INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Oz/K: *A kernel language for component-based open programming*

Michael Lienhardt, Alan Schmitt, Jean-Bernard Stefani

N° 6202

April 2007

Thème COM

Rapport de recherche

ISRN INRIA/RR--6202--FR+ENG

SN 0249-6399

Oz/K: A kernel language for component-based open programming

Michael Lienhardt, Alan Schmitt, Jean-Bernard Stefani

Thème COM — Systèmes communicants
Projets SARDES

Rapport de recherche n° 6202 — April 2007 — 75 pages

Abstract: Programming in a distributed and open environment remains challenging because it requires combining modularity, security, concurrency, distribution, and dynamicity. This has lead recently to interesting programming language developments such as Alice, Acute, Oz, JoCaml, ArchJava, etc, however the combination of all the above features with dynamicity, i.e. the ability to build and modify systems during execution, still remains an open question. In this paper, we propose an approach to open distributed programming that exploits the notion of locality, which has been studied intensively during the last decade, with the development of several process calculi with localities, including e.g. Mobile Ambients, Dπ, and Seal. We suggest to use the locality concept as a general form of component, that can be used, at the same time, as a unit of modularity, of isolation, and of mobility. Specifically, we introduce in this paper Oz/K, a kernel programming language, that adds to the Oz computation model a notion of locality borrowed from the Kell calculus. We present an operational semantics for the language, and several examples to illustrate how Oz/K supports open distributed programming.

Key-words: programming language, distributed programming, open programming, localities, component-based programming

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

OZ/K: Un langage noyau pour la programmation répartie à composants

Résumé : Programmer dans un environnement ouvert reste difficile car cela impose de considérer à la fois des questions de modularité, de sécurité, de concurrence, et de répartition. Ces questions ont donné lieu récemment au développement de plusieurs langages tels qu'Alice, Acute, OZ, JoCaml, ou ArchJava, mais il semble que combiner tous ces aspects avec la dynamicité, c'est-à-dire la possibilité de construire et de modifier des systèmes en cours d'exécution, reste une question ouverte. Dans ce rapport, nous proposons une approche de la programmation répartie ouverte qui exploite la notion de "localité" telle qu'elle a été étudiée dans plusieurs calculs de processus, comme, par exemple, le calcul des Ambients, le π-calculus réparti (Dπ), et le Seal calculus. Nous proposons d'utiliser le concept de localité comme une forme générale de composant logiciel, qui peut être exploité, à la fois, comme unité de modularité, d'isolation, et de mobilité. Nous présentons à cette fin, dans ce rapport, le langage OZ/K, un langage de programmation noyau qui étend le modèle de programmation d'OZ avec une notion de localité tirée du Kell calcul. Nous définissons une sémantique opérationnelle formelle pour le langage, et nous illustrons, grâce à plusieurs exemples, l'intérêt de OZ/K pour une programmation répartie ouverte.

Mots-clés : langage de programmation, programmation répartie, programmation ouverte, localités, programmation à composants

Contents

1 Introduction

Open environments involve distributed users that access and combine multiple services. These services interact, fail, and evolve constantly. Programming in such environments remains challenging because it requires, as pointed out in [91] by the designers of the Alice programming language, the combination of several features, notably: (i) *modularity*, i.e. the ability to build systems by combining and composing multiple elements; (ii) *security*, i.e. the ability to deal with unknown and untrusted system elements, and to enforce if necessary their isolation from the rest of the system; (iii) *distribution*, i.e. the ability to build systems out of multiple elements executing separately on multiple interconnected machines, which operate at different speed and under different capacity constraints, and which may fail independently; (iv) *concurrency*, i.e. the ability to deal with multiple concurrent events, and non-sequential tasks; and (v) *dynamicity*, i.e. the ability to introduce new systems, as well as to remove, update and modify existing ones, possibly during their execution.

Each of these features has been, and continues to be, the subject of active research on its own. Combining them into a coherent and practical programming language, however, is still an open question, despite interesting developments in the past two decades, including languages such as Acute [100], Alice [91], ArchJava [3], Classages [71], Erlang [10], E [80], Java [12], JoCaml [48], Kali Scheme [33], Klaim [18], Nomadic Pict [114], OZ [111], Scala [83]. For instance, the combination of strong objective mobility (i.e. the ability to move an executing component from one location to another) and dynamic linking with sandboxing (i.e. the ability to isolate an untrusted component from the rest of a computation, and to exercize discretionary control over its communication) is either not available in these different languages, or only through the use of relatively complex constructions and programming environment libraries with no formally defined semantics. Among these languages, Acute, Alice, and OZ (the latter with its MOZART environment [57, 93]) provide the most extensive support for open programming, but they still fall short, we argue below, of providing enough support for isolation and dynamic reconfiguration.

In this paper, we propose an approach to open programming that exploits the notion of *locality*. This notion has been studied in several families of process calculi such as Mobile Ambients [31], Dπ [58], Klaim [18], or the Seal calculus [32]. We suggest to use the locality concept as a primitive form of *component* that can be used simultaneously as a unit of *modularity*, of *isolation*, and of *passivation* (we call passivation the ability to freeze and marshall a component during its execution). Conflating these different kinds of units into a single notion provides a way to address the different concerns of open programming with few programming constructs.

Specifically, we introduce the OZ/K kernel programming language, that extends the OZ kernel language with a notion of locality, called *kell*[1], borrowed from the Kell calculus [96], together with a passivation operation, borrowed from the M-calculus [95]. The layered design of the OZ kernel language, and the fact that it supports multiple programming paradigms make it a good substrate for our study, namely the use of locality as a basis for open programming. Because of the multi-paradigm character of the OZ language, extending OZ with localities can provide guidance for similar combinations using different language substrates (e.g. Ocaml [88] for functional and object-oriented programming, Haskell [87] for functional programming and lazy evaluation).

With respect to OZ and MOZART, OZ/K makes a number of contributions: (i) it generalizes the pickling operation in MOZART (i.e. the ability to make values in the language persistent – e.g. for storing them in a file or for sending them in a message) to cover not only stateless values but also complete execution structures; (ii) it allows to define different distributed programming abstractions without depending on a single, pre-defined distribution semantics for the different language entities as is currently the case in MOZART; (iii) it enhances security in OZ through first-class isolation units, and the ability to program sandboxes and security wrappers; (iv) it extends the classical exception handling mechanisms in OZ with failure handling facilities that operate at the component level; and (v) it provides basic support for strong mobility and dynamic reconfiguration through passivation.

Technically, the main contributions of this paper are: (i) the introduction of an extension of the *kell* concept from the Kell calculus [96] and the Kell calculus with sharing [61], with the ability to control communication channels of subordinate kells; (ii) the introduction of a passivation operation, called *packing*, which generalizes the passivation operator of the M-calculus [95] to an execution model with a shared store and logic variables; (iii) the

[1] Localities in the Kell calculus are called *kells*, in a loose analogy to biological *cells*.

introduction of operations on *packed values* (i.e. values resulting from the packing of kells) that provide support for dynamic linking and component replacement; (iv) the introduction of failure handling mechanisms that can deal with thread and component-level failures; (v) a formal operational semantics for the addition of the above constructs to the OZ kernel language.

The paper is organized as follows. Section 2 motivates and introduces our approach in the design of OZ/K. Section 3 defines the abstract syntax and provides an informal overview of the OZ/K kernel language. Section 4 presents several simple examples of open programming in OZ/K. Section 5 defines a formal operational semantics for OZ/K. Section 6 discusses various design decisions and issues. Section 7 discusses related work. Section 8 concludes the paper.

2 Extending OZ for open programming

2.1 OZ and MOZART limitations

The OZ language and its MOZART environment already provide several features for open programming. These include in particular: first class *modules* (records that group together related language entities such as procedures) and *functors* (functions that take modules and functors as arguments, and return modules); *module managers*, that allow access to modules referenced by URLs; *pickles*, that can be used to save complete values (i.e. values that do not contain unbound variables) to files; *tickets*, that constitute references to arbitrary language entities; *connections*, that support the establishment of communication links between remote sites using tickets for cross-site references; a distributed semantics (described in [57, 93]) that assigns sites to certain languages entities such as variables, and cells, together with associated *communication protocols* tailored for achieving network transparency with the different kinds of language entities, namely stateless entities (e.g. base values, records, procedures, functors), and stateful entities (e.g. variables, cells). Despite these features, we can single out three main areas where OZ and MOZART fall short of supporting open programming: isolation, support for dynamic reconfiguration, and distribution semantics.

Isolation. Systems operating in an open environment should be ready to deal with unknown, potentially malicious components. A basic strategy to deal with untrusted components is to set up *sandboxes*, as formalized e.g. by the notion of *wrappers* in the Boxed-π calculus [102]. A sandbox is an execution context that isolates encapsulated computations from the rest of their environment, and that prevents unwanted or suspicious communication attempts. More generally, isolating different parts of a running system from one another is required for performance isolation and for preventing denial of service (e.g. to prevent a component interfering with the execution of another one merely through inordinate resource consumption).

The current OZ language and its MOZART environment fail to support sandboxes formalized as Boxed-π wrappers, which allow a strict control of communications between a module or component and its environment. For instance, while it is possible, through the subclassing of the base MOZART module manager, to forbid a downloaded module to access local resources on installation, it is not possible to control the communication of a module with its environment while it executes, and thus to prevent it from discovering – and accessing – forbidden resources in the process.

Support for dynamic reconfiguration. An open distributed environment is a highly dynamic one, where failures, updates, adaptations, and unplanned changes can occur all the time. A language for open distributed programming should provide the means to change a system's structure and behavior on-the-fly, with no need to stop the whole system in order to perform modifications. Dynamic reconfiguration typically involves: the ability to circumscribe the part of a system which needs changing (the *target*); the ability to suspend the execution of the target in a well-defined state; the ability to replace the suspended target by a different subsystem.

The higher-order character of the OZ language allows to program systems as collections of components (e.g. in the form of *port objects* as described in [111]), and to program these components so that their behavior include some operation to change their state (see for instance the upgradable compute server in Chapter 11 of [111]). However, it is not possible to suspend the execution of a component or to delete it (e.g. if some unwanted behavior like unwarranted resource consumption is detected), unless such behavior is already part of the component program. Thus replacing a faulty or malicious component that does not support the appropriate update behavior is not possible in OZ. In addition, it is not possible to capture as a value the state of an ongoing execution (e.g. to take a checkpoint or to reinstate a failed system from a saved checkpoint).

Distribution semantics. An open environment is essentially heterogeneous, with a wide variety of networks and protocols, supporting different communication semantics and providing different guarantees. Furthermore, depending on the application, different levels of distribution transparency and different views of a networked infrastructure may need to be provided. For instance, a deployment application will likely require an explicit view of the individual sites in the target network, so as to control the placement, installation, and configuration of different software components on different sites. This view may be quite detailed, depending on deployment requirements. For instance, one could consider separate spaces for different users, separate component containers for different applications, different tiers in site clusters, with different interconnection schemas, different subnetworks for fault-tolerance and enhanced performance, etc.

