_R(a,b,h) \Rightarrow R(a,b)$. In this way, the functional specification f is generalised to an input-output specification R.

This technique challenges the traditional algorithms, because more information could be needed from them, not only their result. In the next section we introduce a new kind of algorithm that is the foundation of this approach.

3.3. Certifying algorithms

The term was introduced in (KMMS03), and a more general approach was given in (MEK$^+$05). An algorithm or program is certifying when, along with the result it was supposed to give, it returns a certificate or proof that this result is indeed the correct for the given input. This is a pragmatic approach to program correctness. In general, the caller of a function f has no means of knowing if the result obtained from the invocation of this function f with a as a parameter is b. This program has to believe in this result. Certifying algorithms produce not only the desired result a, but also additional output. This additional output is called a *witness* or a certificate for this computation, and is denoted w.

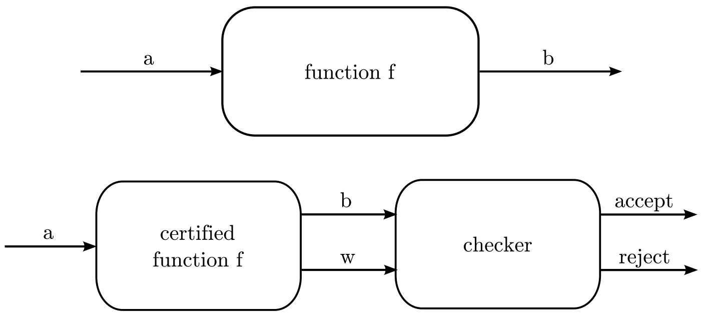

Figure 3.6.: Certified function

Figure 3.6 shows the difference between a traditional function f and its certified counterpart.

Concepts

Certifying algorithms have not been formally defined yet (instead of this, notions about their formalisation are given in (MEK+05), chapter 8). Their definition is beyond the scope of this work. Thus, the concepts used here are notions rather than actual definitions. We will be using a function $f : A \rightarrow B$, and start with the notion of a witness predicate W for f. We call a ternary predicate $W(x, y, w)$ *witness predicate* for f if the following conditions hold:

1. w deserves the name witness, e.g., if $W(x, y, w)$ holds then $y = f(x)$ and if $y = f(x)$ then there is a w with $W(x, y, w)$.
$$\forall x, y\, (\exists w\, W(x, y, w))\, iff\, (y = f(x)) \tag{3.1}$$
2. Given x, y, and w, it is trivial to decide whether $W(x, y, w)$ holds. The authors list a number of possible formalisations for this sentence; we will choose "There is a simple logical system, in which we can decide whether $W(x, y, w)$ holds".
3. The witness property is easily verified, e.g., the implications 3.2 and 3.3 have elementary proofs.
$$\forall x, y, w\, W(x, y, w) \;\rightarrow\; (y = f(x)) \tag{3.2}$$
$$(y \neq f(x)) \rightarrow \nexists w\, W(x, y, w) \tag{3.3}$$

Next, we outline the notion of a certifying algorithm for a function f. "We call Q a certifying program for f if there is a witness predicate W for f such that for all inputs x, Q computes a triple (x, y, w) with $W(x, y, w)$". Note that this implies $y = f(x)$.

There are important points to consider. Here, we will only introduce them. Proofs can be found in (MEK+05):

- *"Monte Carlo Algorithms resist Certification."*
- *"Every deterministic program has a certifying counterpart (with only a constant overhead in running time). "*

Now we summarise some examples of algorithms that check their work.

3.4. Result certification examples

There are a number of examples in the literature, here we present only some of them. It is not our aim to be exhaustive. This list tries to include examples of different types of problems that can have certified counterparts. The algorithms in these examples can be used for result certification.

Extended GCD

The *extended* GCD (Greatest Common Denominator) algorithm is an example of a basic checker. In plain GCD we do not have a simple checker, but this extension allows us to have an efficient one, which is not difficult to compute.

$$GCD(x,y) = d \,\text{where}\, d \mid x \wedge d \mid y \wedge (\forall d', d' \mid x \wedge d' \mid y \Rightarrow d' \mid d) \tag{3.4}$$

In equations 3.4 and 3.5 symbol | means *divides*. The extended GCD algorithm further computes u and v such that $d = ux + vy$.

$$ExtendedGCD(x,y) = (u,v,d)\, where\, d \,|\, x \ \wedge d \,|\, y \ \wedge d = ux + vy \tag{3.5}$$

Such u and v constitute the certificate; in this case, the certificate always exists and implies the minimality of d, i.e., it guarantees that the quantified part of the specification is true. One may notice that it is simple to obtain d once u and v have been provided. *This shows that in a result certification framework, the hint or certificate may become more important than the actual result. A plain result provided by an untrusted party has no value, whereas a certificate does.*

Prime numbers

There are a number of algorithms to test if a given number is prime (see (Ber04)). This is known as *primality test.* The Pocklington criteria (Poc16) is one of these algorithms used to verify if a number is prime. This criteria is defined in this way:

Pocklington's criteria. Given a natural number $n > 1$, a witness a, and some pairs $(p_1, \alpha_1), \dots, (p_k, \alpha_k)$, it is sufficient for n to be prime that the following conditions hold:

$$p_1 ... p_k \, are\, prime\, numbers \tag{3.6}$$

$$(p_1^{\alpha_1} ... p_k^{\alpha_k}) \ \mid \ (n-1) \tag{3.7}$$

$$a^{n-1} \equiv 1 (\mathsf{mod}\, n) \tag{3.8}$$

$$\forall i \in \{1, ..., k\}\, \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) = 1 \tag{3.9}$$

$$p_1^{\alpha_1} ... p_k^{\alpha_k} > \sqrt{n} \tag{3.10}$$

In equation 3.7, symbol | means that all numbers must be divided by $n-1$, in equation 3.8 mod is the modulo operation, and in equation 3.9 gcd is the greatest common divisor. There are slight variations on what the requirements for Pocklington's criteria are, but we will use this for convenience. The numbers $a, p_1, \alpha_1, \dots, q_k, \alpha_k$ constitute a Pocklington certificate.

From the point of view of PCR, there are two simple but central observations to make:

- given n, it requires much more computation power to determine the numbers of the Pocklington certificate, than to check that these numbers verify the conditions 3.7-3.10 above;
- checking primality of a natural number n with certificate $p_1, \alpha_1 \ldots, p_k, \alpha_k$ and a can be reduced to
  - conditions 3.7-3.10 can be checked by purely numerical computations. Verification of condition 3.9, may make further use of the PCR infrastructure by requiring an extended GCD certificate to be provided.
  - verification of condition 3.6 can be done recursively. Note, however, that Pocklington's criteria cannot prove that 2 is prime, so that another form of certificate is required for 2 (there can be no certificate at all, since 2 is trivially prime).

There are other kinds of algorithms which can be used for primality testing: in the case of elliptic curve cryptography, there are some algorithms that suggest that they can be used with this purpose; in the field of combinatorics there are such algorithms as well. In these cases, certificates seem to be difficult to build or check. In (Ber04) the author details certificate construction and checking for many algorithms.

These algorithms of primality checking are interesting in the context of securing distributed computations: we could have devices which cannot compute a large prime number by themselves; instead they could run a simple remote procedure call, and a simple check before that to certify primality.

SAT

SAT is a decision problem, known as the satisfiability problem. The satisfiability problem consists of a logical propositional formula in n variables and the requirement to find a value ($true$or $false$) for each variable that makes the formula true. One important thing about this satisfiability problem is that it is NP-complete.

It is of central importance in various areas of computer science, including theoretical computer science, algorithmics, artificial intelligence, hardware design, electronic design automation, and verification.

These problems can also be used as input for a PCR algorithm.

Graph problems

There are algorithms that are meant to be used with graphs that can be used as certifying algorithms. An example is planarity testing in graphs. A graph is called planar if it can be drawn in the plane without edge crossings (see Figure 3.7).

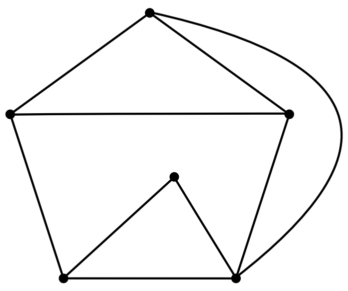

Figure 3.7.: Planar graph

A planarity tester takes a graph G and returns true if G is planar and returns false if G is non-planar. For any graph that is declared planar, a proof must be given in the form of a combinatorial embedding. When a graph is not declared planar, a proof could be a Kuratowski sub-graph. It was shown in (Kur30) that every non-planar graph contains a subdivision of either K_5, the complete graph on five nodes, or $K_{3,3}$, the complete bipartite graph with three nodes on either side (see Figure 3.8).

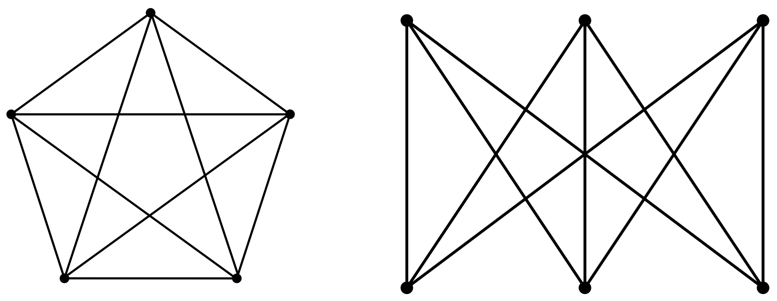

Figure 3.8.: K_5 and $K_{3,3}$ graphs

In this case, the witness of planarity is a planar drawing. This option has some disadvantages:

- there is no linear time algorithm to produce planar drawings of planar graphs
- it is non-trivial to check whether a drawing is actually planar

A less obvious form of witness is a combinatorial planar embedding of a graph. This is a cyclic ordering of the edges incident to any vertex on a graph G. In addition to this, a combinatorial embedding of a graph G is called planar if there is a planar drawing of G in which the clockwise ordering of the edges incident to any vertex agrees with the cyclic ordering specified in the combinatorial embedding. Euler observed that there is a simple algorithm for checking whether a combinatorial embedding is planar; and there are linear time algorithms for computing planar embeddings (i.e (BM99)).

In the context of Proof Carrying Results, these examples show some of the algorithms that could be used. Along with the process of constructing algorithm checkers, there is a recurrent problem: was the checker constructed to correctly check this particular algorithm? This is covered in the next section.

3.5. Correct checkers

Now we proceed to briefly discuss how the process of checking values, obtained by means of applying some certifying algorithm, affects the overall trust on the system. If we happen to have an incorrect checker for correct provided witnesses, the situation remains the same: we cannot trust the result obtained, as if we had no witnesses at all added to our result. This needs to be addressed if we want to have a robust PCR checking process. The checker must be in the trusted computing base.

Therefore, the process of checking witnesses is dependent on *how* these witnesses are checked. There are different approaches to this subject: one is to construct a checker for each case; another one is to rely on external checkers.

The problem of using a self made checker is recursive: who can tell us if this checker is correctly checking your algorithm, or even if it was well constructed at all? You can construct a simple checker, and verify that it is well designed if your language has a correct formal specification. This is not the case for many languages, however.

When using an external checker (e.g. any proof checker), it should be a proved one for it to be reliable. In addition to this, we must trust the connection made with this checker. One problem we have when using an external checker is that propositional formulas must be expressed in a foreign language to your application.

Are there checkers for every algorithm? It is difficult to conceive a lot of algorithms having a certified counterpart. We should take into account that a certified algorithm uses intermediate steps in the computation as a trace to certify correctness. To work towards the goal of having checkers for every algorithm, we must have some structure to save these intermediate steps in the process of obtaining the final result. This structure will be used as the certificate to be checked. It should remain as compact as possible, because otherwise it would use too many resources when being transmitted over the network or when kept in memory.

The next section is dedicated to the introduction of an infrastructure which enables the use of PCR in any application or system.

3.6. Towards a PCR infrastructure

To use PCR effectively, it should be integrated within a system or application. This could be implemented as a library to be included in applications to generate certified results; it is simple to add a library into programs. In this context, an infrastructure must be configured to incorporate the usage of this technique at some point. This way we will have *certified programs*.

Within the infrastructure two entities can be distinguished: an *oracle* and a *checker*, which play complementary roles in the checking process. The former takes an input and generates a certificate for that input. The latter receives the same input, the certificate produced by the oracle, and verifies if the certificate is valid for that input.

There are at least two different approaches to add the necessary infrastructure to a system:

1. independent oracles and checkers. Whenever the checker reaches a point where a certificate is needed, it calls the oracle with the corresponding input. The oracle then returns the certificate, and the checker proceeds.

2. coordinated oracles and checkers. They both receive the same input, and compute using this input until the end of the program. They should either not modify the state of this input, or modify it in the same way, so they can keep working with the same values.

Each of these approaches has advantages and disadvantages. In 1) oracles may be distributed along a number of hosts, which is a positive attribute. As a drawback, this can lead to higher latency times in the calculation of the certificates. This happens because values cannot be reused or precomputed between two distinct oracles. In 2) one advantage is that values can be precomputed to be used by subsequent oracles. But as a drawback, the oracle may need more code to keep synchronisation with the checker.

If we want to use PCR for our work, it is important that we define the way in which we would use a PCR infrastructure.

Furthermore, there are certain points that need to be addressed to create a PCR infrastructure. Even though some were mentioned before, here is a summary:

1. *give emphasis to additional hints even if they imply more computation by the untrusted party.*

2. *provide a formal notion of certificates applicable in a wide range of domains.*

3. *describe communication channel between user and untrusted party such as remote procedure call.*

4. *formally verify result checkers for stronger correctness guarantees.*

We will deal with some of these problems in the next chapter, where we will include this technique in the Acute language. Using a PCR infrastructure will give us a new level of safety against active adversaries.

4. A PCR Infrastructure for ADT values in Acute

The present chapter introduces a general infrastructure for doing PCR. One particular instance of this infrastructure will be used to guarantee robustness of abstract types in the Acute language.

We begin showing that PCR can be used to broaden the cases where communicating values of abstract types could otherwise fail. We continue introducing a protocol for PCR distributed computations, and a proposal of an infrastructure model. Moreover, we present an implementation for the Acute language of this proposal. Finally, there is a description of the implementation details.

4.1. Abstraction-safety and PCR

This section introduces the problem of abstraction-safety in the context of the Acute language. We will mention why this happens using the primitives for distributed computation *marshal*/*unmarshal*, showing the problem with a simple example. In addition to this, we define a protocol for distributed computations that use the PCR technique.

4.1.1. Motivation

Like many languages, Acute is vulnerable to active adversaries when transmitting abstract values using the *marshal* primitive. We have many contexts where we can declare that the language is abstraction-safe: the simplest context could be to work in the same runtime. But when active adversaries can tamper with data, this safety can no longer be assured. This problem is put forward in the following example.

We will use names $Alice$, Bob and $Trudy$, and $Trent$ as our example hosts. $Alice$ and Bob will be the ones that are exchanging information, $Trudy$ will represent an active adversary, and $Trent$ will be a trusted part (if needed). For transmitting values between hosts Acute provides a marshal primitive,

$$\textbf{marshal}\ e_1\ e_2{:}\mathrm{T}$$

Its parameters were explained in 2.3.2.1. The expression e_2 is the one which is actually sent, we will refer to it simply as e. This interaction is shown in Figure 4.1.

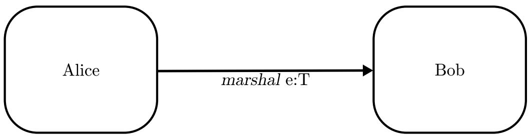

Figure 4.1.: Marshal example

Trudy (the active adversary), could modify the information exchanged between *Alice* and *Bob* in different ways. Figure 4.2 shows an example, where *Trudy* alters values sent from *Alice* to *Bob*.

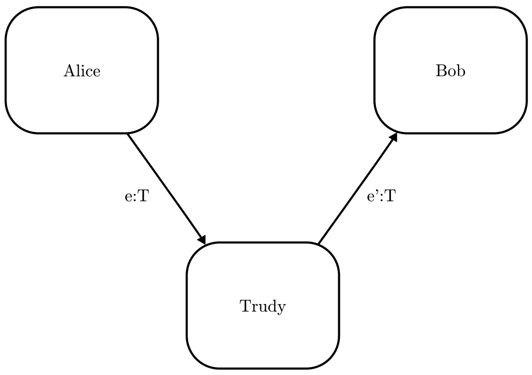

Figure 4.2.: An active adversary

In this work, we have a special interest in values of abstract types. These values have invariants that are protected by the operations of the particular module (or type) they belong to (CW85; Hoa69). These invariants can be broken if the value is modified. For that reason, if *Trudy* modifies values from the expression $e : T$, invariants of type T can be explicitly broken. This would make the language abstraction-unsafe (as seen in Chapter 2) .

To make the system more robust regarding this issue, we propose the use of PCR (Chapter 3) for our solution. To use this technique, we must define an infrastructure for its usage: this PCR infrastructure is defined in Section 4.2.

Before proceeding with the introduction of the infrastructure, we present a protocol which describes PCR distributed computations.

4.1.2. A Protocol for PCR distributed computations

The protocol that we present here is the way that hosts interact with each other, and how they obtain and verify witnesses. Consumers in general will be represented by host *Alice* and producers by host *Bob*. There could be a third host, *Trent*, that would act as a trusted host for verifying witnesses. The function used by the consumers will be denoted ϕ, and used with a parameter a, e.g. $\phi(a)$. This function ϕ should have a certification algorithm for it, that returns a result b, and a witness that certifies the computation. Witnesses, which will be denoted $\overrightarrow{w}$, are a vector of values, with some type.

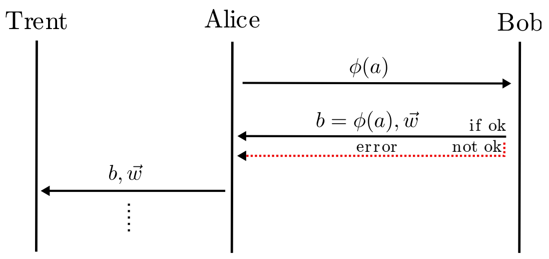

Figure 4.3.: Protocol

The protocol begins when *Alice* asks *Bob* some computation $\phi(a)$. *Bob* will return b to *Alice*, along with a vector $\overrightarrow{w}$, which certifies that $b = \phi(a)$. *Alice* can then verify, using $\vec{w}$, that $b = \phi(a)$. This process can be done only if *Alice* knows how to do it. *Trent*, being the trusted part, comes into scene if *Alice* needs further help to do this.

4.2. A PCR Infrastructure

In this section we present a generic infrastructure for doing Proof Carrying Results. To this aim, we begin introducing primitives that describe flows in distributed computations. We show how the infrastructure is implemented in the Acute language, for certified result communication. Furthermore, it will be used for verifying the witnesses of the computations. We end this section with the implementation in the Acute language of the protocol previously described (in subsection 4.1.2).

4.2.1. Primitives for distributed computation

The primitives introduced here are operations, or a sequence of operations, that describe different ways of obtaining a certified result. Each primitive will then be shown as an interaction flow between entities.

In this case, these entities will be 3 hosts interacting with each other, which are called *Alice*, *Bob* and *Trent*.

The first flow is from *Bob* to *Alice*. *Bob* sends *Alice* a value v of type T. In this case, *Alice* did not directly ask for this value. There is no guarantee that *Alice* may ever receive or process the value sent by *Bob*. Thereafter, we could say that this value has been pushed from *Bob* to *Alice*.

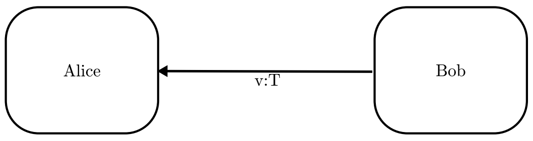

Figure 4.4.: First flow

The second flow is from *Alice* to *Bob*. Now *Alice* asks *Bob* for some value of type T. Then *Bob* returns that value of type T, or an error.

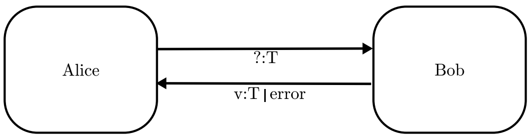

Figure 4.5.: Second flow

The last flow happens between *Alice* and *Trent*. *Alice* has a value v and asks *Trent* for a proof that v is of type T. If *Trent* can construct such proof $\vec{x}$, it returns $\overrightarrow{x}$ to *Alice*. When some exception happens, for example, if no proof can be found at all, an error is returned.

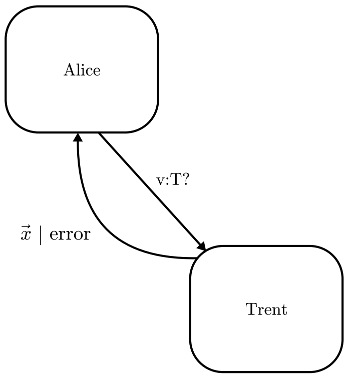

Figure 4.6.: Asking for a proof

These primitives can be combined to increase the types and complexity of flows in the system. We will be interested in the particular flows (or sub-flows) that include proofs of the value sent along the wire. In the next subsection we show how these primitives, combined, are used as a PCR infrastructure model.

4.2.2. Infrastructure model

Using the primitives and flows previously described, we will model an infrastructure for performing result certification. The infrastructure is modelled by the interaction of the 3 hosts shown in Figure 4.7.

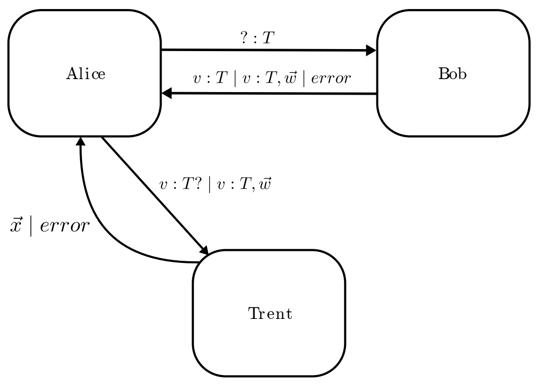
Alice
Bob
Trent
$? : T$
$v : T \mid v : T, \vec{w} \mid error$
$v : T? \mid v : T, \vec{w}$
$\vec{x} \mid error$

Figure 4.7.: Infrastructure model

A simplification of this infrastructure is that $Alice \equiv Trent$. The full infrastructure considers all the flows and primitives presented. In our implementation, we will focus on the primitive $v : T, \vec{w}$.

4.3. Implementing the infrastructure in Acute

Now that we have introduced a general infrastructure for using in distributed computations, we show how it is applied to the Acute language.

4.3.1. Certified result communication in Acute

We extend the way that we use the marshal primitive by adding a witness, obtaining a new syntax for it

marshal e1 e2:T witness

which may now include the information that can be used by the receiver to check an invariant. The use of witnesses for certification is optional: if there are no witnesses the primitive will behave as the old one. In the case that certificates are required by the receiver, the sender cannot choose: the value must be sent with a witness or it will not be used by the receiver, and an *exception* will be raised.

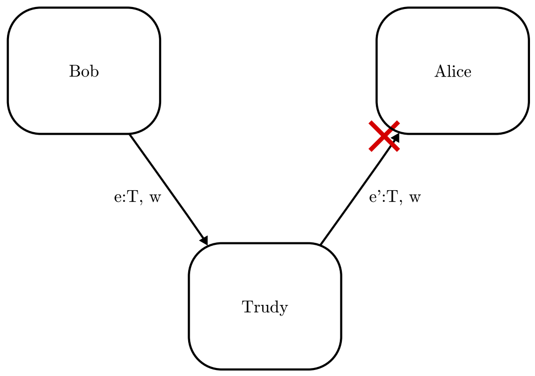

Figure 4.8.: Adversary can not modify values only

The semantic of the primitive is changed accordingly from:

$$mv ::= marshalled(E_n, E_s, s, definitions, e, T)$$

to the new semantic

$$mv ::= marshalled(E_n, E_s, s, definitions, e, c, T)$$

Here e is the core value being shipped, T its type, c its witness, s a store, E_s a store typing, $definitions$ is a sequence of module definitions, and E_n is a name environment. The E_n and E_s would not be shipped in an production implementation, but are needed to state type preservation and for runtime typechecking of reachable states. They are shipped in the Acute implementation only if literal hashes are not being used.

With this extension, our system is more robust against active adversaries (see Figure 4.8). It is important to note that the adversary could modify the value *and* recalculate the witnesses for this new value. This case does not directly affect the problem we are trying to solve. This happens because, even when the original value has been modified, it complies with the invariants the new value must have. The witnesses will be used to check this compliance, and all we should care about is that the witnesses are well generated to certify the value. A trivial example is that we ask a remote host for a prime number, and the remote host sends us some number. In the middle of this transaction, an adversary changes this number. As long as the new number is also prime and it is sent with a proof, things will keep working and abstraction safety is kept.

Verifying witnesses

To complete the process of result certification, the receiver must perform some actions when receiving the value. We suggest the use of the received witness to check that the invariant holds for the value received. Adding up, the primitive for unmarshal

unmarshal e:T'

will be changed to reflect the new extension. The new primitive

unmarshal e:T' witness

will support the use of witnesses. In the unmarshal process we will check that the invariant holds for the module by using that witness. Therefore, our main contribution is to add functionality to Acute that allow us to send values of an abstract type along with a certificate that proves that the value complies with the type invariant. Enforcing the check of witnesses must be supported, because for some types we want to receive certified values for continuing our computation. The unmarshal, as in the marshal case, alters its semantics in a similar fashion to include the witness.

Before going on with the implementation details, we show the model used for our solution.