It is this very diversity that has lead the designers of the Acute language to abstain from incorporating in their language any specific means of remote interaction. In their words, "a general-purpose distributed programming language should not have a built-in commitment to any particular means of interaction" [101, 99]. The current MOZART environment relies on a predefined distribution semantics. We wish to avoid that dependency to keep in line with the above philosophy, and, in contrast to OZ and MOZART, to allow the definition of a distribution semantics and its supporting protocols within the language itself.

2.2 Our approach

To deal with the above issues, we extend the OZ kernel language with a locality construct. The aim is to provide a small and uniform formal basis for open programming capabilities that subsume those of the MOZART environment. As a consequence, open programming features in MOZART which are not expressible in the OZ kernel language (e.g. distribution protocols, or module placement), can now be defined in OZ/K. The OZ kernel language is built using a layered approach, with successive layers adding expressive power and capabilities. The first layer combines logic variables and higher-order procedures. The second layer adds explicit concurrency, in the form of threads. The third layer adds explicit state, in the form of updatable memory cells. The last layer adds lazy execution, in the form of by-need triggers. Our approach adds a new layer to the language, consisting of three main features: (i) a primitive form of component, which we call *kell*; (ii) a primitive operation for passivating kells, which we call *packing*; and (iii) a set of primitive operations for communication between kells, and for manipulating packed values.

A kell acts as a unit of modularity (kells encapsulate data and behavior behind well defined interfaces, called *gates*), a unit of isolation (a kell may fail independently of other kells, and a kell can act as a sandbox for its *subkells*, i.e. for kells that it contains), and a unit of reconfiguration (a kell can be passivated, independently from other kells, then moved, replaced, or deleted). The conflation of these different units in the single notion of kell is the key element of our approach. A kell encapsulates both *activity*, in the form of threads and other (sub) kells, and *state*, in the form of a private data store. Kells can thus be understood as hierarchically organized components, with the same granularity as port objects or active objects in OZ.

In order to achieve isolation, means of communication between kells are restricted to the emission and receipt of messages on *gates*, which are similar to channels in the (synchronous) π-calculus. As a consequence, logic variables, memory cells, and by-need triggers remain private to a kell and cannot be shared between different kells. This design choice is similar to the one made in the Erlang language, where processes, which are the unit of modularity and isolation, only communicate through mailboxes. It is also similar to the one made in the E language, where vats, which are units of concurrency and isolation, only communicate through asynchronous message exchanges (with futures). There are several reasons for this choice, including those well-documented in disfavor of shared state concurrency (see e.g. [69], [9] for a discussion in the context of the Erlang language, and [80] for a discussion in the context of the E language). The overarching consideration in OZ/K is to avoid any form of shared state between kells to guarantee isolation.

OZ/K does not come equipped with a predefined distribution semantics. Instead, kells provide a basic notion of separation, from which different forms of remote interaction can be built, in line with the Acute philosophy discussed above. Communication on gates, which takes the form of atomic rendez-vous, should thus be seen as local communication. Remote interaction in OZ/K can be modeled by a program mediating communications between two or more peer kells (communication can take place via gates between a thread situated in a kell and a thread situated in the immediate parent kell). The net effect of our approach is to replace the network awareness principle that presided to the design of the OZ distribution semantics described in [57, 93, 92], which assigns localities (called *sites*) to language constructs, by a *network independence* principle that makes localities explicit, and does not define a fixed semantics for interaction over a network. A consequence of this design principle is that the distributed semantics developed for OZ is no longer primitive, but can be implemented in OZ/K as a set of abstractions for distributed programming. As a result, an OZ/K virtual machine[2] does not embed any assumption concerning supporting network services. Previous work on a Kell calculus abstract machine [21] has showed that this was an effective approach.

One may ask why we did not consider adding this last layer to OZ as a library instead of language extension. The reason can be given as a three-pronged argument: (i) we wish to have a simple formal semantics for our kernel language; (ii) we consider that a library ought to be programmable (even if not actually implemented) in terms of its host language, so as to avoid introducing constructs that are not definable in the host language semantics; (iii) the isolation achieved by kells, and the passivation operation cannot strictly be expressed in OZ. Consideration (ii) ensures that different forms of remote interaction can be defined and understood by OZ/K programmers as programs that relay information between peer kells.

3 Syntax and overview

The OZ/K kernel programming language retains the OZ general computation model at its core (with some amendments), and extends it with a notion of component directly inspired by the notion of kell in the Kell calculus. We provide below a brief overview of the main constructs in OZ/K. We leave aside in this overview constructs pertaining to lazy evaluation (by-need synchronization).

3.1 OZ core

The basis for OZ/K is the OZ kernel language [111], featuring logical variables (single assignment variables), higher-order procedures, cells (which support multiple assignments), exception handling, concurrent threads, and by-need triggers. A tutorial on OZ is available online [56]. We just recall here the main constructs of the language. The OZ execution model consists of *dataflow threads* that operate on a *shared store*. Threads contain *statement* sequences and communicate through shared references in the store.

The syntax of the OZ kernel language constructs we use in this paper is given in Table 1, where `S` and its decorated variants denote statements; `P`, `X`, `Y`, `C`, and their decorated variants denote variable identifiers; `V` denotes base values (integers and literals – i.e. names or atoms); and `J` denotes patterns. We assume that in any statement defining a lexical scope for a list of variable identifiers, the identifiers in the list are pairwise distinct. Specifically, in statements of the form:

```
local X1 ... Xn in S end
proc{X X1 ... Xn} S end
case X of V(V1:X1 ... Vn:Xn) then S1 else S2 end
```

[2] The operational semantics developed in this paper constitutes the specification of an OZ/K virtual machine, which should be implemented by considering all the different OZ/K primitives, including communication on gates, as actions local to a single site.

we must have `Xi`$\neq$ `Xj`, for all $i \neq j, i, j \in \{1, \ldots, n\}$.

We use the term *variable identifier* to refer to syntactical entities that denote variables. We use the term *variables* to refer to single-assignment variables, or logical variables (semantical entities).

S ::=	**skip**	*empty statement*
|	S1 S2	*sequential composition*
|	**thread**{X} S **end**	*thread creation*
|	**local** X1 ... Xn **in** S **end**	*variable introduction*
|	X = Y	*imposing equality*
|	X = v	*binding to base value*
|	X = l(f1:X1 ... fn:Xn)	*binding to record*
|	{Unify X Y}	*unification*
|	**if** X **then** S1 **else** S2 **end**	*branch statement*
|	**case** X **of** J **then** S1 **else** S2 **end**	*pattern matching*
|	{NewName X}	*name creation*
|	**proc**{P X1 ... Xn} S **end**	*procedure definition*
|	{P X1 ... Xn}	*procedure call*
|	{IsDet X Y}	*testing bound status*
|	{NewCell X C}	*cell creation*
|	{Exchange C X Y}	*cell read-and-update*
|	{WaitNeeded X}	*by-need synchronization*
|	Y = !!X	*read-only variable*
|	**raise** X **end**	*exception handling*
|	**try** S1 **catch** X **then** S2 **end**	
|	{FailedValue X Y}	
|	...	

Table 1: Syntax: OZ core

The syntax for patterns is given in Table 2.

J	::=	V	*base value pattern*
	|	V(V1:X1 ... Vn:Xn)	*record pattern*
V	::=	v	*base value*
	|	!X	*base value of variable*

Table 2: Pattern syntax

Variables and values. References in the store are through *logic variables* (or *variables*, for brevity) that can be bound or unbound. An unbound variable does not yet refer to a value. A bound variable `X` refers to a definite value, which can be a base value (an integer, an atom or a name), or a record.

Atoms are values whose identity is determined by a sequence of printable characters. A record takes the form `lab(f1:X1 ... fn:Xn)`, where `lab` is the *label* of the record, `f1,...,fn` are the *features* of the record, and variables `X1,...,Xn` (which can be bound or unbound) are the *fields* of the record. Assume `R` is a record, with feature `f`. Record selection is written `R.f`, i.e. if `R = lab(... f:X ...)`, then `R.f` evaluates to `X`. Records are used to constructs usual concrete types. Thus, tuples are records with consecutive integer features, starting with `1`. A tuple `lab(X1 ... Xn)` corresponds precisely to the record `lab(1:X1 ... n:Xn)`. Lists are defined as tuples built from the atom `nil`, denoting the empty list, and the record label "|", which can be used as an infix operator. Thus, `H|T` denotes a list whose first element is `H`, and whose tail is the list `T`. Lists can also be written in extension: `[A1 ... An]` stands for the list of n elements `A1`, ..., `An`. Pairs can be built as tuples with label `#`, using an infix notation. Thus `X#Y` corresponds to the tuple `'#'(X Y)`.

New logical variables are introduced with the statement:

```
local X1 ... Xn in S end
```

where `S` is an arbitrary statement, and `X1,...,Xn` are the n new variables being introduced[3]. Variables just introduced are unbound. To bind a variable to a value, one can use equality statements of the forms (where `v` is an arbitrary base value, and `X1,...,Xq` are variables):

```
X = v
X = l(f1:X1 ... fq:Xq)
```

Note that in statement: `X = l(f1:X1 ... fq:Xq)` variables `X1,...,Xq` may be unbound. This allows potentially infinite data structures to be represented in OZ and OZ/K. Two variables `X` and `Y` can be constrained to be bound to the same value through the statement:

```
X = Y
```

Determining the bound or unbound status of a variable `X` is possible via the statement:

```
{IsDet X B}
```

which binds variable `B` to **`true`** or **`false`** if `X` is bound or not.

Names. Names are unforgeable constants, which are typically used to identify various execution entities, such as procedures and threads. There are three special names with reserved keywords: **`unit`**, **`true`**, **`false`**. Names **`true`** and **`false`** denote the boolean values true and false, respectively. The name **`unit`** is typically used as a synchronization token. Names can be created with the statement:

```
{NewName N}
```

which binds the variable `N` to a fresh name, guaranteed to be unique among OZ and OZ/K computations.

Cells. New assignable memory cells are created with the statement:

```
{NewCell X C}
```

which binds the variable `C` to a fresh cell, and stores variable `X` in the newly created cell. An atomic read-and-update of a cell is provided by the statement:

```
{Exchange C A N}
```

which atomically binds the content of cell `C` to variable `A`, and updates the content of cell `C` with variable `N`. Using `Exchange`, one can define an assignment operation to a cell `C`, noted `C:= X`, which updates the content of the cell `C` to the variable `X`, and an operation to access the content of a cell `C`, noted `X = @C`, which binds the content of cell `C` to variable `X`.

[3]We keep in OZ/K the OZ syntactic constraint that variables start with an upper case letter. A lexical token which is not a keyword and starts with a lower case letter is deemed to be an atom. Thus `Var` is an identifier for a variable, whereas `var` denotes the atom `'var'`.

Procedural abstraction. A procedure definition takes the form:

```
proc{P X1 ... Xn} S end
```

where the variable `P` is bound to the name of the newly created procedure, the identifiers `X1 ... Xn` correspond to the formal parameters of the procedure, and `S` is a statement that constitutes the body of the newly created procedure. Identifiers `X1 ... Xn` are bound in the procedure definition, and their scope is the body `S` of the procedure. A procedure definition can also be written:

```
P = proc{$ X1 ... Xn} S end
```

where `$` is an anonymous marker, to emphasize the fact that a procedure definition binds the name of the newly created procedure to variable `P`, and puts a procedure value (a closure) in the store. Note that formal parameters of a procedure can be input parameters (variables which are bound prior to the procedure execution) or output parameters (variables which are bound during the procedure execution). This means that a procedure may return any number of results, including none.