4.3.2. Acute instantiation of the protocol

This section will give semantics to each part of the infrastructure model flows described previously. We can distinguish five main primitives:

$? : T$	$v : T$	$v : T?$	$v : T, \vec{w}$	$\vec{x} \vert error$

There is an interaction happening between a sender and a receiver, generated by each primitive. We are going to explain, intuitively, what each primitive represents.

The first one, $? : T$, represents *Alice* asking *Bob* for a value of type T. This could generate a response from *Bob* with value $v : T$, which is in fact the next primitive. The meaning of the second primitive is that *Bob* is *pushing* some value $v : T$ to *Alice*, even if she did not ask for it.

The notion of checking can be associated to primitive $v : T?$. This represents *Alice* asking a trusted party ($Trent$) to certify whether the value v is of type T. The primitive on which we made focus is $v : T, \vec{w}$, and represents a value coming with a witness of the computation of that value. The primitive $\vec{x}|error$ represents the response to a previous check for a value, and $Trent$ can return to *Alice* a vector $\vec{x}$ or an error if he could not verify this value or any other error occurred.

These primitives can be combined to obtain any of the flows described before. As an example, we show how to implement the model seen in Figure 4.6 in Acute on listing 4.1. This example is for two hosts: *Alice*, whose code is on the left, and $Trent$ on the right.

```
IO.send(marshal "StdLib" v:T)              ...
...                                        let v = unmarshal (IO.receive()):T) in
...                                        let w = cons_witness v "algorithm" in
...                                        IO.send(marshal "StdLib" v:T w)
...                                        ...
let r = unmarshal (IO.receive()):T) w      ...
```

Listing 4.1: Model example

Note that there is an implicit sequence defined. Only after *Trent* receives the value will he begin processing. The type used for representing the response vector $\vec{x}$ or an error *error* is:

```
type witness = certificate | error

and certificate = (assertion * proof) list
```

With these definitions, we can easily code any of the flows defined. In this work we will only consider the $v : T, \vec{w}$ flow, and implement it as new primitive in Acute.

The next section is dedicated to explaining the inclusion of this extension in the Acute language.

4.4. Extension of the Acute language

Based on the protocol and infrastructure model, this section shows how the primitive introduced in 4.1 was added to the Acute language. The two main changes that had to be made were, on one hand, the use of a certificate in the marshal primitive and, on the other hand, the verification of that certificate on the receiver side, in the unmarshal primitive. Both primitives needed alterations to support this new syntax and semantics. Next is a summary of the changes made to the language, then we show *how* these changes were made, and finally how the language connects with the Oracle.

4.4.1. Overview

To identify the precise locations for the changes, it was important to know the language source code files modules and structure. It is usual for languages to have an abstract syntax tree (AST) for representing the language syntax. For recognising that particular syntax, we have the parser. Closely related is the lexer, which defines tokens from source code strings. Therefore, the first files to be reviewed were the ones that contained the lexer, parser and the AST codification. Also, they were the ones to include the first modifications.

In the case of the abstract syntax tree, the main addition to the internal structures of the language was the support for certificates. There are some places where the values assigned to certificates had to be instantiated onto the language internals. The first place was in the user code, where a certificate is defined. The fragment of code that deals with user code can be viewed in Listing 4.2.

When a user introduces a certificate to be marshalled, e.g.

let foo = IO.send (**marshal** 17:Prime "bar...") **in** ...

the structures defined there are filled with the values (in the example "bar...") to be used before.

```
...
(* PCR additions *)

and assertion_item =
    | Assertion of string
    | AssertionExpr of prim_expr

and proof_item =
      | Proof of string
      | ProofExpr of prim_expr
      | NotProved

(* A certificate is a list of assertions and proofs *)
and certificate = (assertion_item * proof_item) list
...
```

Listing 4.2: Fragment of *ast.ml* file: definition of certificate

The second place where certificates were introduced is in the definition of `marshalled_body`. This structure is shown in Listing 4.3, and contains all the information that will be used in the marshalling process: its content will be marshalled, and later on unmarshalled and used by the receiver.

Support in the lexer was simple and straightforward: we only needed to recognise some tokens, mainly *certificate*.

In the parser, the additions were focused on filling the structures of the AST. It is important to note that there were two different parsers which needed modifications: one was the Acute language parser, and the other was the marshalled expression parser, whose modification was important for our purposes. The marshalled expression parser fills the information into the `marshalled_body` structure (in Listing 4.3), when receiving a marshalled value.

The medular change in the semantics of the two mentioned primitives was made in the evaluator. The evaluator is the process which takes a compiled code, and executes the program step-by-step. In particular, the operations related to marshal/unmarshal had to be modified to cope with the new semantic of the operations. In the marshal operation we added the certificate to the marshalled body to be sent over the wire. In the case of the unmarshal operation, there was more work to do by the receiver. This is why more code was needed on the evaluator, to "glue" the Acute language with our Oracle for checking certificates.

With the modifications introduced, this operation uses the `marshalled_body` structure, which contains the value and the certificate to be checked, and is filled accordingly by the parser when received. The evaluator then sends these values to the Oracle to do the verification. If it receives an exception as a response from the Oracle, this exception will be propagated. If the verification is indeed successful, the program continues executing

```
...
(* Marshalled data *)
type marshalled_body = { mb_ne : nameenv option;
                         mb_defs : definitions;
                         mb_store : store;
                         mb_storeenv : loctyplist;
                         mb_expr : expr;
                         mb_typ : typ;
                         mb_cert : certificate; }

(* the following will be the result of unserialising a
   message string;by contrast, we use Marshalled foo
   as a subexpression of the expression language *)

and marshalled_value = Marshalled_value of marshalled_body
...
```

Listing 4.3: Fragment of *ast.ml* file: additions for marshalling

as before. In next subsection we show these modifications in depth, showing portions of code of each of the mentioned parts.

4.4.2. In depht modifications

In the previous subsection we showed an overview of the changes made to the abstract syntax tree and its related files: it is a good starting point, because the structures defined there are used extensively along the language implementation. Here we extend that overview, also showing the changes in other parts of the language in depth.

The lexer was extended by adding the word *certificate* to the recognised words of the language. Listing 4.4 contains the modified part of the lexer.

```
...
/* For receiving the token from the lexer */
%token CERTIFICATE
...
let lc_keyword_table =
  create_hashtable 149 [
    "amodule", AMODULE;
    "and", AND;
    "as", AS;
    "assert", ASSERT;
    "begin", BEGIN;
    "by", BY;
    "certificate", CERTIFICATE; (* added certificate keyword *)
    "cimport", CIMPORT;
    "cmodule", CMODULE;
```

```
...
```

Listing 4.4: Fragment of *lexer.mllp* file

This had to be introduced for recognising this token, to be sent to the parser.

The parser has more complex stuff to do. It will recognise the new marshal expression, when we have the certificate added, and instantiate the parts of the abstract syntax tree with the needed information.

```
...
/* This is the certificate in source code  */
/* Certificate marshalled is different, and is
 * under these definitions */

/* A certificate is a list of assertions and proofs */

certificate:
    /* empty */  { fun ps ->
       Printf.printf "No certificate found\n"; [] }
  | certificate_list_non_empty { fun ps ->
       Printf.printf "Certificate found, filling list\n";
       ($1 ps) }
;
certificate_list_non_empty:
      certificate_item { fun ps ->
        Printf.printf "New item on certificate list\n";
        [$1 ps] }
  | certificate_list_non_empty COMMA certificate_item
        { fun ps -> $3 ps :: $1 ps }
;
certificate_item:
  | LPAREN assertion_subexpr COMMA proof_subexpr RPAREN
      { fun ps -> new_certificate_item ($2 ps) ($4 ps) }
;

assertion_subexpr:
  | STRING { fun ps ->
      assertion_item_of_string($1) }
  | simple_expr { fun ps ->
      assertion_item_of_expr(exprtoprim($1 ps)) }
;

proof_subexpr:
  | STRING { fun ps ->
      proof_item_of_string($1) }
  | simple_expr { fun ps ->
      proof_item_of_expr(exprtoprim($1 ps)) }
;
...
```

Listing 4.5: Fragment of *parser.mlyp* file: certificate definition in user source code

The main parser file is called *parser.mlyp*, and we show three fundamental parts here. The first is in Listing 4.5, and shows the definitions for parsing the certificate in user source code. The second part is the expression for recognising certificates, also in user source code. It is added to the new marshal primitive, as shown in Listing 4.6.

```
...
simple_expr:
...
| MARSHAL simple_expr simple_expr certificate
   { let l = symbol_rloc yy_envp in fun ps ->
        mkexp (LocMarshal($2 ps, $3 ps, $4 ps,
          newtypvar ps)) l }
#ifdef compiled
| MARSHALZ STRING simple_expr certificate
   { let l = symbol_rloc yy_envp in fun ps ->
        mkexp (LocMarshalz($2, $3 ps, $4 ps,
          newtypvar ps)) l }
#endif
| UNMARSHAL simple_expr certificate COLON certificate_type
   { let l = symbol_rloc yy_envp in fun ps ->
        mkexp (LocUnmarshal($2 ps, $3 ps, $5 ps,
          newtypvar ps)) l }
;
...
```

Listing 4.6: Fragment of *parser.mlyp* file: certificate addition to (un)marshal in user source code

The third part was added for parsing the extended marshal expression when unmarshalling. This expression can contain a certificate to be used at unmarshal time. This is shown in Listing 4.7.

```
...
marshalled_certificate:
  /* empty certificate, default case */
  UNDERSCORE     { fun ps -> [] }
| CERTIFICATE LBRACKET certificate_list_non_empty RBRACKET
   { fun ps -> Printf.printf "Marshalled certificate found\n";
      ($3 ps) }
;

/* function for processing certificates */
marshalled_value_pri:
 MARSHALLED LPAREN marshalled_body RPAREN
 { fun ps ->
    let newps = new_parser_state ps.pm
     (Some (ps.econst_ident_of_string,ps.econst_string_of_ident))
    in
```

```
        only_in marshalled_mode newps "marshalled not allowed"
         (Marshalled_value ($3 newps)) }
;
...
```

Listing 4.7: Fragment of *parser.mlyp* file: at unmarshal, we must recognise certificates

For all this new semantics to be taken into account, the language evaluator was modified. This evaluator is coded mainly in the *eval.ml* file. It contains the actions to be executed for all the language primitives. In particular, the marshal and unmarshal primitives and code associated can be viewed there.

We begin describing the changes in the marshal primitive. In this case we needed to include the corresponding certificate, presented by the user, to be marshalled to a string. This was achieved by adding the certificate (`Ast.certificate`) to the parameters of the *marshal* function, and to add it to the *marshalled_body* structure (see line 14 in Listing 4.8).

```
 1  ...
 2  let marshal
 3    : Ast.smallstep_outer -> string -> Ast.prim_expr ->
 4        Ast.certificate -> Ast.typ -> Ast.prim_expr
 5    = fun config mk expr cert ty ->
 6  ...
 7        let cert' = cert in
 8        let marshalled_body = { mb_ne = en';
 9                                mb_defs = defs'';
10                                mb_store = store'';
11                                mb_storeenv = storeenv';
12                                mb_expr = (primtoexpr expr);
13                                mb_typ = ty;
14                                mb_cert = cert;
15                              }
16        in
17        let s = print_marshalled_value (ps_empty_marshal ())
18                  (Marshalled_value (marshalled_body))
19        in
20        Debug.print' Opts.DBC_marshal
21          (fun () -> ("marshalling␣to:" ^ s));
22
23  ...
```

Listing 4.8: Fragment of *eval.ml* file: filling the structure to be sent

There are many functions used all along the code for printing strings at the standard output, and pretty printing the language source code files. They are used, for example, to generate the compiled code, which is like an abstraction of the user source code. Of course, we needed to modify many of them, and also introduce new printing primitives to support our extensions. The additions made to these functions were a consequence of the changes made to other functions. For completeness, they are mentioned here.

The connection to our PCR infrastructure is made at unmarshal time. The unmarshal function can be viewed in Listing 4.9, which contains this connection.

```
1  ...
2  let unmarshal: ...
3  ...
4    (* here we should call our oracle, to check if certificate
5       matches with the expected one *)
6    let matches =
7      ( match mb.mb_cert with
8      | [] -> true
9      | _ ->  (* First, check the invariant of each type *)
10         if (inv_typecheck_with_marshalled_type mb cty) then
11          Oracle.check mb.mb_expr mb.mb_cert
12         else
13          raise (Certificate_check_failure "Certificate type
14           does not match invariant type of marshalled value")
15     )
16   in
17    if (matches = false) then
18     raise (Certificate_check_failure "Certificate sent does
19      not certify marshalled value")
20    else
21     Debug.print ' Opts.DBC_marshal
22      (fun () -> ("Oracle returned: " ^ string_of_bool matches));
23
24 ...
```

Listing 4.9: Fragment of *eval.ml* file: calling our oracle at unmarshal time

In Acute, this connection is represented by a “simple” call to the check function of the Oracle module (in line 11). This module is where all the corresponding verifications are made. The process of unmarshalling could now return a new exception when invoked, as a side effect of checking the certificate. If the certificate sent does not match the marshalled value, an exception is raised (see Line 19).

The module that describes the connection with the Oracle is an important contribution made by our work, presented in the next subsection.

4.4.3. Certificate checking

For the process of checking the certificate we decided to use the COQ proof assistant. Even thought COQ has many impressive functionalities, in this case it is used only as a proof checker for the received certificate. To interact with COQ, the user has a command-line interface. This is because it is commonly used interactively. We have that a simple proof contains definitions, declarations, and the actual proof. This proof is surrounded by the “Proof” and “Qed” words. At first sight, we did not have the prerequisites of interaction, we just needed to have certain proof approved or not. This is why we first began searching for an interface which could be easily programmable for accessing COQ's own functions. Then, providing it with the corresponding information on these definitions, thesis and hypothesis, we intended to use the proof checker as a library. Unfortunately, we could not find a suitable interface for this purpose.

Following the advice of Benjamin Gregoire (Everest Team at Sophie-Antipolis), we focused on another solution. The value and its certificate, along with other pre-defined constants and COQ commands, were written into a file. After that, we call the COQ compiler passing this file to be compiled. This was easy to implement, and we could obtain basic results with it: the compiler returned a value 0 if the proof was successful or $\neq 0$ if it wasn't. This solution requires forking a new process, the COQ compiler. Figure 4.9 shows the architecture of the connection with COQ.

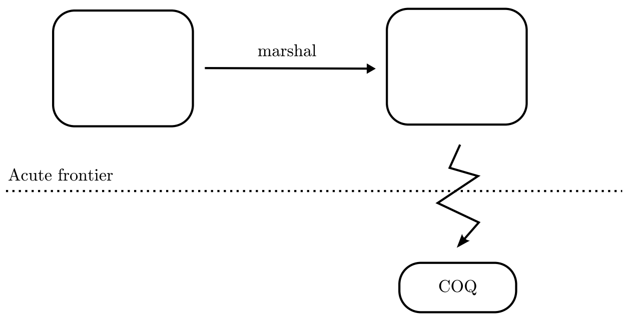

Figure 4.9.: Architecture of the connection with COQ

When writing the file to be sent to COQ we had to take several things into account. First of all, a valid proof had to be constructed with the assertions and proofs that were present in our certificate, including the received value. In the end, this procedure resulted in a concatenation of strings in the file, ordered by some criteria. Following the Figure 4.10 we provide a description of the process.

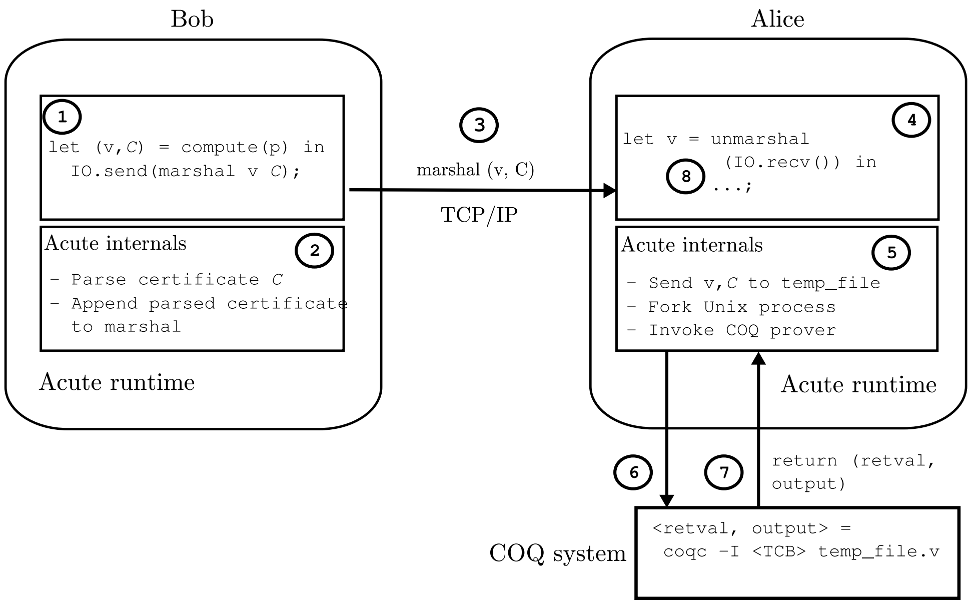

Figure 4.10.: Full proof sequence

In this Figure there are three different parts represented:

- the Acute runtime, responsible of executing Acute source code,
- the Acute (FreshML) internal execution, and
- COQ.

We describe the process using eight steps. Bob generates some value v with a witness C of that computation, and sends this value to Alice using the marshal primitive. In step (1), there is sample code that *Bob* uses to create the corresponding value v, and its certificate. Next (2), the Acute internal runtime adds the certificate to the value to be marshalled. After that (3), the marshal is transmitted over some network medium (e.g. TCP/IP). Alice, at (4), expects this value after invoking the unmarshal primitive. After the low-level network transport has delivered the message, the Acute internal runtime uses the values received to construct a valid proof sequence to be sent to COQ. This is represented in step (5). Here, a new Unix process forks and the COQ compiler is executed with the specified parameters, the value and its certificate (step (6)). After this fork, the system waits until a response is generated by COQ. When we have a successful compilation of the proof sequence, COQ ends returning a value of 0 (no error), as usual in Unix systems. If any kind of error occurs, the COQ process will return a value different from 0, and some string output is shown at the standard output of the process (step (7)). We catch this string output, and return it to the Acute runtime, to be used by the language Acute if needed, for example, to raise an exception. This is shown in step (8).

We must take into account that many proof assistants have reserved words for achieving different behaviours. COQ has its own reserved words that could affect the checking process: a consequence of a deliberate misuse could be incorrect input validated as if it was correct. An easy example is the use of COQ reserved words like *Axiom* or *Parameter*. Consequently, these words must be banned from proofs sent by clients.

Therefore, there is an extra check which leaves out these reserved words, to increase security. Finally, the COQ prelude must be empty for all proofs: this forces the presence of proofs for every step by clients.

5. Case Study: Using PCR for certified prime computation

This section introduces an example which shows the consolidation of the new functionality in the Acute language. We begin by setting a scenario for this example, and continue with a simple test case. In addition to this, we explore an extension of the protocol defined before for doing distributed calculation of witnesses and also certificate checking.

5.1. Scenario

Alice wants to start a secure communication with some host, which acts as a server. At some point in this process, she will need one large prime number to use as a seed for some cryptographic algorithm. *Alice* does not have computational resources to do this computation by herself. Thus, she will outsource this computation to *Bob*.

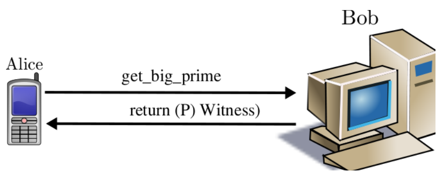

Figure 5.1.: Alice and Bob

Bob is an untrusted host for *Alice*, but with lots of resources to compute this number. Knowing this, *Alice* requests *Bob* a prime number. *Bob* generates this number, along with a proof that it is prime. When this number is returned to *Alice*, she will use the proof to check that the number is prime. If this checking succeeds, she can start the secure communication with the server.

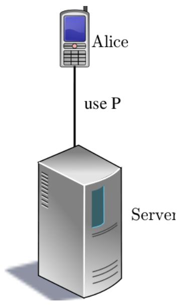

Figure 5.2.: Alice and Server

All hosts will communicate using the **marshal** / **unmarshal** primitives for the transmission of values with each other. This **marshal** primitive can be used with a witness. At unmarshal time that witness will be checked. This check happens automatically, and is made in a transparent way. In this process, the Acute language will try to verify all properties and values that come in the witness.

This can be done in two different ways:

- using only the information received. In this case the proof must be complete, with all needed sub-proofs. The COQ example for this case is when we have no *prelude* at all.
- relying on some pre-shared information. For example, there could be a *trusted common base* of algorithms and proofs. This base will be referred to in some way by the sender. An example TCB table could be 5.1. When a host needs to prove some property, it will look up its table to find out how to check that property. In the case that the property to check is not on the table, the request cannot be fulfilled, and an exception will be returned.

Property name	COQ Theorem	COQ Code
Prime	Pocklington	...
Even	EvenProp	...
Odd	OddProp	...
GCD	GCDProp	...
OrderedLists	SortedProp	...

Table 5.1.: TCB Properties and proofs

In any case, the needed properties must have a proof to be used by the COQ proof assistant. In the next subsection we instantiate this simple scenario into a simple test case, in which we make the proof of concept of our proposal.

5.2. Pocklington certificates

There are different ways to *certify* that one particular number is prime (Ber04). Pocklington's theorem(Poc16), shown in 3.4 provides a sufficient condition for primality of natural numbers. We show a resume of it here to help the reader.

Theorem 5.1. *Given a natural numbers $n > 1$ and a witness a and some pairs $(p_1, \alpha_1), \ldots, (p_k, \alpha_k)$, it is sufficient for n to be prime that the following conditions hold:*

$$p_1 ... p_k \, are \, prime \, numbers \tag{5.1}$$

$$(p_1^{\alpha_1} ... p_k^{\alpha_k}) \mid (n-1) \tag{5.2}$$

$$a^{n-1} = 1(\mathsf{mod}\, n) \tag{5.3}$$

$$\forall i \in \{1, ..., k\} \, \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) = 1 \tag{5.4}$$

$$p_1^{\alpha_1} ... p_k^{\alpha_k} > \sqrt{n} \tag{5.5}$$

We highlight these two points from the previous definition:

- Given n, it is required much more computational power to determine numbers for $a, p_1, \alpha_1, \ldots, p_k, \alpha_k$ than to check that these numbers verify the conditions 5.2-5.5. Therefore, we can say that $a, p_1, \alpha_1, \ldots, p_k, \alpha_k$ form a $Pocklington\, Certificate$.
- For a natural number n, provided we are given $p_1, \alpha_1 \ldots, p_k, \alpha_k$ and a, checking primality of n is made by numerical computations for conditions 5.2-5.5, and condition 5.1 can be done recursively.

In (GTW06), there is a proposal for using pocklington certificates for certifying prime numbers. We used some of the results obtained in that work (e.g. the *pocklington* executable) in the course of constructing this test case.

5.3. Simple test case

After setting the scenario, we will present a test case in detail. We will be using a number of Acute modules, the COQ proof assistant and a marshalled value to be tested. The following Acute modules will be used for this example:

- *Generator*: used by module *prime* for generating a new prime number using an known algorithm.
- *Prime*: this module will have the code to work with prime numbers.

Alice will be using the prime Module for receiving a prime number.

```
(* Prime module *)
includesource "prime.ac"

(* Alice just receive a prime number *)
let p = (unmarshal (IO.receive ()):Prime.t)
    in
    Pervasives.print_endline "Prime number is: ";
    Prime.print_prime p;
    Pervasives.print_newline ();;
```

Listing 5.1: Alice's code

In this listing, *Alice* receives a marshalled prime number, and prints it at the standard output. In the **unmarshal** operation is where the underlying work of certificate checking is done.

```
(* Prime module *)
includesource "prime.ac"

(* Generate a prime number to be sent to Alice *)
let p = Prime.generate(Prime.init) in
  IO.send(marshal "StdLib" p
    ("prime","apply (Pocklington_refl
      (Pock_certif 1234567891 2
       ((3607, 1)::(2,1)::nil) 12426)
        ((Proof_certif 3607 prime3607) ::
         (Proof_certif 2 prime2) ::
          nil)).
 exact_no_check (refl_equal true).
"));;
```

Listing 5.2: Bob's code

Bob generates a new prime, using some seed, and then sends this prime with a certificate for it.

In the process of generating certificates we will be using a tool called *pocklington* from the CoqPrime project (INR). For example, for generating the certificate for number 1234567891, the command *pocklington* 1234567891 generates the COQ code in Listing 5.3.

```
Require Import PocklingtonRefl.

Set Virtual Machine.
Open Local Scope positive_scope.

Lemma prime1234567891 : prime 1234567891.
```

```
Proof.
  apply (Pocklington_refl
   (Pock_certif 1234567891 2 ((3607, 1)::(2,1)::nil) 12426)
   ((Proof_certif 3607 prime3607) ::
     (Proof_certif 2 prime2) :: nil)).
  exact_no_check (refl_equal true).
Qed.
```

Listing 5.3: Coq Program

With this helper program, we can easily generate a certificate for any prime number generated by *Bob*. This test case simplifies the scenario, and just sends a prime number from *Bob* to *Alice*.

When we execute both programs, the prime number is marshalled and sent by *Bob*, and received and unmarshalled by *Alice*. The Acute language, at the moment of receiving the certificate, uses the checker to certify this prime number using the algorithm defined in its table. The implementation adds two important files, which must be part of the TCB. This files are to be used by the proof assistant:

- A TCB prelude is for defining which prelude to add to a proof (if any), and
- A TCB list is used for defining the list of local directories to be included (-I) for compiling the proofs.