A call to the procedure named `P` takes the form:

```
{P A1 ... An}
```

where `A1 ... An` are variables corresponding to the actual parameters of the call. Since a variable can be bound to a procedure name, procedures in OZ and in OZ/K, are higher-order. For instance, the following program leads to an infinite execution:

```
local P in
  proc{P X} {X X} end
  {P P}
end
```

Control flow and concurrency. Statements can be composed sequentially. The statement: `S1 S2`, is the sequential composition of statement `S1` with statement `S2`. The empty statement is **`skip`**. The statement

```
thread{T} S end
```

creates a new thread that executes statement `S`, and binds its (freshly generated) name to variable `T`.

The statements above are non-blocking. The basic conditional statement in OZ and OZ/K:

```
if X then S1 else S2 end
```

blocks until variable `X` is bound to a boolean value `true` or `false` (and then executes statement `S1` or statement `S2`, respectively).

The pattern matching statement in OZ and OZ/K:

```
case X of J then S1 else S2 end
```

also blocks till variable `X` is bound to a value. The pattern `J` is then matched with this value. If the match is successful, i.e. if unification between the value and `J` succeeds, the pattern variables in `J` are bound and statement `S1` is executed. The identifiers in `J` that correspond to pattern variables are bound in the **`case`** statement; their scope is the statement `S1`. For instance, if `X` is bound to the record `rec(a:V1 b:V2)`, then the statement

```
case X of rec(a:X1 b:X2) then {P X1 X2} else skip end
```

evaluates to `{P V1 V2}` (pattern variables `X1` and `X2` are bound during pattern matching to `V1` and `V2`, respectively).

Exception handling in OZ is standard, and available through the statements

```
raise X end
try S1 catch X then S2 end
```

3.2 OZ/K constructs

To the OZ core, OZ/K adds three main elements: *kells*, *gates*, and *packing*. The syntax for the OZ/K-specific constructs is given in Table 3, where K, X, Y, Z, G denote variable identifiers.

S	::=	...	
	|	**kell**{K} S **end**	*kell creation*
	|	{NewGate X}	*gate creation*
	|	{Send G X}	*emitting message X on gate G*
	|	{Receive G X}	*receiving message X on gate G*
	|	{Open K G}	*grant kell K access to gate G*
	|	{Close K G}	*revoke access to gate G for kell K*
	|	{Pack K X}	*packing kell K*
	|	{Unpack X Y}	*unpacking packed value X*
	|	{Mark X Y Z}	*marking X with names in Y*
	|	{Status K X}	*get status of thread K*

Table 3: Syntax: OZ/K extensions

Kells. A *kell* is a computational location, i.e. a form of concurrent component, which associates a *named locality* to part of an OZ/K computation. Localities are organized in a tree where each node contains a (logically) private store and several running threads. Kells are created via statements of the form:

```
  kell{K} S end
```

where K is bound to the (freshly generated) name of the newly created kell. Statement S corresponds to the body of the kell. Upon creation of kell K, the execution of S starts in a new thread running within K. In order to ensure isolation, S must contain only *strict* variables (except for K which is bound during the creation of the kell). Strict variables are variables which are bound to strict values, i.e. values which, recursively, do not contain unbound variables. In effect, kells partition OZ/K computations into isolated subsets, organized in a tree, that can only communicate through *gates*.

As an example, consider the statement

```
kell{Server} {Serve In Out} end
```

This statement creates a new kell named Server. Once created, the kell starts executing its body (in this case, a call to the procedure Serve) in a new thread. Procedure Serve can be defined e.g. as follows:

```
proc{Serve In Out}
  Message Response Handle in
    proc{Handle M R} ... end
    {Receive In Message}
    {Handle Message Response}
    {Send Out Response}
    {Serve In Out}
end
```

Serve first receives a message Message on gate In. The message is then handled by procedure Handle, which returns a result Response. The result is then sent on gate Out, and procedure Serve calls itself recursively, which will trigger the handling of the following input message on gate In. This example illustrates that kells can typically be programmed in much the same way as active objects or port objects in OZ, or as processes in Erlang.

Gates and communication. A *gate* in OZ/K denotes an interaction point for a kell. It is similar to a π-calculus channel: communication is pairwise and bidirectional, and gate names can be sent across gates. A new gate `G` can be created via a call of the form:

```
{NewGate G}
```

Once the gate `G` has been created, it can be used to send values via the statement

```
{Send G X}
```

or to receive them, via the statement

```
{Receive G X}
```

Communication through gates is by atomic rendez-vous: a `Receive` statement is successful only if there is a matching `Send` statement available in a different thread. This mode of communication on gates, together with the isolation property, allows locality passivation (packing) to take place at any point in time during an execution. Having an atomic rendez-vous as a primitive form of communication allows to derive other forms of interaction, including ones that implement flow control between emitters and receivers. In particular, component connectors can be realized as kells that mediate communication between two or more peer kells. Only *strict* values can be sent through a gate. This restriction ensures that kells remain isolated during execution, and that gates form the only means of communication between kells. Communication on gates should be understood as local, i.e. as taking place on a single machine. Remote communication in OZ/K can be *modeled*, as illustrated in the next section, by programs that relay information between two or more peer kells, using two or more gates.

Controlling communication. In order to support sandboxing, kell boundaries can impose restrictions on communications. By default, communication may cross at most one kell boundary: it is allowed within a kell, and between a kell and its parent-kell. Direct communication on some gate `G` between two threads separated by more than one kell boundary is only allowed if every kell boundary crossed by the communication has this gate *opened*[4]. A gate `G` can be opened in the boundary of a kell `K` by its parent-kell using the procedure call

```
{Open K G}
```

To allow two sibling kells `K1` and `K2`, children of kell `K`, to communicate directly on a gate `G`, one has to open `G` for either `K1` or `K2`, for instance via this statement in kell `K`:

```
{Open K1 G}
```

To make a parent kell *transparent* for some or all of its child-kells, i.e. to allow all child-kells to communicate directly between them, or with the parent kell environment, one can use the key-word `all` to reference all the child-kells, and all the gates in a kell. Thus, the statement

```
{Open all all}
```

opens all the gates for all the children-kells of the current kell, whereas the statement

```
{Open K all}
```

opens all the gates for child-kell `K` of the current kell.

[4]The exact condition, defined formally in Appendix A, is a bit more complex than that, because it takes into account the base case where all gates are opened for communication between a thread in a child kell and a thread in a parent kell. Hence, when a gate `G` is opened for a child kell, all the threads in the child kell have the same communication possibilities, on gate `G`, than thread in the parent kell.

Packing and unpacking. `{Pack K V}` is the statement implementing passivation. It suspends the execution of the child `K` of the current kell and marshalls it, together with the relevant portion of the store, in a *packed value* bound to the variable `V`. Packed values can be modifed using the `Mark` operation. Specifically, `{Mark V1 R V2}` returns in `V2` the packed value `V1` modified according to the instruction given by tuple `R`. If `R=gate(G1 G2)`, the gate `G1` is replaced in `V2` by `G2`. If `R=prc(P Q)`, the procedure `P` is replaced by `Q`. A side-effect of `Mark` is that it prevents *marked* names to be changed during unpacking of the packed value, as described below. Thus, the statement `{Mark V1 gate(G G) V2}` only marks gate `G` to prevent it being renamed when unpacking `V2`.

The statement `{Unpack V R}` can be used to unpack a packed value `V`. Unpacking creates an execution structure similar to the one which has been packed, with new names for its gates, kells, and procedures, with the exception of the ones which have been marked. The new gate names are returned in the list of pairs `R`. The first element of a pair in `R` is the name of a gate in the packed value. The second element of a pair in `R` is the corresponding new name for the gate. A `Mark` operation on a packed value can be understood as a dynamic linking operation that connects a kell about to be unpacked to its new environment.

Failure handling. OZ has only classical exception handling. In the context of OZ/K, we need to deal with thread-level and kell-level failures. This is made possible by the detection of thread failures, via the `Status` statement. Briefly, the statement

```
{Status T X}
```

returns in `X` the termination status of thread `T`. More precisely, `X` gets bound to either `terminated`, if thread `T` has terminated successfully, or a failed value reflecting the unsuccessful termination of a thread. See Section 5 for details.

3.3 OZ values and syntactic conveniences

OZ values. We occasionnally employ in our examples procedures that do not belong to the OZ/K kernel language, but that can be defined in terms of the kernel language, and which belong to the OZ base environment. We refer the reader to [56, 40] for more details on the OZ base environment. In particular, we use the notion of *chunk*, which is a basic data type provided in OZ. A chunk behaves much like a record, except that its label is always a name (and not an atom), and it is not possible to obtain its list of features through the `Arity` operation.

Syntactic conveniences. We use syntactic conveniences to abbreviate OZ/K programs. Thus, variable introduction

```
local X1 ... Xn in S end
```

can be abbreviated as

```
X1 ... Xn in S
```

Also, variables can be both declared and initialized at variable introduction. Thus,

```
local X in
   X = 10 {P X}
end
```

can be abbreviated as

```
X = 10 in {P X}
```

Likewise, a procedure declaration of the following form, where `S` is some statement,

```
Pr in
 proc {Pr ...} ... end
 S
```

can be abbreviated as

```
proc {Pr ...} ... end in S
```

or

```
Pr = proc {$ ...} ... end in S
```

Nested case statements can be abbreviated using `[]` to discriminate between different cases in a case statement: for instance,

```
case X of
   a(X1 X2) then {P1 X1 X2}
[] b(Y1 Y2) then {P2 Y1 Y2}
else skip
end
```

abbreviates

```
case X of a(X1 X2) then {P1 X1 X2}
else
  case X of b(Y1 Y2) then {P2 Y1 Y2}
  else skip
  end
end
```

Tuples are record with integer features. They can be written with their features left implicit. Thus `X = r(X1 X2)` is the same as `X = r(1:X1 2:X2)`. Nested values can be written directly, without introducing variables to hold intermediate values. Thus, `X = rec(a:11 b:r(1 2))` abbreviates:

```
local X1 X2 X3 X4 in
  X1 = 11
  X2 = 1
  X3 = 2
  X4 = r(X1 X2)
  X = rec(aX1 b:X4)
end
```

We can use the wildcard "_" in places where a variable is needed but not subsequently used. Thus **thread**`{_} S` **end** abbreviates:

```
local X in thread{X} S end
```

We also abbreviate: **thread**`{_} S` **end** to the simpler: **thread** `S` **end**. Likewise, we abbreviate: **kell**`{_} S` **end** to: **kell** `S` **end**.

We often make use of a list version of the `Open` and `Close` primitives, writing for instance:

```
{Open K1 [G1 G2 G3]}
```

for the statements:

```
{Open K1 G1} {Open K1 G2} {Open K1 G3}
```

Finally we make use of the nested marker `$` to simplify the writing of expressions, i.e. statements that return a value. For instance, if `P` is a procedure of $n + 1$ arguments, that returns its result on the last argument, we can write: `X = {P A1 ... An $}` for: `{P A1 ... An X}`, and:

```
{P {P A1 ... An $} B2 ... Bn X}
```

for:

```
Y in {P A1 ... An Y} {P Y B2 ... Bn X}
```

4 Open programming in Oz/K

In this section, we present several simple examples that illustrate how Oz/K supports various open programming features. In the process, we discuss the motivation for the features presented.