In this case, we rely on code provided by the COQ prime project to prove the number is prime using the certificate received. The result of the execution of *Alice* code is in listing 5.4.

```
:::::::: +++Desugar source definition ::
:::::::: +++Compile source definition ::
:::::::: +++Valuabilities derived:  ::
:::::::: +++Done source definition ::
:::::::: ++Done source file ::
:::::::: ++Scope-resolve source definition ::
:::::::: ++Infer source definition ::
:::::::: ++Desugar source definition ::
:::::::: ++Compile source definition ::
:::::::: ++Valuabilities derived:  ::
:::::::: ++Done source definition ::
:::::::: +Done source file ::
:::::::: +Scope-resolve expression ::
:::::::: +Infer expression ::
:::::::: +Desugar expression ::
:::::::: +Compile expression ::
:::::::: +Strip location information ::
:::::::: +Done expression ::
:::::::: Done source file ::
:::::::: Run ::
:::::::: Executing program ::
Type to check for in certificate found
No certificate found
```

```
Marshalled certificate found
New item on certificate list
Finished writing file with proofs.
/tmp/cert_d8a3a3.v
/usr/bin/coqc -I /home/fzipi/sources/coqprime/Tactic
  -I /home/fzipi/sources/coqprime/N
  -I /home/fzipi/sources/coqprime/Z
  -I /home/fzipi/sources/coqprime/PrimalityTest
  -I /home/fzipi/sources/coqprime/List
  -I /home/fzipi/sources/coqprime/elliptic
  -I /home/fzipi/sources/coqprime/num
  -I /home/fzipi/sources/coqprime/num/W/W8
  -I /home/fzipi/sources/coqprime/examples
  -I /home/fzipi/sources/coqprime /tmp/cert_d8a3a3.v
COQ said: (no error)
Prime number is:
1234567891
Thread exited cleanly:
n0: [RUNNABLE] ;; ()
:::::::: Done ::
Success:
, Es:{}, s:{}, threads:{}, (RUNNABLE: , SLOWCALL: ), MX:{}, CV:{}
```

Listing 5.4: Result of execution

This achieves our primary goal: now the Acute language is more abstraction-safe than before. In the next sections we will extend our proposal, distributing the work of generating witnesses for a computation among multiple hosts.

5.4. Towards a distributed certifying infrastructure

Extending the proposal presented in Chapter 4, we can try to distribute the task of generating witnesses. This can be achieved by using a protocol for doing verification (or calculation) of witnesses in a distributed fashion. Using this approach will lead us to a way of checking more computation intensive properties, by sending the witness to hosts which could have better (or more specialised) hardware than us. This will be described in detail in the next sections.

It is important to note that we do not want to solve the problem of automatic generation of proof conditions. Many experienced research groups are working on this subject, which is still an open problem. Solving this problem exceeds the limits of this work. We base this extension in that proofs could be divided into smaller sub-proofs, and the process of verification can behave independently in each of these sub-proofs.

5.5. Extending the protocol

The protocol in 4.1.2 must be extended to include this case. Now we have to consider additional hosts, and each additional host will aid us to check some witness.

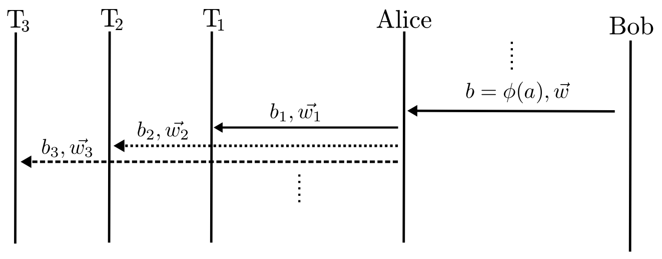

Figure 5.3.: Extension of the protocol

In the previous protocol, *Bob* returned a vector $\vec{w}$ to *Alice*. Suppose that we have an example of such vector in equation 5.6. In this vector there will be some values that *Alice* can verify; and there will be other values that may need further verification steps. This is shown in equation 5.7.

$$\overrightarrow{w} = (w_1, w_2, \ldots, w_n) \tag{5.6}$$

We call the properties that have been proved p_i, and the properties that remain to be proved ϕ_i. For each proven property p_i there will be a vector $\vec{w_i}$ that will be the witness of this proof. Thus, we have that

$$\forall i \begin{cases} w_1 & \mathit{proved} \Rightarrow \mathit{Prop}(p_1, w_1) \\ \phi_2 & \mathit{not\,proven} \\ \ldots & \ldots \\ w_j & \mathit{proved} \Rightarrow \mathit{Prop}(p_j, w_j) \\ \ldots & \ldots \\ \phi_n & \mathit{not\,proven} \end{cases} \tag{5.7}$$

The rest of the necessary verifications to prove that property *Prop* holds can be done in a distributed way. If a host can check some properties directly, we call those properties *local properties.* There will be properties that we cannot check by ourselves, which must be forwarded to some other host (*Trent*, in our example) that knows *how* to verify their values; these will be called *remote properties.* Accordingly, we obtain that

$$p_1 \wedge \phi_2 \wedge \cdots \wedge p_j \wedge \cdots \wedge \phi_n \rightarrow \mathit{Prop}(b, \phi(a)) \tag{5.8}$$

When we are checking all the properties that must be verified, some of them will be local and others remote. But we must verify all properties to prove that the main property Prop holds. A similar example to the one that we introduced before with prime numbers can be used to describe the new behaviour of the distributed generation of witnesses.

Another task that could be distributed is the generation of proof conditions. For example, we could ask a third party to give us the properties that must be proved, instead of having these properties in every host.

In the next section, we will show how the protocol is applied to the calculation of a prime number, to see how it works.

5.6. Protocol example with prime numbers

In this instance, our function ϕ will be *GeneratePrime* (with or without parameters, it does not matter right now). We will call our result n instead of b, and the vector containing the witnesses $\vec{w}$.

At some point of the execution of a program, *Alice* needs a prime number. She knows (by configuration, or by other means) that *Bob* can give her that number by calling the *GetPrime* primitive. *Alice* then sends a message *GeneratePrime* to *Bob*. *Bob* will then return n to *Alice*, along with a witness $\vec{w}$ that n is a prime number. There could be an exception here: if *Bob* cannot construct the prime number in the way that *Alice* asked, or if another error appeared in the calculation of n and its witnesses vector $\vec{w}$. This part is similar to the protocol introduced previously, and now comes the additional part that distributes computations.

Alice will ask *Trent* for witnesses of prime value n. We will use the Pocklington certificates (5.2) in this example.

If *Trent* can find a Pocklington certificate $\overrightarrow{w}$ of n, it is returned to *Alice*. This certificate will be a list of integers, with this form:

$$\overrightarrow{w} = (p_1, \alpha_1, p_2, \alpha_2, \ldots, p_k, \alpha_k, a) \tag{5.9}$$

This list will be used to verify that Pocklington's criteria holds, so we must check that:

$$p_1...p_k \, are \, prime \, numbers \tag{5.10}$$

$$(p_1^{\alpha_1}...p_k^{\alpha_k}) \mid (n-1) \tag{5.11}$$

$$a^{n-1} = 1(\mathsf{mod}\, n) \tag{5.12}$$

$$\forall i \in \{1, ..., k\} \, \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) = 1 \tag{5.13}$$

$$p_1^{\alpha_1}...p_k^{\alpha_k} > \sqrt{n} \tag{5.14}$$

As we have already seen, these values will be passed to a proof checker like COQ which has the same set of equations/properties. We will need additional properties to be proved for this criteria. In particular, we will name the property of a number n being a prime and verifiable with Pocklington's criteria as PocklingtonProp n. There are more properties that complete this criteria, namely ModuloProp, DivideProp, GCDProp, GreaterThanProp. We now summarise the properties that must be proved in order to certify our prime:

- PrimeProp $\mathsf{p_i}$ $\mathsf{w_i}$, which is true if p_i is prime
- DivideProp $\mathsf{p_1^{\alpha_1}}$ ... $\mathsf{p_k^{\alpha_k}}$ n, which is true if $(p_1^{\alpha_1}, ..., p_k^{\alpha_k}) \mid (n-1)$
- ModuloProp $\mathsf{a^{n-1}}$ 1 n, which is true if $a^{n-1} = 1(\mathsf{mod}\, n)$

- $\mathsf{GCDProp\ a\ p_i\ p\ k}$, which is true if $\mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) = 1$
- $\mathsf{GreaterThanProp\ p_1^{\alpha_1}\ ...\ p_k^{\alpha_k}\ sqrt\ n}$, which is true if $\forall i \in \{1, ..., k\}\, p_i^{\alpha_i} > \sqrt{n}$

Other common properties, that will help us are:

- $\mathsf{SQRTProp\ sqrt\ n}$, which is true if $sqrt\, n = \sqrt{n}$
- $\mathsf{ExpProp\ p}\, \alpha\, \beta$, which is true if $p^\alpha = \beta$
- etc.

After adding these properties, the next proposition must hold

$$\begin{array}{rcl}
\forall i,\ \mathsf{IsPrimeProp\ p_i\ w_i} & \wedge & \\
\mathsf{DivideProp\ p_1^{\alpha_1}\ ...\ p_n^{\alpha_n}\ n} & \wedge & \\
\mathsf{ModuloProp\ a^{n-1}\ 1\ n} & \wedge & \\
\forall i, \mathsf{GCDProp\ a\ p_i\ n\ k} & \wedge & \\
\mathsf{GreaterThanProp\ p_1^{\alpha_1}\ ...\ p_k^{\alpha_k}\ sqrt\ n} & \rightarrow & \mathsf{IsPrimeProp\ n}
\end{array}$$

Here, IsPrimeProp represents the property of a number being prime. This way we can achieve distribution of the generation of certificates for prime numbers. For each w_i to be tested for primality, we just send it to another node for constructing a certificate for it, or just return an error if the number can not be certified as prime.

In the next chapter we present the conclusions reached by this work.

6. Conclusions, related and further work

This chapter is devoted to the conclusions obtained developed in this thesis. We also present related work done in other languages, or different approaches to this subject. Finally, we suggest future work to be carried out in this area.

6.1. Conclusions

We have defined and implemented an infrastructure for result certification. This infrastructure can be used in other places where there is a need for using the proof carrying results technique and not only in this particular case. It is independent from the language, and also, with little or no effort, from the proof assistant used.

There is a successful implementation, for the Acute language, of a solution for the detected problem. This solution uses the infrastructure defined here, and its implementation is available for further testing (Zip08). In order to achieve this, we used the technique of proof carrying results, where values transmitted to other hosts carry a witness that proves the computation has been done in a correct way. For the process of verifying that the witness certifies these transmitted values, we used the COQ proof assistant. The proofs are made over these values of abstract types and their properties, carried in a certificate.

As we successfully added this technique to an arbitrary language, it should not be difficult to extend the framework to any other programming language which suffers from similar problems. This means that if we can run a COQ proof assistant, and either 1) we have access to the source code of the programming, or 2) the programming language has an API for interfacing in a similar way with the system; then it should be a reasonable amount of work to add this mechanism to the language/system.

Working with a proof assistant is a good way of delegating the checking process. It is important to use a well-designed, well proven tool for doing this step. Luckily, the knowledge on the COQ system by the members of our research group was very helpful to bypass some of the drawbacks that we had when establishing the connection with the Acute language.

Proof Carrying Results is a promising new approach. The progress on this subject is directly bound to the progress on Certifying Algorithms. This happens because it is difficult to find an algorithm that works for a general case. Despite this consideration, there are many applications of this technique in a number of places, given that we do not trust other hosts. In particular, in global grids where we could have many distributed computations, this technique can obtain good results. Finally, we have made a proof of concept that implementations can be carried to extend systems in a successful way. The source code of the Acute language with support for PCR can be found in (Zip08). There are many improvements that can be made that complement the work done up to now, and these are presented in the next section.