4.1 Components

The kell construct provides a form of component that is close to the software architecture [50, 103] notion: encapsulation behind well defined interfaces (gates), separation between interface and implementation, first-class notion of *connector* for supporting interaction between components. Apart from usual software engineering considerations (e.g. software quality, ease of maintenance and evolution), an explicit software architecture is interesting to combat architecture erosion, to facilitate system configuration and assembly, and to automate system management functions, as demonstrated e.g. in architecture-based management approaches [24, 35].

To illustrate support for component-based concepts, we present below three examples: the first one illustrates changes in component implementation exploiting only standard Oz constructs; the second one shows how one can recover the notion of owned component interface using gates; the third one illustrates the use of kells as both components and connectors, and kell-based dynamic reconfiguration.

As a first example, consider the kell `Server`, created by the **`kell`** statement below.

```
kell{Server}
 ServerState = {NewCell unit $}
 Serve = proc{$ InGate OutGate Handler State}
    Message in
      {Receive Ingate Message}
      case Message of
         replace(NewServe) then {NewServe InGate OutGate ServerState}
      [] update(NewHandler) then {Serve InGate OutGate NewHandler ServerState}
      [] msg(Op Args Continuation) then
                        Response in
                         {Handler Op Args Response}
                         {Send OutGate resp(Continuation Response)}
                         {Serve InGate Outgate Handler ServerState}
      else skip end
 end
 Handle = proc{$ X Y Z} ... end
 in
  {Serve In Out Handle ServerState}
end
```

This kell corresponds to a component with two interfaces, gates `In` and `Out`, and an initial implementation given by the statement `{Serve In Out Handle ServerState}`. The cell `ServerState` holds the internal state of the `Server` component. The implementation of kell `Server` does a simple job: upon receipt on gate `In` of a message of the form `msg(Op As C)` (where `Op` denotes the name of the operation to perform, `As` is a list of arguments for the operation, and `C` is a continuation), the operation name and the arguments are passed to an internal procedure (initially, `Handle`) for evaluation; when the call to the procedure returns, the result `R` is send together with the continuation `C` as a response message on gate `Out`. In addition, the implementation (given by procedure `Serve`) can be changed partially, upon receipt of an `update` message, which changes only the internal `Handle` procedure, or completely, upon receipt of the `replace` message. The latter illustrates the separation that is achieved between interfaces (gates) and implementation (a call to a procedure taking gates as arguments).

Apart from the **`kell`** construct, the above example uses only standard Oz constructs. It illustrates what one may call *planned reconfiguration*. Reconfiguration in `Server` can take place, as a consequence of the receipt of `update` or `replace` messages, in between the handling of request messages of the form `msg(Op As C)`. The reconfiguration triggered by the `update` and `replace` messages is *planned* (or *subjective*) in the sense that the code of the `Server` component itself contains instructions for changing its own internal configuration. In OZ/K, the **`kell`** construct also supports *unplanned* (or *objective*) reconfigurations, by means of passivation. We illustrate this in the second component example below.

Before we come to this example, let us remark that the analogy with the usual notion of component is not perfect, for a component typically *owns* its ports or interfaces, which cannot be shared with other components. This is not the case with kells and gates, however it is possible to enforce a form of "gate ownership", by turning a gate into a unidirectional communication port and exporting only to the environment of a kell one "side" of a gate. We can do this using chunks as capabilities, as shown below.

```
proc{NewHalfGate Dir GI GO}
 G Anchor in
  {NewGate G} {NewName Anchor}
  proc{NSend X} {Send G X} end
  proc{NReceive X} {Receive G X} end
  proc{NOpen K} {Open K G} end
  case Dir of
     send then Z in {NewChunk r(send:NSend open:NOpen) GO}
                    {NewChunk r(receive:NReceive Anchor:Z) GI}
  [] recv then Z in {NewChunk r(receive:NReceive open:NOpen) GI}
                    {NewChunk r(send:NSend Anchor:Z) GO}
  else skip end
end
```

In the above code snippet, we define a new procedure `NewHalfGate` which creates two "half gates", one for receiving and one for emitting. The additional argument `Dir` to `NewHalfGate` specifies which capability is to be exported: if it is `send`, then the send capability is exported (notice how the ability to open the gate is attached to the send capability in this case, meaning that communication across kell boundaries can only be done by passing *this* capability). Note the extra feature `Anchor` that is added to the non-exportable half gate. The field associated with `Anchor` remains unbound, which prevents the corresponding chunk from being communicated outside the current kell, because of the strictness requirement on gate communication. This suffices to ensure that only the proper half gate can be known outside of the current kell, and that the other half remains only known inside the current kell. This ensures a form of ownership of the gate since only the originating kell can either send or receive on the private half gate.

Our second component example illustrates that component configurations can be organized hierarchically, and can be changed in an *objective* fashion by packing kells. Consider the following code:

```
proc{Link I O} M in {Receive I M}{Send O M}{Link I O} end

kell{Comp}
 I1 O1 I2 O2 I3 O3 Comp1 Comp2 Comp3 Con1 Con2 Con3 in
  {NewGate [I1 O1 I2 O2 I3 O3]}

  kell{Comp1} {Beh1 I1 O1} end
  kell{Comp2} {Beh2 I2 O2} end
  kell{Comp3} {Beh3 I3 O3} end
  kell{Con1} thread {Link In I1} end {Link O1 I3} end
  kell{Con2} {Link O3 Out} end
  {Open Con1 [In I1 O1 I3]} {Open Con2 [O3 Out]}

  {Receive G M}
  case M of switch
  then {Pack Con1 _}
       kell{Con3}
```

```
        thread {Link In I2} end
        {Link O2 I3}
      end
      open Con3 [In I2 O2 I3]
  else skip
  end
end
```

In the above example, kell `Comp` has three subcomponents `Comp1`, `Comp2` and `Comp3`, whose behavior is defined by three procedures, `Beh1`, `Beh2`, and `Beh3`, defined elsewhere. Initially, `Comp` is configured as a pipeline of two subcomponents, `Comp1` and `Comp3`, with the gate `In` linked to gate `I1`, gate `O1` linked to gate `I3`, and gate `O3` linked to gate `Out`. In this initial configuration, `Comp2` is not in use since its gates are not linked. Note that procedure `Link` acts as a channel between an input gate and an output gate, and that kells `Con1`, `Con2`, and `Con3` are used as *connectors*, that bind several gates at once (for instance, gates `In` and `I1`, as well as `O1` and `I3`, in the case of `Con1`). Upon receipt of the `switch` event on the `G` gate, defined elsewhere, the initial configuration is changed to a pipeline of `Comp2` and `Comp3`. The reconfiguration is effected by removing the connector `Cn1`, via packing, and by replacing it with a new connector kell, `Cn3`. Notice that the reconfiguration code above does not pay any consideration to the exact state of the components: in particular, the switch may lose messages being processed by `Comp1` and by `Con1` at the moment of the switch. Note also that, in this instance, one could have programmed the removal of `Con1` directly in OZ, e.g. by having its behavior dependent on checking whether a given variable, indicating termination, is bound or not. However, the use of packing here provides an example of unplanned reconfiguration, where the code of connector components is independent of external reconfiguration actions.

4.2 Distribution

As explained in Section 2.2, OZ/K has no built-in support for remote communications. However, because of their inherent separation, kells in can be used to *model* different sites, communicating using different communication semantics. For instance, here is a simple configuration, with two sites `Site1` and `Site2`, running programs `P1` and `P2` respectively. The kell `Net` acts like an interconnecting asynchronous network, relaying messages from one site to another: we suppose that `S1` (resp. `S2`) listen on the gate `In1` (resp. `In2`) and emit on `Out1` (resp. `Out2`).

```
kell{Site1} {P1} end
kell{Site2} {P2} end
kell{Net}
  Relay in
    proc{Relay G1 G2}
      M in
      {Receive G1 M} thread {Send G2 M} end {Relay G1 G2}
    end
    thread {Relay Out1 In2} end
    thread {Relay Out2 In1} end
end
{Open Net all}
```

The `Net` component simply relays messages that are sent on output gates `Out1` and `Out2` to their destination sites, designated by their addresses, i.e. input gates `In1` and `In2`. The `Relay` procedure simply forwards asynchronously (due to the triggering of a separate thread for forwarding messages to their destination) messages between two gates. The `Net` component is allowed to communicate on all gates with its siblings, namely sites `Site1` and `Site2`. Note also that `Site1` and `Site2` are not allowed to communicate directly with each other since there are no gates explicitly opened for them (as a result, threads in `Site1` and `Site2` can only communicate with threads in `Net`).

This (evidently simplistic) example illustrates how the separation between different loci of computation can be used to model a networked environment. Note that we encapsulated the network in a separate kell: this would allow us, for instance, to model failures of the `Net` component independent from failures of sites. A programmer can thus be provided with a semantics for distributed computation in terms of the OZ/K computation model. Importantly, this semantics can be adapted to different network environments, and arbitrary details of the supporting infrastructure revealed to the programmer, without having to change the language semantics. One can thus provide different abstractions to distributed programmers, depending on their needs, the network environment considered, and the level of distribution transparency required, as in [21].

4.3 Modules and dynamic linking

Modules. The notion of kell unifies notions of software modules and components, and packing provides both a generalization and a formal interpretation of the pickling construct provided by the MOZART environment for the OZ language. Consider for instance the following code, where `G` is a gate:

```
kell{Mod}
  P1 P2 M T in
    proc{P1 A} ... end
    proc{P2 A} ... end
    proc{T X} {Send G X}{T X} end
    M = module(op1:P1 op2:P2)
    {T M}
end
```

This code fragment creates a new kell `Mod` which simply defines two unary procedures, `P1` and `P2`, and gathers them in a record `M`, which is repeatedly sent on gate `G`. In effect, `M` corresponds to a software module that consists of just two procedures, accessed through the names `op1` and `op2`. To use the module is simple; just retrieve the module proper on gate `G`, and use the module's procedures through their advertised names, `op1` and `op2`:

```
  {Receive G Y}
  {M.op1 X}
```

Importantly, kell `Mod` can be packed and sent to a different site (another kell), so that the module can be made available there. For instance, assuming `Out` is a gate on a channel to a different site (as illustrated in Section 4.2), then:

```
  {Pack Mod Z}
  {Send Out Z}
```

illustrates how to marshall the kell `Mod` using the packing operation, and how to send the resulting packed value for use of the module at a different site.

Strong mobility and dynamic linking. Assume a distributed environment similar to the one in Section 4.2. Assume further that each site upholds the convention that the atom `service` denotes a local module, consisting of two operations `op1` and `op2` (like the module in the previous example), which have different implementations at each site. How can we ensure that code programmed to use the `service` module at one site can be moved safely to a different site and use the local `service` module implementation ? We cannot simply use the previous `Mod` construction: the variable `M` references the module available at gate `G`, which does not refer to the correct implementation at other sites. One solution is to ensure each copy of the module dynamically retrieves the local implementation upon each call, so as to take into account possible moves from clients of the module. The following code implements this:

```
kell{DynMod}
   T P1 P2 DynM Ploc1 Ploc2 Loc1 Loc2 in
     proc{P1 A} ... end
```

```
     proc{P2 A} ... end
     {NewName Loc1}{NewName Loc2}
     proc{T X} {Send G X}{T X} end
     proc{Ploc1 M} Z in {Receive G Z}{Z.Loc1 M} end
     proc{Ploc2 M} Z in {Receive G Z}{Z.Loc2 M} end
     DynM = {NewChunk m(op1:Ploc1 Loc1:P1 op2:Qloc2 Loc2:P2) $}
     {T DynM}
end
```

As before, the procedures `P1` and `P2` constitute the local implementation of the module's functionality, and the procedure `T` aes available the module `M` on gate `G`. Using the module `DynMod` remains similar to the previous example: just access the procedure through the module's features `op1` and `op2`. The features `Loc1` and `Loc2`, that store the local implementation of the procedures `P1` and `P2`, are not directly used by the client of the module. Access to the private features `Loc1` and `Loc2` is protected by gathering all the features of module `DynM` in a chunk.

Assume an agent, modelled as a kell, that moves from site to site, through a series of packing/sending/receiving/unpacking moves, and that requires access at each site to the `service` module. Now, as the agent moves from the site `S1` to the site `S2`, the gate where the module is available changes, raising the necessity to modify the reference of the gate in the agent's code. This modification of the agent's code is done using the procedure `Mark`, as presented in the next example (we suppose that the module is available at `G1` – resp. `G2` – at site `S1` – resp. `S2`):

```
%% Site 1
{Pack Agent Z}
{Send Out1 msg(service:G1 pack:Z)}

%% Site 2
Message K Agent in
 {Receive In2 Message}
 case Message of msg(service:OldGate pack:PackedAgent) then
   kell{Agent}
     {Mark PackedAgent gate(OldGate G2) K}
     {Unpack K _}
   end
 else skip end
```

The dynamic linking technique introduced above incurs an overhead at each call, since it requires retrieving the local copy of the module before the actual call. We can provide an optimized version of dynamic linking by changing directly procedures in packed values. Assume a module defined like `Mod` above, with local copies of the same form at different sites. We can optimize the transfer of a mobile agent and the execution of dynamically linked procedures, by proceeding as follows: before sending the agent (in packed form), replace the procedures from module `Mod` by place-holder ones; upon receiving the agent in packed form, replace the place-holder procedures by those of the local copy of `Mod`. Sending of the mobile agent `Agent` would look like this, assuming `Agent` designates a kell, `G` denotes the gate at which the module `Mod` is available, `service` is the well-known name under which `Mod` is known at the different sites, and `Out` denotes a gate for sending to the chosen remote site:

```
Z1 Z2 M PackedAgent P1 P2 PH1 PH2 in
  proc{PH1 A} skip end
  proc{PH2 A} skip end
  {Pack Agent Z1}
  {Receive G M}
  P1 = M.op1  P2 = M.op2
  {Mark Z1 prc(P1 PH1)  Z2}{Mark Z2 prc(P2 PH2) PackedAgent}
  {Send Out msg(s:service prc:[PH1 PH2] agent:PackedAgent}
```

Receiving and linking the mobile agent at the remote site, would look like this, assuming that `In` denotes a gate for receiving from the original site, and that `G` denotes the gate at which `Mod` is known at the receiving site:

```
Message M Z1 Z2 Agent P1 P2 K in
  {Receive In Message}
  case Message of msg(s:service prc:[PH1 PH2] agent:PackedAgent)
  then {Receive G M}
       P1 = M.op1  P2 = M.op2
       kell{Agent}
         {Mark PackedAgent prc(PH1 P1) Z1}{Mark Z1 prc(PH2 P2) K}
         {Unpack K _}
       end
  else skip end
```

In effect, we directly replace in the agent code the place-holder procedures `PH1` and `PH2` by the local procedures `P1` and `P2`. This solution for dynamically linking a mobile agent to local copies of a well-known module, has less overhead than the previous one, since there is no need to first retrieve the local module copy prior to invoking an operation of the module. Note that the above techniques for dynamic linking can be used in conjunction with a name server at each site. Provided that all sites agree of a single atom such as `service` to refer to this name server, then this is enough to bootstrap dynamic linking (and dynamic binding) of services referenced across sites using well-known names (e.g. atoms or strings).

4.4 Isolation

The kell construct provides the ability to build very configurable sandboxes. Consider the case of a plug-in of dubious origin. It is possible to isolate it in different ways. A first example is provided by the following code, which is a straightforward application of dynamic linking. In this case, the sandbox `Sandbox` allows communication of the plug-in only on the gate `G`, that correspond to the communications advertised as required by the plug-in, under the well-known name `service`. Once received, the plug-in is placed inside the new kell `K`, inside the sandbox. It is then marked with the local gate `G`, and unpacked. The double inclusion is necessary to avoid any communication of the plug-in with the environment of the sandbox, apart from communications on gate `G`.

```
Sandbox in
  {Receive In Message}
  case Message of msg(service:OldGate plugin:PlugIn) then
    kell{Sandbox}
     K P in
      kell{K} {Mark PlugIn gate(OldGate G) P}{Unpack P _} end
      {Open K G}
    end
  else skip end
```

The behavior of a sandbox can be more complex. For instance, we may allow the plug-in to request the opening of some gate for communication. The sandbox can then check the security of such an opening, using the procedure `Check`, and allow it or not. The control policy module can take the form of a procedure `Control` listening on a given gate identified by a well-known name such as `control`. The resulting sandbox can take the following form:

```
Check = proc{$ K X Y B} ... end
Control = proc{$ SandBoxedKell CtrlGate}
  {Receive CtrlGate Message}
  case Message of r(service:S gate:G returnGate:R)
  then B in
    {Check SandBoxedKell S G B}
    if B then {Open SandBoxedKell G} {Send R ok} else {Send R nok} end
  else skip end
  {Control SandBoxedKell CtrlGate}
end
```

```
in
  {Receive In Msg}
  case Msg of msg(control:G plugin:P) then
      kell{Sandbox}
        Ctrl SndBoxK P1 in
         {NewGate Ctrl}
         kell{SndBoxK} {Mark P gate(G Ctrl) P1}{Unpack P1 _} end
         {Control SndBoxK Ctrl}
      end
  else skip end
```

The encapsulation realized by the kell construct allows in particular to build wrappers as in the Boxed-π calculus [102]. For instance, we can build a simple *filtering wrapper* for some untrusted plugin, which requires the use of a `service`, where the required service is made available locally (after filtering) on gate `SV`.

```
Filter Msg Sandbox in
  proc{Filter G1 G2} ... end
  {Receive In Msg}
  case Msg of msg(service:PG plugin:P) then
    kell{Sandbox}
      K P1 G in
       {NewGate G}
       kell{K} {Mark P gate(PG G) P1} {Unpack P1 _} end
       thread {Filter G SV} end
    end
  else skip end
```

In this example, the procedure `Filter` acts as a partial relay between the gates `G` and `SV`, transmitting only *valid* messages and erasing the others.

4.5 Handling failures

Failure handling in OZ/K bears a strong similarity with failure handling in Erlang [9], and with a recent proposal for enhanced failure handling in OZ [39]. Units of failure in OZ/K are threads and kells. Handling a failure in a thread or a kell requires setting up an independent thread that can monitor state changes in the supervised thread or kell. Setting up a monitoring thread can be done as in the following program:

```
proc {NewMonThread Body Gate}
 M in
 thread{Th} {Body} end
 thread{M}
  S = {Status Th $} in
   case S of failed(Z) then {Send Gate thFail(Th Z)} else skip end
 end
end
```

The above program creates two threads, the monitoring thread `M`, and the monitored thread `Th`. The behavior of `M` is simple: it waits for `Th` to fail, and then notifies this failure on gate `G`. The program makes use of the operation `Status`, that returns the *execution status* of a thread. A thread execution status can essentially be in two states: active or failed. It is manifested by a 'read-only) variable that is either unbound, signifying that the thread is active, or bound to a failed value of the form `failed(X)`, signifying that the thread has failed with failure cause `X`.

It is also possible to force a kell to abort upon the occurrence of some failure in one of its threads, thereby obtaining a similar effect to process linking in Erlang, which causes a group of Erlang processes to fail together if one of the processes in the group fails. In our case, we can *link* threads by placing them in a kell and setting up an appropriate monitoring structure. This is illustrated in the following program, where two threads are linked in a kell, which is aborted as soon as one the two threads fails. The code snippet below also illustrates how kells themselves can be monitored for failure (in this case, a failure message is sent on the monitoring gate `MG`).

```
G K in
  {NewGate G}
  kell{K} {NewMonThread Beh1 G} {NewMonThread Beh2 G} end
  {Receive G M}
  case M of thFail(T Z) then {Pack K _} {Send MG kFail(K Z)}
  else skip end
```

5 Oz/K operational semantics

The operational semantics of the Oz/K kernel language is given in terms of a reduction relation $\rightarrow \subseteq (\texttt{Store} \times \texttt{Task})^2$, which is a binary relation on execution structures. We call *execution structure* an element of $\texttt{Store}\times\texttt{Task}$, i.e. a pair consisting of a store and a task. We assume given the following infinite countable and mutually disjoint sets: `Ident`, the set of variable identifiers, `Var` the set of logical variables, `Name` the set of names, `Atom` the set of atoms. We denote `Int` the set of integers, which we assume also disjoint from $\texttt{Ident} \cup \texttt{Var} \cup \texttt{Name} \cup \texttt{Atom}$. `Name` contains the following distinguished elements: **true**, **false**, **unit**, $\top$. The latter name denotes the name of the root of the kell tree (or *top-level kell*).

Statements. The set of statements, `Statement`, corresponds to the set of terms S given by the grammar productions in Table 1 and Table 3. The set of *extended statements*, $\texttt{Statement}^{\dagger}$, consists in the set of statements augmented with the set of terms S where logical variables are substituted to some or all variable identifiers in the term S, i.e.

$$\texttt{Statement}^{\dagger} = \texttt{Statement} \cup \{S\theta \mid S \in \texttt{Statement},\ \theta : \texttt{Ident} \rightarrow \texttt{Var}\}$$

The effect of a substitution θ on a statement S is defined in Section A.

Tasks. The set of tasks, `Task`, consists of elements $\mathcal{T}$ given by the following grammar (where η denotes a name, and S denotes an extended statement):

$$\mathcal{T} ::= \eta : T \quad | \quad \mathcal{T}\ \mathcal{T} \qquad\qquad \textit{tasks}$$
$$T ::= \langle\rangle \quad | \quad \langle S\ \ T\rangle \qquad\qquad \textit{thread stacks}$$

Intuitively, a task $\mathcal{T}$ is a multiset (parallel composition) of named threads $\eta : T$. As a notational convenience, when making explicit the structure of a named thread, we often elide the ':', thus "$\eta : \langle S\ \ T\rangle$" is often noted "$\eta\langle S\ \ T\rangle$".

Stores. The set of stores, `Store`, consists of elements σ given by the grammar in Figure 1, where x, y and their decorated variants range over variables; l, and its decorated variants range over literals (atoms and names); f and its decorated variants range over integers and literals; ξ, η, ζ and their decorated variants range over names.