6.2. Future work

There are a number of lines that can be followed starting from this point. Throughout this work we have opened many possible interesting lines of research. We will try to summarise them here:

- The work on certifying algorithms is a promising way for algorithm designers. Thinking about new applications for certifying algorithms in other contexts is interesting since it might allow us to add them to the list of possible ways to reach a solution.
- Proof Carrying Results can be implemented in other distributed languages, with the same problems that we introduced in Acute. The most natural is HashCaml, because its features were adapted from the ones that are problematic (for our purposes) in Acute, like global hashes for marshalling. Microsoft's F# could be an interesting domain of application as well. If we have few resources for performing computations (e.g. Java cards, cellular phones, etc.) it is an ideal place to be used. This is because computations can be performed in a server with lots of computing resources, and just checked by the client application, given a clear API for doing result certification.
- For the COQ proof assistant, it could be interesting to have a *Proof* service: this should be a server, that listens in a TCP/IP port and forwards the terms and lemmas received to an executing COQ process. This connection must be performed over SSL/TLS and using a trusted PKI certificate. In return, it should send proof obligations to clients, or a "Qed" if nothing remains to be proved. For maximum portability, the exchange can be made in XML language. COQ already has a DTD defined for the purpose of reading and writing XML files (dt08b). The exported files are an XML encoding of the lambda-terms used by the Coq system. The implementation details of the Coq system are hidden as much as possible, so that the XML DTD is a straightforward encoding of the Calculus of (Co)Inductive Constructions.
- Distributed generation of certificates, as we mentioned before, is an open field of research by many groups. It is related to the automatic generation of verification conditions. There are many things that must be accomplished to use distribution. Some of them are related to formal steps that must be covered by any solution introduced: for example, termination must not be altered by this distribution. The idea is to bind this generation to more complex questions.
- Only do the certificate check if the type of the received value is abstract. This feature was not implemented this time, because it could break soundness equations.

To have this feature available, the equations must be analysed carefully so as not to break type soundness.

6.3. Related Work

There are three basic types of work related to ours. First, there is work that use similar techniques to connect to the proof checkers. Second, there are distributed languages with properties similar to the ones in the language we worked with, Acute, that try to solve similar kinds of problems. But almost none of the ML-like languages that have type-safe marshalling deal with dynamic type equality across programs in the presence of abstract types. Third, there is work on other techniques, which are probabilistic instead of being a verification technique like the one presented in this work.

Proof assistant connection

Yves Bertot and Laurent Thery introduced guidelines for building generic user interfaces for theorem provers (BT98). CtCOQ (Gro99), provided a first graphical working environment for the COQ proof system, based on these guidelines. The PCoq tool (Gro) continued the work done in CtCOQ, and extended it with a Java user interface. After that project, the approach was inherited and used by the COQIde interface (GTK+).

ProofGeneral (Asp) for Emacs also followed the work done in CtCOQ. It communicates with the theorem prover by using the Emacs's facilities for shell interaction, and has some common functionality with the other projects mentioned. Besides this interaction, none of these projects used the proof assistant as a library because they were oriented towards simplifying the interactive proof process.

The future plans of ProofGeneral include the PGIP, which is the *Proof General Interaction Protocol*, a message passing protocol for communicating proof components, primarily with interactive theorem provers and their interfaces (Asp07). This protocol has an architecture that includes a Broker, Proof Assistants, and some kind of display interface: a Web browser, text interface, etc. See Figure 6.1 for details.

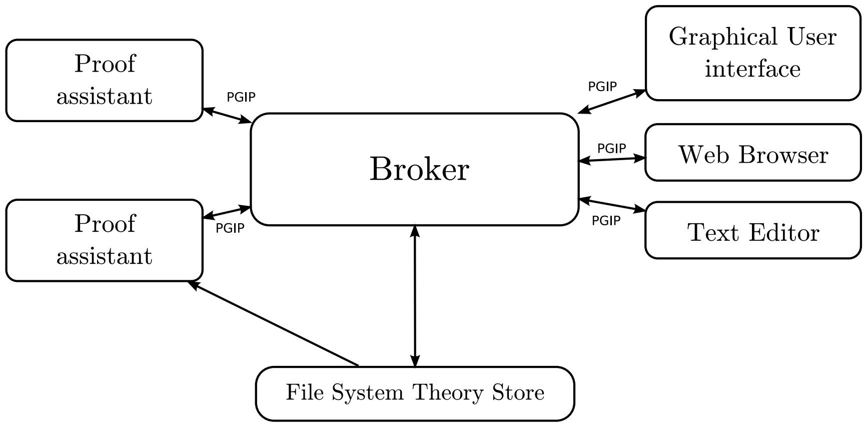

Figure 6.1.: PGIP Architecture

In addition to this, Cezary Kaliszyk implemented a Web front-end for accessing various proof assistant engines, by using the technique of writing to a file and parsing strings (KvRWH07).

Successful work on the use of the internal structure of the COQ proof assistant was done in the MOEBIUS project by the MOEBIUS team. The team worked on a Verification Conditions Generator (VCGEN) for Java (GS07). This project needed to access the COQ engine, and they needed to access its internal structures.

Programming languages

We will not be exhaustive with programming languages that support typed distributed communications.

Previous work on ML-like distributed languages began with JoCaml (Fou98): a language for Distributed Concurrent and Mobile support. The implementation was recently relaunched to make it more compatible with OCaml (Fes07). For concurrent programming, JoCaml implemented the *join calculus* primitives into Caml. The language has limitations in its actual implementation, basically in message marshalling. Code mobility has not been implemented yet, which means that we cannot send a function over a communication channel. Communications between producers and consumers are made by using a name service which registers services, and is used to search for available functions.

The Alice (Kor01) distributed programming language is closer in many aspects to the work in Acute. It evolved from a research language to a full language, it has been continuously improved, and it is very stable. This is why it has become a complex piece of software, with the introduction of many features: futures, higher order modules, packages, pickling, components, distribution and constraints. Furthermore, it includes a rich system of applications for producing software: a virtual machine, compiler, interpreter and linker, data inspector, connection with SQL databases and XML documents. Pickling is the term used for type-safe marshalling in Alice. When working with abstract types, they are all dynamically generated. In order to establish a shared abstract type across the distributed system, it can be pre-evaluated and the results distributed.

HashCaml is the last example of functional languages. It is an extension of the OCaml byte-code compiler, with support for type and abstraction-safe marshalling. In order to connect different hosts we use the marshal primitive. This extension uses the research done in Acute, and the idea is to be an unobtrusive modification to the OCaml core. The work on global runtime type names is also extended, by including more key OCaml type constructs like user-defined variant and record types, substructures, functors, separate compilation and external C functions. It is mentioned in (BSSS06) that the language protects itself only against accidental errors, not adversaries of any kind.

In the context of object oriented languages, Java (AG96) boosted the distributed programming paradigm, permitting the execution on lots of hosts by using its virtual machine as a basis. Primitives for doing distributed computations included the notion of marshalling (serialisation). This facility included version identifiers with the class definitions of serialised objects, as we can see in the Remote Method Invocation facility (WRW96). The main problem for Java programs is that classes are only identified by their (syntactic) name and class loader. This introduces problems when trying to use two classes with the same name but different methods.

The experimental language X10 (SSvP07) may be viewed as *generic* Java, without concurrency, arrays or built-in types; with the addition of *places*, *clocks*, *activities* and distributed multi-dimensional arrays. Conceptually, places are “virtual shared-memory multi-processors”. This language was conceived for doing high-end, high-performance, high-productivity computing. An X10 program acts on data objects, using lightweight threads called activities. It has a unified global access space, which means that objects can be referenced by activities in any place. Programs written in X10 are guaranteed to be statically type, memory and pointer-safe. No work can be identified to preserve these safety properties in the presence of active adversaries.

There is a project called Distributed Java (OAY05), to add type-safe higher-order code mobility to Java. The project mixes the functional and object-oriented paradigms by adding functions to the Java programming language as first-class citizens. The language integrates type-safe higher-order functions for Java, full integration with Java RMI, and allows fine-grained control of class loading in distributed applications. It is based on the Polyglot Extensible Compiler Framework (Mye). The project mentions that the integrity of resources can be checked dinamically when code arrives (e.g. an invariant), by adding constraints in a particular primitive (*RC-Defrost*). But there is no specific mention to security as a global concern, or how type-safety is assured in the presence of active adversaries.

For the Microsoft .NET framework, we have languages that rely on the virtual machine that are functional or object oriented. An example of a functional language for .NET is F# (Res07). This language is like a porting of OCaml for the .NET framework, with some additions. F# was developed as a pragmatically-oriented variant of ML that shares a core language with OCaml. Unlike many type-inferred, statically-typed languages it also supports many dynamic language techniques, such as property discovery and reflection where needed. F# includes extensions for working across languages and for object-oriented programming, and it works seamlessly with other .NET programming languages and tools.

Related techniques for securing distributed computations

In this document we used one technique for verification of distributed computations to make them more secure. There are other techniques that introduce the utilisation of probabilistic methodology for detecting possible alterations (or cheating) in the context of grid computations. The work by Wenliang Du et. al (DJMM04) on grid computing and on distributed computations by Phillipe Golle and Ilya Mironov (GM01) are examples of these other techniques. The main approaches are different, and even when they have very low probabilistic numbers of not being caught cheating with the results, they are not one hundred percent reliable. This is, of course, dependent on what kind of adversaries you have, and the threat analysis you made for your system. Following the strategy of checking the results of computations, Grid result checking (GRMR05) focuses on statistical checking. There, the authors define a process that:

1. ensures that a set of results is indeed correct (with high probability)
2. does not unduly eliminate results which are actually correct (false positives)
3. keeps the test low-cost

The first point is for users, as they need the result to be correct. The second point is for the Grid system, because it is important not to waste CPU cycles in the global grid. The main technique used is *sequential analysis* (GS91), which is widely used in statistical analysis. The third point is rather obvious, but it is mentioned for completeness.

Bibliography

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996. ISBN 0-201-63455-4.

[And82] Gregory R. Andrews. Distributed programming languages. In *ACM 82: Proceedings of the ACM '82 conference*, pages 113–117, New York, NY, USA, 1982. ACM. ISBN 0-89791-085-0. `doi:10.1145/800174.809772`.

[Asp] David Aspinall. Proof general. Available from: `http://proofgeneral.inf.ed.ac.uk/` [cited 2008.09.05].

[Asp07] David Aspinall. Proof general interaction protocol, 2007. Available from: `http://proofgeneral.inf.ed.ac.uk/wiki/PG/PGIP` [cited 2008.09.05].

[Ber04] Daniel J. Bernstein. Distinguishing prime numbers from composite numbers: the state of the art in 2004, 12 2004. Aimed at Math. Comp. Available from: `http://cr.yp.to/primetests/prime2004-20041223.pdf` [cited 2008.04.29].

[BHII87] Farokh Bastani, Wael Hilal, S. Sitharama Iyengar, and S. Sitharama Iyengar. Efficient abstract data type components for distributed and parallel systems. *Computer*, 20(10):33–44, 1987. `doi:10.1109/MC.1987.1663386`.

[BHS$^+$03] Gavin M. Bierman, Michael W. Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In Colin Runciman and Olin Shivers, editors, *ICFP*, pages 99–110. ACM, 2003. ISBN 1-58113-756-7. `doi:10.1145/944705.944715`.

[BM99] John M. Boyer and Wendy J. Myrvold. Stop minding your p's and q's: A simplified O (n) planar embedding algorithm. In *SODA*, pages 140–146, 1999. `doi:10.1145/314500.314545`.

[BP06] Gilles Barthe and Fernando Pastawsky. Notes on proof carrying results. INRIA Sophia-Antipolis Technical Report, 2006.

[BS79] Daniel M. Berry and Richard L. Schwartz. Type equivalence in strongly typed languages: one more look. *SIGPLAN Not.*, 14(9):35–41, 1979. `doi:10.1145/988113.988117`.

[BSSS06] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for ocaml. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 20–31, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. `doi:10.1145/1159876.1159881`.

[BT98] Yves Bertot and Laurent Thery. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(2):161–194, 1998. Available from: `http://citeseer.ist.psu.edu/bertot98generic.html`.

[Car96] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1996.

[CDJ$^+$89a] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The modula 3 type system. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 202–212, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. `doi:10.1145/75277.75295`.

[CDJ$^+$89b] Luca Cardelli, James Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, DEC SRC, November 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985. Available from: `http://citeseer.ist.psu.edu/cardelli85understanding.html`.

[DJMM04] Wenliang Du, Jing Jia, Manish Mangal, and Mummoorthy Murugesan. Uncheatable grid computing. In *24th International Conference on Distributed Computing Systems (24th ICDCS'2004)*, pages 4–11, Tokyo, Japan, March 2004. IEEE Computer Society.

[dt08a] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2008. Version 8.1pl3. Available from: `http://coq.inria.fr` [cited 2008.10.01].

[dt08b] The Coq development team. Exporting coq theories to xml, 2008. Version 8.1pl3. Available from: `http://coq.inria.fr/V8.1pl3/refman/Reference-Manual016.html#toc88` [cited 2008.10.01].

[Fes07] Fabrice Le Fessant. Jocaml: a language for concurrent distributed and mobile programming, 2007. Available from: `http://jocaml.inria.fr/` [cited 2008.09.05].

[Fou98] Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* Ph.D. thesis, Ecole Polytechnique, 1998. Available from: `http://research.microsoft.com/copyright/accept.asp?path=/users/fournet/papers/dissertation.pdf`.

[GM01] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In David Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 425–440. Springer, 2001. ISBN 3-540-41898-9. Available from: `http://link.springer.de/link/service/series/0558/bibs/2020/20200425.htm` [cited 2008.02.20].

[GMZ00] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst.*, 22(6):1037–1080, 2000. `doi:10.1145/371880.371887`.

[GRMR05] Cécile Germain-Renaud and Dephine Monnier-Ragaigne. Grid result checking. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-019-1. `doi:10.1145/1062261.1062280`.

[Gro] Lemme Group. Pcoq: a graphical user-interface for Coq. Available from: `http://www-sop.inria.fr/lemme/pcoq/`.

[Gro99] Lemme Group. Ctcoq: an environment for mathematical reasoning. *SIGSAM Bull.*, 33(3):21–22, 1999. `doi:10.1145/347127.347405`.

[GS91] B.K. Ghosh and P.K. Sen, editors. *Handbook of Sequential Analysis.* CRC, 1 edition, April 1991. ISBN 978-0824784089.

[GS07] Benjamin Grégoire and Jorge Luis Sacchini. Combining a verification condition generator for a bytecode language with static analyses. In Gilles Barthe and Cédric Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2007. ISBN 978-3-540-78662-7. `doi:10.1007/978-3-540-78663-4\_4`.

[GTW06] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. A computational approach to pocklington certificates in type theory. In *FLOPS*, pages 97–113, 2006. `doi:10.1007/11737414_8`.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. `doi:10.1145/363235.363259`.

[INM84] INMOS Ltd. *OCCAM Programming Manual.* Prentice-Hall International, 1984.

[INR] INRIA. Coq prime project. Certifying Prime Numbers with the Coq prover. Available from: `http://coqprime.gforge.inria.fr/` [cited 2008.06.04].

[JGF96] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL*, pages 295–308, 1996. Available from: `ftp://ftp.supelec.fr/lsi/pub/lambda/haskell/doc/concurrent-haskell.ps` [cited 2008-10-07], `doi:10.1145/237721.237794`.

[JHM73] Jr. James H. Morris. Types are not sets. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 120–124, New York, NY, USA, 1973. ACM. `doi:10.1145/512927.512938`.

[KMMS03] Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 158–167, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0-89871-538-5.

[Kor01] Leif Kornstaedt. Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Electronic Notes in Computer Science*, volume 59, Firenze, Italy, September 2001. Elsevier Science Publishers.

[Kur30] C. Kuratowski. Sur les problèmes des courbes gauches en Topologie. *Fund. Math.*, 15:271–283, 1930.

[KvRWH07] Cezary Kaliszyk, Femke van Raamsdonk, Freek Wiedijk, and Maxim Hendriks. Proofweb - web interface for proof assistants, 2007. Available from: `http://www.cs.ru.nl/~cek/proofweb/` [cited 2008.05.26].

[LPS$^+$03] James J. Leifer, Gilles Peskine, Peter Sewell, Keith Wansbrough, and Inria Rocquencourt. Global abstraction-safe marshalling with hash types, July 09 2003. Available from: `http://www.sics.se/pepito/D3.3/papers/leifer-globas.ps` [cited 2007.12.05].

[MEK$^+$05] Kurt Mehlhorn, Arno Eigenwillig, Kanela Kanegossi, Dieter Kratsch, Ross McConnel, Uli Meyer, and Jeremy Spinrad. Certifying algorithms (a paper under construction). Max-Planck-Institut für Informatik working paper, 2005. Available from: `http://www.mpi-inf.mpg.de/~mehlhorn/ftp/CertifyingAlgorithms.pdf` [cited 2007.04.24].

[Mye] Andrew Myers. Polyglot extensible compiler framework. Available from: `http://www.cs.cornell.edu/Projects/polyglot/` [cited 2008.09.05].

[NL96] George C. Necula and Peter Lee. Proof-carrying code, October 28 1996. Available from: `http://citeseer.ist.psu.edu/50371.html;http://www.cs.cmu.edu/~necula/tr96-165.ps.gz` [cited 2007.08.30].

[OAY05] Karen Osmond, Alex Ahern, and Nobuko Yoshida. Distributed java project, 2005. Available from: `http://dj-project.sourceforge.net` [cited 2008.09.05].

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002. ISBN 0-262-16209-1.

[PMMR06] Gian Pietro Picco, Matteo Migliavacca, Amy L. Murphy, and Gruia-Catalin Roman. Distributed abstract data types. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (2)*, volume 4276 of *Lecture Notes in Computer Science*, pages 1594–1612. Springer, 2006. ISBN 3-540-48274-1. `doi:10.1007/11914952_40`.

[Poc16] H. C. Pocklington. The determination of the prime or composite nature of large numbers by Fermat's theorem. *Proceedings of the Cambridge Philosophical Society*, 18:29–30, 1914-1916.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus, March 19 1997. Available from: `http://citeseer.ist.psu.edu/26844.html;http://www.cs.indiana.edu/pub/pierce/pict-design.ps.gz` [cited 2007.08.30].

[Res07] Microsoft Research. F#: A succinct, type-inferred, expressive, efficient functional and object-oriented language for the .net platform, 2007. Available from: `http://research.microsoft.com/fsharp/fsharp.aspx` [cited 2008.10.01].

[SH00] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 214–227, January 2000.

[SLW$^+$04a] 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 UCAM-CL-TR-605, University of Cambridge Computer Laboratory, October 2004. Also published as INRIA RR-5329. 193pp. Available from: `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-605.html`.

[SLW$^+$04b] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute - high-level programming language design for distributed computation: Design rationale and language definition. Technical report, University of Cambridge and INRIA Rocquencourt, October 2004. Available from: `http://www.cl.cam.ac.uk/users/pes20/acute` [cited 2007.01.04].

[SSvP07] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *PPOPP*, page 271. ACM, 2007. ISBN 978-1-59593-602-8. `doi:10.1145/1229428.1229483`.

[SW99] Peter Sewell and Pawel T. Wojciechowski. Nomadic pict: Language and infrastructure design for mobile agents. October 1999. First International

Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents.

[Tuc97] Allen B. Tucker, editor. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997. Available from: `http://lucacardelli.name/Papers/TypeSystems.pdf`.

[U.S85] U.S. Department of Defense. Trusted computer systems evaluation criteria. (Orange Book) 5200.28-STD, National Computer Security Center, Fort Meade, MD, December 1985. Available from: `http://nsi.org/Library/Compsec/orangebo.txt` [cited 2008.10.10].

[Woj00] Pawel T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* PhD thesis, Wolfson College, University of Cambridge, March 2000.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.

[Zip08] Felipe Zipitría. PCR extension to the acute source code, 2008. Available from: `http://www.fing.edu.uy/inco/grupos/gsi/sources/acute-pcr/index.html` [cited 2008.10.01].

Index

A. Extension of Acute language

The Acute language, extended with PCR support, can be found in the following URL:`http://www.fing.edu.uy/inco/grupos/gsi/sources/acute-pcr/`.

There are two files listed there: the file called acute-pcr-src.tar.gz is the full source code. To compile the Acute interpreter, simply follow the instructions for Acute (`README.distro-src`). The other file, called acute-pcr-bin.tar.gz, has the full source code and includes a binary code, compiled for Linux in a self-contained package.

Basically, Acute needs FreshOCaml to run correctly. Therefore, we packaged FreshOCaml into a subdirectory of the distributed file.

Examples of the usage can be found in the 'demo' subdirectory.

For simplifying some infrastructure details to the programmer, we extended the command line syntax. The extended syntax is on Listing A.1

```
-coqc <filename>
   pcr: Use <filename> as COQ compiler (default: /usr/bin/coqc)
-tcb <filename>
   pcr: Use <filename> as TCB for coqc (default: none)
-prelude <filename>
   pcr: Use <filename> as prelude for COQ compiler (default: none)
```

Listing A.1: Acute command line extended syntax

The new options include an option "`-coqc`", which points to the path of the COQ compiler. This is used only if it is not installed in the default place in unix (`/usr/bin/coqc`). Option "`-tcb`" refers to a file where are listed, one per line, the directories in the local filesystem where trusted *Tactics* reside. In addition, option "`-prelude`" is a file that contains some default prelude for COQ.

Both files can be empty, but COQ will be limited only to tactics predefined (or no tactic at all).

B. Glossary

Abstract data type A set of data values and associated operations that are precisely specified independent of any particular implementation.

Abstraction safety defined operations on this type are the only means of interacting with values of this particular type.

Formal Verification A formal verifier shows that the code is not going to harm system resources by using provable, mathematical logic.

Functional programming Is a paradigm that treats computation as the evaluation of mathematical functions.

Marshalling is the process of gathering data and transforming it into a standard format.

Proof assistant an interactive computerised helper for developing machine proofs.

Proof Carrying Results a host consumer sends an untrusted part some computation to be done remotely. The untrusted part then returns the result of this computation, with a certificate that the computation has been done in a correct way.

Trusted Computing Base is everything in a computing system that provides a secure environment.

Unmarshalling is the marshalling reverse process, which transforms back data from standard format to its original form.

C. Demo source code

This appendix contains the source code presented in the demo. It is written here for the mere purpose of completing it when the demo is fully extended.

C.1. Demo examples

This example show the creation and pushing *of* a prime number to a host. This number comes equipped with a COQ proof that the receiver will use to certify the number as prime.

C.1.1. Prime number generator

The prime numbers are generated using the pocklington program, along with its certificate.

C.1.2. Sender example

Generate and send a prime, with its corresponding proof.

```
(* Prime module *)
includesource "prime.ac"
includesource "certified-prime.ac"

(* certificate is in certified-prime.ac *)

(* Generate a prime number to be sent to Alice *)
let p = Prime.generate(Prime.init) in
  IO.send(marshal "StdLib" p
    ("prime", certificate);;
```

Listing C.1: Sender's code

C.1.3. Receiver example

Receives and verifies the prime number.

```
(* Prime module *)
includesource "prime.ac"

(* Alice just receive a prime number *)
```

```
let p = (unmarshal (IO.receive ()):Prime.t)
    in
    Pervasives.print_endline "Prime␣number␣is:␣";
    Prime.print_prime p;
    Pervasives.print_newline ();;
```

Listing C.2: Receiver's code

D. Example: Breaking abstractions

As we mentioned before, we constructed a simple example for showing us that it was easy to circumvent abstraction-safety and change data to break abstractions.

```
module EvenCounter
  : sig
      type t
      val start:t
      val get:t->int
      val up:t->t
    end
  = struct
      type t=int
      let start = 0
      let get x = x
      let up x = 2+x
    end
  IO.send( marshal "StdLib" EvenCounter.start )
```

Listing D.1: EvenCounter example

We followed this steps to construct this example:

1. We executed the code in listing D.1, in the Acute with the flag '-production'. This showed the hashes created by the runtime to identify this particular type (EvenCounter).
2. At some point of the execution, the string is printed by Acute. This string represents

```
"marshalled␣(\n␣{␣␣},␣{␣},␣{},␣{},[␣0␣]_
{␣0#C067F3015C262E10C93BF33D00AE2023.t␣=␣int␣}^
{␣0#C067F3015C262E10C93BF33D00AE2023.t␣},\n
0#C067F3015C262E10C93BF33D00AE2023.t)"
```

   the information to be sent to the other process.
3. With the string obtained, we changed the number 0 between the square brackets, for a number 5 (clearly not even).
4. As the languague uses this string as an intermediate representation, it has no difference with any string, so we made use of this representation, modifying it. We made the following program

```
"marshalled␣(\n␣{␣␣},␣{␣},␣{},␣{},[␣5␣]_
{␣0#C067F3015C262E10C93BF33D00AE2023.t␣=␣int␣}^
{␣0#C067F3015C262E10C93BF33D00AE2023.t␣},\n
0#C067F3015C262E10C93BF33D00AE2023.t)"


(* Begin sender code *)

IO.send ( "marshalled␣(\n␣{␣␣},␣{␣},␣{},␣{},[␣5␣]_
{␣0#C067F3015C262E10C93BF33D00AE2023.t␣=␣int␣}^
{␣0#C067F3015C262E10C93BF33D00AE2023.t␣},\n
0#C067F3015C262E10C93BF33D00AE2023.t)" );;

(* End sender code *)
```

5. When the unmarshal is done, Acute outputs 5!. What we sent then is something like marshal StdLib 5: EvenCounter.t, but using a forged string. Because of this, type-checking was skipped on the sender side (because it is a simple string), and also the verification step on the receiver. This example was also tested with the Acute flag '-mttc (*) rttc: Do unmarshaltime typechecking', and worked fine (printed 5).