A store consists in a conjunction (noted $\wedge$) of primitive assertions. Primitive assertions comprise:

- Variable in store assertions of the form x, which indicates that variable x is in the store domain, which we denote by: $x \in \texttt{dom}(\sigma)$.
- Variable bindings, of the form $x = V$, where x is a variable and V is some value (integer, atom, name, record, failed value, or packed value), or $x = y$, where x and y are both variables. Notice that the assertion $x = \texttt{pack}(\zeta, \mathcal{T}, \sigma, \mu)$ corresponds to a binding of a variable to a packed value, which happens only as a side-effect of passivation.
- Name bindings, of the form $\xi : T$, where T is some semantical value such as a procedure or a gate. Notice that a semantical value T embeds explicit type information about the nature of elements which are referred to by a name.

$\sigma ::= x$	*variable in store*
$\mid\ x = u$	*binding to base value*
$\mid\ x = l(f_1 : x_1, \ldots, f_n : x_n)$	*binding to record*
$\mid\ x = y$	*equality between variables*
$\mid\ x = \texttt{failed}(y)$	*failed value*
$\mid\ x = \texttt{pack}(\zeta, \mathcal{T}, \sigma, \mu)$	*variable bound to packed value*
$\mid\ \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\}S\ \texttt{end}$	*procedure value*
$\mid\ \xi : \texttt{thread}(x)$	*thread pointer*
$\mid\ \xi : \texttt{cell}(x)$	*cell value*
$\mid\ \xi : \texttt{kell}(\pi, x)$	*kell pointer*
$\mid\ \xi : \texttt{gate}$	*gate*
$\mid\ \texttt{need}(x)$	*needed variable*
$\mid\ \texttt{read}(x, y)$	*read-only variable*
$\mid\ \texttt{read}(x)$	
$\mid\ \texttt{in}(\xi, \zeta)$	*kell in kell*
$\mid\ \texttt{inth}(\xi, \zeta)$	*thread in kell*
$\mid\ \texttt{subg}(\xi, \zeta)$	*sub gate*
$\mid\ \sigma \wedge \sigma$	*store conjunction*
$\mu ::= \emptyset \ \mid\ \{\xi_1, \ldots, \xi_n\}$	*mark set*
$\pi ::= \emptyset$	*empty grant set*
$\mid\ \{\xi \cdot \gamma\}$	*gates γ for subkell ξ*
$\mid\ \xi \cdot \texttt{G}$	*all gates for subkell ξ*
$\mid\ \texttt{K} \cdot \gamma$	*gates γ for all subkells*
$\mid\ \texttt{K} \cdot \texttt{G}$	*all gates for all subkells*
$\mid\ \pi \cup \pi$	*grant set union*
$\mid\ \pi \setminus \pi$	*grant set difference*
$\gamma ::= \xi \ \mid\ \xi^{\texttt{r}} \ \mid\ \xi^{*}$	*gate, or gate and subordinates*

Figure 1: Store grammar

- Additional assertions; of the form pred(...), where pred is some predicate qualifying or relating names, or variables.

The packed value, thread, and kell constructs warrant some explanation. A packed value $v = \texttt{pack}(\kappa, \mathcal{T}, \sigma, \mu)$ comprises four elements: a suspended task $\mathcal{T}$, and its associated store σ; the name κ of the kell that has been packed; a set of names μ that have been *marked* to not be affected during unpacking. The suspended task and the store are captured by packing an executing kell. The set of names μ can include the name of the packed kell, the name of procedures in the packed kell, and the name of gates in the packed kell.

A thread binding $\tau : \texttt{thread}(x)$ refers to a thread named τ, whose execution status is given by the (read-only) variable x. While the thread is running, variable x remains unbound. If the thread terminates normally, then x becomes bound to the value terminated. If the thread fails, because of an uncaught exception, then x becomes bound to a failed value of the form $\texttt{failed}(y)$, where y is the exception that caused the thread to fail. The status of a thread can be obtained using the Status operation.

A kell binding takes the form $\kappa : \texttt{kell}(\pi, x)$, where κ is the name of the kell, π is called the *grant set* of the kell, and where the variable x contains the execution status of the kell. Variable x remains unbound while the kell is executing. It becomes bound to the value packed when the kell is packed. The status of a kell is not directly accessible, but it can be obtained indirectly when packing a kell, as the kell monitoring example in Section 4 illustrates. The packed status of a kell is checked when replacing a kell (only a packed kell can be replaced). The grant set corresponds to a specification of the gates that have been opened for communication to and from subkells of ξ. If $\kappa \cdot \gamma \in \pi$, then gate γ is opened to subkell κ for communication. If $\texttt{K} \cdot \gamma \in \pi$, then the gate γ is opened to all subkells. If $\kappa \cdot \texttt{G}$, then all gates are opened to subkell κ for communication. If $\kappa \cdot \gamma^{\texttt{r}} \in \pi$, then gate γ and all its immediate subordinate gates are opened to subkell κ for communication. If $\kappa \cdot \gamma^{*} \in \pi$, then gate γ and all its (recursively) subordinate gates are opened to subkell κ for communication. If $\texttt{K} \cdot \texttt{G} \in \pi$, then all gates are opened to all subkells.

The predicate need is used for lazy evaluation. Specifically, it is used for the definition of the WaitNeeded operation: the statement {WaitNeeded X} blocks until the variable X references is needed elsewhere in a computation. The predicate read is used for read-only variables. $\texttt{read}(x)$ just indicates that the variable x is read only, while $\texttt{read}(x, y)$ indicates that the variable x is read only and that its value, when it is determined, will be that of variable y. The predicate $\texttt{in}(\xi, \zeta)$ indicates that the kell ζ is located inside kell ξ. The predicate $\texttt{inth}(\xi, \zeta)$ indicates that the thread ζ is located inside kell ξ.The predicate $\texttt{subg}(\xi, \zeta)$ indicates that the gate ζ is a subordinate of the gate ξ.

Reduction relation. The reduction relation $\rightarrow$ is defined as the smallest subset of $(\texttt{Store} \times \texttt{Task})^2$ that satisfies the set of inference rules given in Section 5.1 below. To facilitate the comparison with the original OZ operational semantics, and to stay close to the definition of an abstract machine for OZ/K, we use the same approach to operational semantics than the one defined in chapter 13 of [111]. In particular, we use the same notational conventions, noting $\langle \sigma, \mathcal{T} \rangle \rightarrow \langle \sigma', \mathcal{T}' \rangle$ as $\dfrac{\mathcal{T} \;\|\; \mathcal{T}'}{\sigma \;\|\; \sigma'}$. The reduction rules take the form of inference rules of the form

$$\frac{\mathcal{T} \;\|\; \mathcal{T}'}{\sigma \;\|\; \sigma'} \quad \texttt{if } C$$

where C is some condition on $\mathcal{T}$, σ, $\mathcal{T}'$ and σ'. We use a number of abbreviations to simplify the writing of reduction inference rules. The table below gathers the different abbreviations. By definition, names and tasks that appear on the right column, but that do not appear on the left column, are different from the latter, and mutually distinct, but otherwise arbitrary. Intuitively, a decorated statement such as $S \mid_{\kappa}$ refers to a statement occurring within a thread of the kell named κ.

Rule	**Abbreviates**
$\dfrac{S}{\sigma} \Big\Vert \dfrac{S'}{\sigma'}$ `if` C	$\dfrac{\tau\langle S\ T\rangle}{\sigma} \Big\Vert \dfrac{\tau\langle S'\ T\rangle}{\sigma'}$ `if` C
$\dfrac{\mathcal{T}}{\sigma \models \phi} \Big\Vert \dfrac{\mathcal{T}'}{\sigma'}$ `if` C	$\dfrac{\mathcal{T}}{\sigma} \Big\Vert \dfrac{\mathcal{T}'}{\sigma'}$ `if` $C \wedge \sigma \models \phi$
$\dfrac{S \mid_\kappa}{\sigma} \Big\Vert \dfrac{S'}{\sigma'}$ `if` C	$\dfrac{\tau\langle S\ T\rangle}{\sigma \models \texttt{inth}(\kappa, \tau)} \Big\Vert \dfrac{\tau\langle S'\ T\rangle}{\sigma'}$ `if` C
$\dfrac{S_1 \mid_{\kappa_1}\ \ S_2 \mid_{\kappa_2}}{\sigma} \Big\Vert \dfrac{S_1'\ \ S_2'}{\sigma'}$ `if` C	$\dfrac{\tau_1\langle S_1\ T_1\rangle\ \ \tau_2\langle S_2\ T_2\rangle}{\sigma \models \texttt{inth}(\kappa_1, \tau_1) \wedge \texttt{inth}(\kappa_2, \tau_2)} \Big\Vert \dfrac{\tau_1\langle S_1'\ T_1\rangle\ \ \tau_2\langle S_2'\ T_2\rangle}{\sigma'}$ `if` C
$\dfrac{S \mid_\kappa\ \ \mathcal{T}}{\sigma} \Big\Vert \dfrac{S'\ \ \mathcal{T}'}{\sigma'}$ `if` C	$\dfrac{\tau\langle S\ T\rangle\ \ \mathcal{T}}{\sigma \models \texttt{inth}(\kappa, \tau)} \Big\Vert \dfrac{\tau\langle S'\ T\rangle\ \ \mathcal{T}'}{\sigma'}$ `if` C

Table 4: Abbreviations for reduction rules

5.1 Reduction rules

We give in this section the inference rules that define the reduction relation. Some of these rules make use of various auxiliary functions and relations. In this section, we only present them informally. They are defined formally in Appendix A. To simplify the presentation, we do not present straightforward failure rules, which specify that a given operation fails in case its arguments are ill-typed. Failure rules are given in Appendix B. We also do not present garbage collection or obvious optimization rules which can be applied during an OZ/K computation, for instance when packing a kell and its associated store.

In the following rules, unless explicitly stated otherwise: σ and its decorated variants denote stores; ϕ, ψ and their decorated variants denote assertions; ξ, η, ζ and their decorated variants denote names; κ and its decorated variants denote kell names; τ and its decorated variants denote thread names; γ and its decorated variants denote gate names; x, y, z, w, r, s and their decorated variants denote logical variables; u, v and their decorated variants denote values (i.e. integers, atoms, names, failed values, records); S and its decorated variants denote extended statements; T and its decorated variants denote thread stacks; $\mathcal{U}$, $\mathcal{T}$ and their decorated variants denote tasks.

Auxiliary functions and relations. The reduction relation depends on a number of functions and relations. The first relation is an equivalence relation, noted $\equiv$, between tasks and between stores. Intuitively, the equivalence relation between tasks asserts that the parallel operator between tasks is commutative, and associative, and that thread stacks that differ only from an alpha-renaming of variable identifiers in the statement they contain, are equivalent. The equivalence relation betwen stores asserts that the conjunction of stores is commutative and associative, and that two stores are equivalent if they entail the same assertions. The entailment relation between stores and assertions, noted $\models$, characterizes the logical assertions that can be derived from a store. The function `dom` takes a store σ as parameter and returns the set of all the names and variables used in σ. The predicate $\texttt{strict}_\sigma(\texttt{v})$ is true if `v` is a *strict* value in the store σ. We extend this predicate on variables x and statements `S`. The predicate $\texttt{strict}_\sigma(\texttt{S}, V)$ is true if all the variables in extended statement `S`, except those in V, are strict. The assertion $\texttt{access}_\sigma(\gamma, \kappa, \kappa')$ means that gate γ is accessible for communication between the kells κ and κ'. For this to be true, γ must have been opened for communication for all the kells on the path that connects κ and κ' in the kell tree[5], unless they are separated by at most one kell boundary. The function $\texttt{grant}_\sigma$ associates to the pair of variables (k, g) a set: the singleton pair corresponding to their names if k denotes a kell and g denotes a gate, and $\emptyset$ otherwise. The last auxiliary functions are $\texttt{subk}_\sigma$ and $\texttt{subth}_\sigma$: $\texttt{subth}_\sigma(\kappa)$ returns the set of names of all the threads contained by the kell κ and all its descendant kells, $\texttt{subk}_\sigma(\kappa)$ returns the set of names of all the descendant kells of kell κ.

Structural rules

The contextual rules define reductions under execution contexts, – namely parallel task contexts –, and for equivalent execution structures (i.e. pairs $\langle \texttt{store}, \texttt{task} \rangle$). Rules PAR and EQUIV are already present in OZ semantics.

$$[\textsc{Par}]\ \frac{\mathcal{T}\ \mathcal{U}}{\sigma} \Bigg\| \frac{\mathcal{T}'\ \mathcal{U}}{\sigma'} \ \text{ if } \ \frac{\mathcal{T}}{\sigma} \Bigg\| \frac{\mathcal{T}'}{\sigma'}$$

$$[\textsc{Equiv}]\ \frac{\mathcal{V}}{\gamma} \Bigg\| \frac{\mathcal{V}'}{\gamma'} \ \text{ if } \ \frac{\mathcal{U}}{\sigma} \Bigg\| \frac{\mathcal{U}'}{\sigma'} \ \text{ and } \mathcal{U} \equiv \mathcal{V}, \mathcal{U}' \equiv \mathcal{V}', \sigma \equiv \gamma, \sigma' \equiv \gamma'.$$

Sequential execution

$$[\textsc{Skip}]\ \frac{\tau\langle \textbf{skip}\ \ T\rangle}{\sigma} \Bigg\| \frac{\tau : T}{\sigma}$$

$$[\textsc{SeqTh}]\ \frac{\tau\langle (S_1\ S_2)\ \ T\rangle}{\sigma} \Bigg\| \frac{\tau\langle S_1\ \ \langle S_2\ \ T\rangle\rangle}{\sigma}$$

$$[\textsc{Nil}]\ \frac{\tau\langle\rangle}{\sigma \models \tau : \texttt{thread}(x) \wedge x = \bot} \Bigg\| \frac{}{\sigma \wedge x = \texttt{terminated}}$$

The rules for sequential execution are identical to those in OZ, modulo the introduction of named threads, and the garbage collection rule NIL, that replaces the equivalent rule NIL in the OZ semantics given in [111]. Notice that only the thread stack is collected: the termination status x of thread τ can be still be accessed.

Thread creation

$$[\textsc{NewTh}]\ \frac{\tau\langle \textbf{thread}\{x\}\ S\ \textbf{end}\ \ T\rangle}{\sigma \models x = \bot \wedge \texttt{inth}(\kappa, \tau)} \Bigg\| \frac{\tau : T\quad \tau'\langle S\ \ \langle\rangle\rangle}{\sigma \wedge \sigma'} \ \ \tau', w \notin \texttt{dom}(\sigma)$$

where

$$\sigma' \equiv x = \tau' \wedge \tau' : \texttt{thread}(w) \wedge w \wedge \texttt{read}(w) \wedge \texttt{inth}(\kappa, \tau')$$

Variable introduction

$$[\textsc{Var}]\ \frac{\textbf{local}\ X_1 \ldots X_n\ \textbf{in}\ S\ \textbf{end}}{\sigma} \Bigg\| \frac{S\{X_1 \rightarrow x_1, \ldots, X_n \rightarrow x_n\}}{\sigma \wedge x_1 \wedge \ldots \wedge x_n} \ \ x_i \notin \texttt{dom}(\sigma)$$

[5]The exact rule is a bit more subtle, but see Appendix A for a formal definition.

Read-only variables

$$[\textsc{Read}]\ \frac{x = !!y}{\sigma \models x = \bot} \Bigg\| \frac{\texttt{\textbf{skip}}}{\sigma \wedge z \wedge x = z \wedge \mathtt{read}(z, y)}\ \ z \notin \mathtt{dom}(\sigma)$$

$$[\textsc{ReadU}]\ \frac{}{\sigma \wedge \mathtt{read}(z, y) \models y \neq \bot} \Bigg\| \frac{}{\sigma \wedge z = y}$$

Binding

We adopt a different approach than MOZART for variable bindings: we consider only basic bindings, i.e. bindings where, in a statement $x = y$, only one of x or y, previously unbound, gets bound. This behavior is captured by the rules below, where v is a value (i.e. either a base value, integer or literal, a record, a failed value, or a packed value — and hence $v \neq \bot$). We note $\mathtt{rread}(x)$ the predicate $\mathtt{read}(x)\ \vee\ \exists y\ \mathtt{read}(x, y)$.

$$[\textsc{BindV}]\ \frac{x = v}{\sigma \models x = \bot \wedge \neg\mathtt{rread}(x)} \Bigg\| \frac{\texttt{\textbf{skip}}}{\sigma \wedge x = v}$$

The following rule, which defines the semantics of the variable equality statement, is actually a rule schema, with correlated ϕ and $\sigma(\phi)$, given by the table below the rule:

$$[\textsc{BindXY}]\ \frac{x = y}{\sigma \models \phi} \Bigg\| \frac{\texttt{\textbf{skip}}}{\sigma \wedge \sigma(\phi)}$$

where

ϕ	$\sigma(\phi)$
$x = v \wedge y = \bot \wedge \neg\mathtt{rread}(y)$	$x = y$
$y = v \wedge x = \bot \wedge \neg\mathtt{rread}(x)$	$x = y$
$x = \bot \wedge \neg\mathtt{rread}(x) \wedge y = \bot \wedge \neg\mathtt{rread}(y)$	$x = y$
$x = \bot \wedge y = \bot \wedge \mathtt{rread}(x) \wedge \neg\mathtt{rread}(y)$	$x = y \wedge \mathtt{read}(y)$
$x = \bot \wedge y = \bot \wedge \neg\mathtt{rread}(x) \wedge \mathtt{rread}(y)$	$x = y \wedge \mathtt{read}(x)$

$$[\textsc{BindR}]\ \frac{x = y.z}{\sigma \models x = \bot \wedge \neg\mathtt{rread}(x) \wedge y = l(f_1 : w_1 \ldots f_n : w_n)_{\mathtt{m}} \wedge z = f_i} \Bigg\| \frac{\texttt{\textbf{skip}}}{\sigma \wedge x = w_i}$$

Unification

The `Unify` operation is defined by the rules UNI and UNIF. It essentially implements the *naive tell semantics* discussed in chapter 13 of [111].

$$[\textsc{Uni}]\ \frac{\{\mathtt{Unify}\ x\ y\}}{\sigma} \Bigg\| \frac{\texttt{\textbf{skip}}}{\sigma'}\ \ \text{if } \sigma' = \mathtt{Unify}(x, y, \sigma) \not\equiv \bot$$

$$[\textsc{UniF}]\ \frac{\{\mathtt{Unify}\ x\ y\}}{\sigma} \Bigg\| \frac{\texttt{\textbf{raise}}\ \mathtt{error}(\mathtt{uni}(x\ y))\ \texttt{\textbf{end}}}{\sigma}\ \ \text{if } \mathtt{Unify}(x, y, \sigma) \equiv \bot$$

Equality between values

Two operations are possible on all values. They correspond to equality and inequality tests. Note that as syntactic convenience we write `X == Y` for `{Equal X Y $}` (i.e. the value of `X == Y` is the boolean returned by operation `Equal`), and `X \= Y` for `{NotEqual X Y $}`, where the function `NotEqual` can be defined as

```
proc{NotEqual X Y R}
  if {Equal X Y $} then R = false else R = true end
end
```

Note that the operation `Equal` suspends if the checks $x \equiv_\sigma y$ and $x \bowtie_\sigma y$ cannot take place.

$$[\textsc{EqTrue}]\ \frac{\{\texttt{Equal}\ x\ y\ r\}}{\sigma \models r = \bot} \Big\| \frac{\textbf{skip}}{\sigma \wedge r = \texttt{true}} \ \text{if } x \equiv_\sigma y$$

$$[\textsc{EqFalse}]\ \frac{\{\texttt{Equal}\ x\ y\ r\}}{\sigma \models r = \bot} \Big\| \frac{\textbf{skip}}{\sigma \wedge r = \texttt{false}} \ \text{if } x \bowtie_\sigma y$$

Status

The operation `Status` returns the status of a thread. This is captured by the following rule. Note that it is only possible to check the status of a thread that resides in the current kell: this is to ensure separation between kells.

$$[\textsc{Status}]\ \frac{\{\texttt{Status}\ x\ y\}\ |_\kappa}{\sigma \models y = \bot \wedge x = \tau \wedge \tau : \texttt{thread}(w) \wedge \texttt{in}(\kappa, \tau)} \Big\| \frac{\textbf{skip}}{\sigma \wedge y = w}$$

If statement

The if statement is identical to the original Oz if statement.

$$[\textsc{IfTrue}]\ \frac{\textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}}{\sigma \models x = \texttt{true}} \Big\| \frac{S_1}{\sigma}$$

$$[\textsc{IfFalse}]\ \frac{\textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}}{\sigma \models x = \texttt{false}} \Big\| \frac{S_2}{\sigma}$$

Case statement

The case statement is identical to the original Oz case statement.

$$[\textsc{Case}]\ \frac{\textbf{case}\ x\ \textbf{of}\ J\ \textbf{then}\ S_1\ \textbf{else}\ S_2}{\sigma} \Big\| \frac{S_1\theta}{\sigma} \ \text{if } \texttt{match}_\sigma(x, J) = \theta$$

$$[\textsc{CaseU}]\ \frac{\textbf{case}\ x\ \textbf{of}\ J\ \textbf{then}\ S_1\ \textbf{else}\ S_2}{\sigma} \Big\| \frac{S_2}{\sigma} \ \text{if } \texttt{match}_\sigma(x, J) = \bot$$

Names

$$[\textsc{NewName}]\ \frac{\{\texttt{NewName}\ x\}}{\sigma \models x = \bot} \Big\| \frac{\textbf{skip}}{\sigma \wedge x = \eta} \ \eta \notin \texttt{dom}(\sigma)$$

Procedure abstraction

Rules governing the introduction of procedures (PNEW), and procedures calls (PCALL) are similar to the ones in OZ. However, compared to OZ, we allow a dynamic update of procedure values, through the rule PREP.