The receiving program used is on listing D.2.

```
(* Begin receiver code *)

module EvenCounter
   : sig
       type t
       val start:t
       val get:t->int
       val up:t->t
     end
  = struct
       type t=int
       let start = 0
       let get x = x
       let up x = 2+x
     end
  IO.print_int (EvenCounter.get
       (unmarshal (IO.receive ()):EvenCounter.t));;

(* End receiver code *)
```

Listing D.2: Receiver example

E. Acute Syntax

The Acute Technical Report includes this appendix which gives most of the Acute syntax. We show it here for reference. This is the fully type-annotated source language, including sugared forms, together with other non-source constructs that are needed to express the semantics. The implementation can infer many of the type annotations, and the *mode*, *withspec*, *likespec*, *vce*, *vne*, and *resolvespec* annotations on module and import default to reasonable values if omitted. The internal parts M, t and x of identifiers M_M, t_t and x_x are inferred by scope resolution.
Abstract names n **Store locations** l
Standard library constants (with arity) x^n

Kinds

$$K \ ::= \ \mathit{TYPE} \,|\, \mathit{EQ}(T)$$

Types

$$\begin{aligned} T \ ::= \ & \mathbf{int} \,|\, \mathbf{bool} \,|\, \mathbf{string} \,|\, \mathbf{unit} \,|\, \mathbf{char} \,|\, \mathbf{void} \,|\, T_1 * \cdots * T_n \,|\, T_1 + \cdots + T_n \,|\, T \to T' \,| \\ & T\,\mathbf{list} \,|\, T\,\mathbf{option} \,|\, T\,\mathbf{ref} \,|\, \mathsf{exn} \,|\, M_M.t \,|\, t \,|\, \forall t.T \,|\, \exists t.T \,|\, T\,\mathbf{name} \,|\,|\, T\,\mathbf{tie} \,| \\ & \mathbf{thread} \,|\, \mathbf{mutex} \,|\, \mathbf{cvar} \,|\, \mathbf{thunkifymode} \,|\, \mathbf{thunkkey} \,|\, \mathit{thunklet} \,|\, h.t \,|\, n \end{aligned}$$

Constructors

$$\begin{aligned} C_0 \ &::= \ \ldots \\ C_1 \ &::= \ ... \end{aligned}$$

Operators

$$\begin{aligned} op \ ::= \ & \mathbf{ref}_{\mathbf{T}} \,|\, (=_T) \,|\, (<) \ | \ (\leq) \,|\, (>) \ | \ (\geq) \ | \, \mathbf{mod} \,|\, \mathbf{land} \,|\, \mathbf{lor} \,|\, \mathbf{lxor} \,|\, \mathbf{lsl} \,|\mathbf{lsr} \,|\mathbf{asr} \\ & (+) \,|\, (-) \ | \ (*) \,|\, (/) \,| \ - \ |\, @_T \,|\, (\hat{}) \,|\, \mathbf{create\_thread}_{\mathbf{T}} \,|\, \mathbf{self} \ | \ \mathbf{kill} \,|\, \mathbf{lock} \,|\, \mathbf{try\_lock} \,| \\ & \mathbf{unlock} \,|\, \mathbf{create\_cvar} \,|\, \mathbf{wait} \,|\, \mathbf{signal} \,|\, \mathbf{broadcast} \,|\, \mathbf{exit}_{\mathbf{T}} \,| \\ & \mathbf{compare\_name}_{\mathbf{T}} \,|\, \mathbf{thunkify} \,|\, \mathbf{unthunkify} \end{aligned}$$

Expressions

$$\begin{array}{rcl} e & ::= & C_0 \,|\, C_1\, e \,|\, e_1 \;::\; e_2 \,|\, (e_1, .., e_n) \,|\, \mathbf{function}\, mtch | \,\mathbf{fun}\, mtch \,|\, l \,|\, op^n\, e_1 ... e_n | \\ & & x^n\, e_1 \ldots e_n \,|\, x \,|\, M_M.x \,|\, \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 \,|\, \mathbf{while}\, e_1\, \mathbf{do}\, e_2\, \mathbf{done} \,| \\ & & e_1 \,\&\&\, e_2 |\, e_1 \,||\, e_2 |\, e_1\, ;\; e_2 |\, e_1\, e_2 |\, e_1 \,\&\&\, e_2 |\, !_T e \,|\, e_1 \;:=_T\; e_2 |\, \mathbf{match}\, e\, \mathbf{with}\, mtch \,| \\ & & \mathbf{let}\, p \;=\; e_1\, \mathbf{in}\, e_2 \,|\, \mathbf{let}\, x : T\, p_1..p_n = e \,|\, \mathbf{let\,rec}\, x : T \;=\; \mathbf{function}\, mtch\, \mathbf{in}\, e \,| \\ & & \mathbf{let\,rec}\, x : T\, p_1..p_n = e_1\, \mathbf{in}\, e_2 |\, \mathbf{raise}\, e \,|\, \mathbf{try}\, e\, \mathbf{with}\, mtch \,|\, \Lambda\, t \;\rightarrow\; e \,|\, e\, T \,| \\ & & \{T, e\}\, \mathbf{as}\, T_1 \,|\, \mathbf{let}\, \{t, x\} = e_1 \mathbf{in} e_2 |\, \mathbf{marshal}\, e_1\, e_2 \;:\; T |\, \mathbf{unmarshal}\, e\, \mathbf{as}\, T | \\ & & \mathbf{fresh_T} |\, \mathbf{cfresh}_T |\, \mathbf{hash}(X.x)_T |\, \mathbf{hash}(T, e_2)_T |\, \mathbf{hash}(T, e_1, e_2)_T |\, \mathbf{swap}\, e_1\, \mathbf{and}\, e_2\, \mathbf{in}\, e_3 \,| \\ & & e_1\, \mathbf{freshfor}\, e_2 \,|\, \mathbf{support}_T \,|\, M_M @ x \,|\, \mathbf{nameoftie}\, e \,|\, \mathbf{valoftie}\, e \,| \\ & & \mathbf{namecase}\, e\, \mathbf{with}\, t, (x_1, x_2)\, \mathbf{when}\, x_1 = e \rightarrow e_2\, \mathbf{otherwise} \;\rightarrow e_3 |\, e_1 ||| \, e_2 | \\ & & n_T \,|\, h.x \,|\, e_1 \;:=_T^{\prime}\; e_2 \,|\, \mathbf{marshalz}\, s\, e \;:\; T \,| RET_T |\, SLOWRET_T \,|\, TERM \,| \\ & & \mathbf{op}(op^n)^n\, e_1 ... e_n |\, \mathbf{op}(x^n)^n\, e_1 ... e_n \,|\; [e]^T_{eqs} \;|\, \mathbf{resolve}(M_M.x, M'_{M'}, resolvespec) \,| \\ & & \mathbf{resolveblocked}(M_M.x, M'_{M'}, resolvespec) \end{array}$$

Matches and Patterns

$$\begin{array}{rcl} mtch & ::= & p \rightarrow e \,|\, (p \rightarrow e \,|\, mtch) \\ p & ::= & (\_ : T) \,|\, (x : T) \,|\, C_0 \,|\, C_1\, p \,|\, p_1 :: p_2 \,|\, (p_1, \ldots, p_n) \,|\, (p : T) \end{array}$$

Signatures and Structures

$$\begin{array}{rcl} sig & ::= & empty \,|\, \mathbf{val}\, x_x \;:\; T\, sig \,|\, \mathbf{type}\, t_t :\; K\, sig \\ & & Sig \;::=\; \mathbf{sig}\, sig\, \mathbf{end} \\ str & ::= & empty \,|\, \mathbf{let}\, x_x \;:\; T\, p_1, \ldots, p_n \;=\; e\, str \,|\, \mathbf{type}\, t_t =\; T\, str \\ & & Str \;::=\; \mathbf{struct}\, str\, \mathbf{end} \end{array}$$

Version and version constraint expressions

$$\begin{array}{rl} avne ::= \underline{n} \,|\, \underline{N} \,|\, h \,|\, mynameb & avce ::= ahvce \,|\, \underline{n} \\ vne ::= avne \,|\, avne.vne & dvce ::= avce \,|\, \underline{n} - \underline{n}' \,|\, - \underline{n} \,|\, \underline{n} - \,|\, * \,|\, avce.dvce \\ ahvce ::= \underline{N} \,|\, h \,| M_M & vce ::= dvce \,|\, \mathbf{name} = ahvce \end{array}$$

Source definitions and Compilation Units

$$\begin{array}{rcl} \mathit{sourcedefinition} & ::= & \textbf{module}\,\mathit{mode}\,M_M\ :\ \mathit{Sig}\,\textbf{version}\,\mathit{vne}\ =\ \mathit{Str.withspec} \\ & & \textbf{import}\,\mathit{mode}\,M_M\ :\ \mathit{Sig}\,\textbf{version}\,\mathit{vce}\,\mathit{likespec}\,\textbf{by}\,\mathit{resolvespec} = \mathit{Mo} \\ & & \textbf{mark}\,\mathit{MK} \\ & & \textbf{module}\,M_M\ :\ \mathit{Sig}\ =\ M'_{M'} \end{array}$$

$$\begin{array}{rcl} \mathit{mode} & ::= & \textbf{hash} \,|\, \textbf{cfresh} \,|\, \textbf{fresh} \,|\, \textbf{hash!} \,|\, \textbf{cfresh!} \\ \mathit{withspec} & ::= & \mathit{empty} \,|\, \textbf{with}\,!\mathit{eqs} \\ \mathit{likespec} & ::= & \mathit{empty} \,|\, \textbf{like}\,M_M \,|\, \textbf{like}\,\mathit{Str} \\ \mathit{resolvespec} & ::= & \mathit{empty} \,|\mathit{STATICLINK},\ \mathit{resolvespec} \,|\mathit{HereAlready},\ \mathit{resolvespec} \,| \\ & & \mathit{URI},\ \mathit{resolvespec} \\ M_0 & ::= & M_M \,|\, \mathit{UNLINKED} \end{array}$$

$$\begin{array}{rcl} \mathit{compilationunit} & ::= & \mathit{empty} \,|\, e \,|\, \mathit{sourcedefinition}\,;;\,\mathit{compilationunit} \,| \\ & & \textbf{includesource}\,\mathit{sourcefilename}\,;;\,\mathit{compilationunit} \,| \\ & & \textbf{includecompiled}\,\mathit{compiledfilename}\,;;\,\mathit{compilationunit} \end{array}$$

Compiled Definitions and Compiled Units

$$\begin{array}{rcl} \mathit{definition} & ::= & \textbf{cmodule}\,\ldots \,|\, \textbf{cimport}\,\ldots \,|\, \textbf{module}\,\textbf{fresh}\,\ldots \,|\, \textbf{import}\,\textbf{fresh}\,\ldots \,| \\ & & \textbf{mark}\,\mathit{MK} \\ \mathit{compiledunit} & ::= & \mathit{empty} \,|\, e \,|\, \mathit{definition}\,;;\,\mathit{compiledunit} \,| \end{array}$$

Marshalled value contents (marshalled values are strings that unmarshal to these)

$$\mathit{mv}\ ::=\ \textbf{marshalled}(E_n,\ E_s,\ s,\ \mathit{definitions},\ e,\ T)$$

Module names (hashes and abstract names)

$$\begin{array}{rcl} h & ::= & \textbf{hash}(\textbf{hmodule}_{\mathit{eqs}}\,M\ :\ \mathit{Sig}\,\textbf{version}\,\mathit{vne} = \mathit{Str}) \,| \\ & & \textbf{hash}(\textbf{himport}\,M\ :\ \mathit{Sig}_0\,\textbf{version}\,\mathit{vc}\,\textbf{like}\,\mathit{Str} \,|\, n \\ X & ::= & M_m \,|\, h \end{array}$$

Expression name values

$$\mathbf{n}\ ::=\ n_T \,|\, \textbf{hash}(h.x)_T \,|\, \textbf{hash}(T'.\underline{s})_T \,|\, \textbf{hash}(T',\underline{s},\mathbf{n})_T$$

(In the implementation all h and $\mathbf{n}$ forms can be represented by a long bitstring taken from $\mathbb{H}$, ranged over by $\underline{N}$.)

Type equation sets (the M_M forms occur in the source language)

$$eqs \quad ::= \quad \emptyset \,|\, eqs,\ X.t \approx T$$

Type Environments (for identifiers and store locations — not required at run-time in the implementation)

$$E \quad ::= \quad \mathit{empty} \,|\, E, x : T \,|\, E, l : T\,\mathsf{ref} \,|\, E, t : K \,|\, E, M_M : \mathit{Sig}$$

Type Environments (for global names — not required in the implementation)

$$\begin{aligned} E_n \quad ::= \quad & \mathit{empty} \,|\, E_n, n : \mathbf{nmodule}_{eqs} M : \mathit{Sig}_0\, \mathbf{version}\, vne = \mathit{Str} \,| \\ & E_n, n : \mathbf{nimport}\, M : \mathit{Sig}_0 \mathbf{version}\, vc\, \mathbf{like}\, \mathit{Str} \,| \\ & E_n, n : \mathsf{TYPE} \,|\, E_n, n : T\,\mathsf{name} \end{aligned}$$

Processes

$$P \quad ::= \quad 0 \,|\, (P_1 | P_2) \,|\, \mathbf{n} : \mathit{definitions}\, e \,|\, \mathbf{n} : \mathrm{MX}(\underline{b}) \,|\, \mathbf{n} : \mathrm{CV}$$

Single-Machine Configurations

$$\mathit{config} \quad ::= \quad E_n\,;\, \langle E_s,\ s,\ \mathit{definitions},\ P \rangle$$