The introduction of a new procedure is governed by the following rule.

$$[\text{PNEW}]\quad \frac{\textbf{proc}\{x\ X_1\dots X_n\}\ S\ \textbf{end}}{\sigma\models x=\bot}\ \Bigg\|\ \frac{\textbf{skip}}{\sigma\wedge x=\xi\wedge\xi:\texttt{proc}\{\$\ X_1\dots X_n\}S\ \texttt{end}}\quad \xi\notin\texttt{dom}(\sigma)$$

Calling a procedure is governed by the following rule.

$$[\text{PCALL}]\quad \frac{\{x\ x_1\dots x_n\}}{\sigma\models x=\xi\wedge\xi:\texttt{proc}\{\$\ X_1\dots X_n\}S\ \texttt{end}}\ \Bigg\|\ \frac{S\{X_1\rightarrow x_1,\dots,X_n\rightarrow x_n\}}{\sigma}$$

Replacing a procedure is governed by the following rule. The rule expects an already existing procedure under the name ξ, and just replaces the closure associated with the name ξ. The replacement procedure must have the same number of arguments than the replaced one.

$$[\text{PREP}]\quad \frac{\textbf{proc}\{x\ X_1\dots X_n\}\ S\ \textbf{end}}{\sigma\wedge\xi:Q\models x=\xi}\ \Bigg\|\ \frac{\textbf{skip}}{\sigma\wedge\xi:P}\quad \text{if } C$$

where

$$Q=\texttt{proc}\{\$\ X_1\dots X_n\}S'\ \texttt{end}\qquad P=\texttt{proc}\{\$\ X_1\dots X_n\}S\ \texttt{end}\qquad C\equiv\texttt{strict}_\sigma(S,\emptyset)$$

Checking determinacy

$$[\text{DETTRUE}]\quad \frac{\{\texttt{IsDet}\ x\ y\}}{\sigma\models x\neq\bot\wedge y=\bot}\ \Bigg\|\ \frac{\textbf{skip}}{\sigma\wedge y=\texttt{true}}$$

$$[\text{DETFALSE}]\quad \frac{\{\texttt{IsDet}\ x\ y\}}{\sigma\models x=\bot\wedge y=\bot}\ \Bigg\|\ \frac{\textbf{skip}}{\sigma\wedge y=\texttt{false}}$$

Cells

$$[\text{NCELL}]\quad \frac{\{\texttt{NewCell}\ x\ y\}}{\sigma\models y=\bot}\ \Bigg\|\ \frac{\textbf{skip}}{\sigma\wedge\xi:\texttt{cell}(x)\wedge y=\xi}\quad \xi\notin\texttt{dom}(\sigma)$$

$$[\text{ECELL}]\quad \frac{\{\texttt{Exchange}\ x\ y\ z\}}{\sigma\wedge x=\xi\wedge\xi:\texttt{cell}(t)\models y=\bot}\ \Bigg\|\ \frac{\textbf{skip}}{\sigma\wedge x=\xi\wedge\xi:\texttt{cell}(z)\wedge y=t}$$

Exception handling

$$[\text{TRYU}]\quad \frac{\textbf{try}\ S_1\ \textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end}}{\sigma}\ \Bigg\|\ \frac{S_1(\textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end})}{\sigma}$$

$$[\text{TRYC}]\quad \frac{\textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end}}{\sigma}\ \Bigg\|\ \frac{\textbf{skip}}{\sigma}$$

$$[\text{RAISEW}]\quad \frac{\tau\langle\textbf{raise}\ x\ \textbf{end}\ \ \langle S\ \ T\rangle\rangle}{\sigma}\ \Bigg\|\ \frac{\tau\langle\textbf{raise}\ x\ \textbf{end}\ \ T\rangle}{\sigma}\quad S\not\equiv\textbf{catch}\dots\textbf{end}$$

$$[\textsc{Raise}]\quad \frac{\tau\langle \textbf{raise}\ x\ \textbf{end}\ \ \langle S\ \ T\rangle\rangle}{\sigma} \,\Big\|\, \frac{\tau\langle S_2\{X \to x\}\ \ T\rangle}{\sigma} \quad S \equiv \textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end}$$

$$[\textsc{RaiseS}]\quad \frac{\tau\langle \textbf{raise}\ x\ \textbf{end}\ \ \langle\rangle\rangle}{\sigma \models \tau : \mathtt{thread}(w) \wedge w = \bot} \,\Big\|\, \frac{}{\sigma \wedge w = \mathtt{failed}(x)}$$

When a thread statement sequence has finished executing in a failed state, raised exceptions can be handled through the `Status` operation. Note that the RAISES rule is a form of garbage collection for abnormally terminated threads that complements the NIL garbage collection rule for normally terminated threads.

By-need synchronization

The rule for by-need synchronization is given below. It depends on the relation need, which is defined below.

$$[\textsc{WaitN}]\quad \frac{\{\mathtt{WaitNeeded}\ x\}}{\sigma \models \mathtt{need}(x)} \,\Big\|\, \frac{\textbf{skip}}{\sigma}$$

The predicate $\mathtt{need}(x)$ is added to the store according to the following rules:

$$[\textsc{Need}]\quad \frac{S}{\sigma \not\models \mathtt{need}(x)} \,\Big\|\, \frac{S}{\sigma \wedge \mathtt{need}(x)} \quad \text{if } \mathtt{need}_\sigma(S, x)$$

$$[\textsc{NeedD}]\quad \frac{}{\sigma \models x \neq \bot} \,\Big\|\, \frac{}{\sigma \wedge \mathtt{need}(x)} \quad \text{if } \sigma \not\models \mathtt{need}(x)$$

The assertion $\mathtt{need}_\sigma(S, x)$ is true if and only if the following conditions hold:

1. $\sigma \models x = \bot$.
2. No reduction is possible for S with store σ.
3. There exists a set β of variable bindings such that $\sigma' \wedge \beta$ is consistent and a reduction is possible for S with store $\sigma' \wedge \beta$.
4. For all β satisfying the above condition, $\sigma' \wedge \beta \models x \neq \bot$.

Failed values

The rule for the creation of failed values is given below.

$$[\textsc{FailC}]\quad \frac{\{\mathtt{FailedValue}\ x\ y\}}{\sigma \models y = \bot} \,\Big\|\, \frac{\textbf{skip}}{\sigma \wedge y = z \wedge z = \mathtt{failed}(x)} \quad z \notin \mathtt{dom}(\sigma)$$

The second rule for failed values ensures that needing a failed value raises an exception.

$$[\textsc{FailW}]\quad \frac{S}{\sigma \models y = \mathtt{failed}(x)} \,\Big\|\, \frac{\textbf{raise}\ x\ \textbf{end}}{\sigma} \quad \text{if } \mathtt{need}_\sigma(S, y)$$

Strictness check

The rules for checking whether a value is strict or not are given below.

$$[\textsc{StrictTrue}]\quad \frac{\{\mathtt{IsStrict}\ x\ y\}}{\sigma \models y = \bot} \,\Big\|\, \frac{\textbf{skip}}{\sigma \wedge y = \mathtt{true}} \quad \text{if } \mathtt{strict}_\sigma(x)$$

$$[\textsc{StrictFalse}]\quad \frac{\{\mathtt{IsStrict}\ x\ y\}}{\sigma \models y = \bot} \,\Big\|\, \frac{\textbf{skip}}{\sigma \wedge y = \mathtt{false}} \quad \text{if } \neg\mathtt{strict}_\sigma(x)$$

Gate abstraction

The rules for the creation of new gates are given below. The second rule creates a gate that is subordinate to an existing one.

$$[\textsc{NewG}]\quad \frac{\{\texttt{NewGate}\ x\}}{\sigma \models x = \bot} \;\Bigg\|\; \frac{\textbf{skip}}{\sigma \wedge x = \gamma \wedge \gamma : \texttt{gate}} \quad \gamma \notin \texttt{dom}(\sigma)$$

$$[\textsc{NewGS}]\quad \frac{\{\texttt{NewGate}\ x\#z\}}{\sigma \models z = \bot \wedge x = \gamma \wedge \gamma : \texttt{gate}} \;\Bigg\|\; \frac{\textbf{skip}}{\sigma \wedge z = \gamma' \wedge \gamma' : \texttt{gate} \wedge \texttt{subg}(\gamma, \gamma')} \quad \gamma' \notin \texttt{dom}(\sigma)$$

The rule COM governs communication through gates.

$$[\textsc{Com}]\quad \frac{\{\texttt{Send}\ g\ x\}\ |_{\kappa} \quad \{\texttt{Receive}\ h\ y\}\ |_{\kappa'}}{\sigma \models y = \bot \wedge \phi} \;\Bigg\|\; \frac{\textbf{skip} \quad \textbf{skip}}{\sigma \wedge y = x} \quad \text{if } \texttt{strict}_{\sigma}(x) \wedge \texttt{access}_{\sigma}(\gamma, \kappa, \kappa')$$

where

$$\phi \equiv g = \gamma \wedge h = \gamma \wedge \gamma : \texttt{gate}$$

Opening and closing

The ability for a kell to communicate with its environment is governed by the `Open` and `Close` operations. Operation `Open` opens a gate for communication for a subkell of the current kell, whereas `Close` closes this gate for communication. There are thus two prerequisites for a successful communication: (i) knowing a gate name, and (ii) having an access path established (through previous `Open` operations) to cross the required kell boundaries. Note that both arguments to primitives `Open` and `Close` can take the value `all`. If the first argument is `all`, this means that the gate specified in the second argument is opened or closed to all children of the current kell. If the second argument is `all`, this means that all the gates are opened, or closed, to the subkell specified in the first argument.

The rules that define the semantics of operations `Open` and `Close` are given below.

$$[\textsc{Open}]\quad \frac{\{\texttt{Open}\ k\ g\}\ |_{\kappa}}{\sigma \wedge \kappa : \texttt{kell}(\pi, w)} \;\Bigg\|\; \frac{\textbf{skip}}{\sigma \wedge \kappa : \texttt{kell}(\pi \cup \texttt{grant}(\sigma, k, g), w)} \quad \text{if } \texttt{grant}(\sigma, k, g) \neq \emptyset$$

$$[\textsc{Close}]\quad \frac{\{\texttt{Close}\ k\ g\}\ |_{\kappa}}{\sigma \wedge \kappa : \texttt{kell}(\pi, w)} \;\Bigg\|\; \frac{\textbf{skip}}{\sigma \wedge \kappa : \texttt{kell}(\pi \setminus \texttt{grant}(\sigma, k, g), w)} \quad \text{if } \texttt{grant}(\sigma, k, g) \neq \emptyset$$

Kell abstraction

The rules pertaining to the kell abstraction deal with the creation and the replacement of kells. The rule for kell creation is similar to the rule for thread creation. It creates a new kell as well as a new thread which begins executing the body of kell statement:

$$[\textsc{NewKell}]\quad \frac{\textbf{kell}\{y\}\ S\ \textbf{end}\ |_{\kappa}}{\sigma \models y = \bot} \;\Bigg\|\; \frac{\textbf{skip} \quad \tau'\langle S\ \ \langle\rangle\rangle}{\sigma \wedge \sigma'} \quad \text{if } C$$

where

$$C \equiv \kappa', \tau', w, r \notin \texttt{dom}(\sigma) \wedge \texttt{strict}_{\sigma}(S, \{y\})$$

$$\sigma' \equiv y = \kappa' \wedge \kappa' : \texttt{kell}(\emptyset, w) \wedge w \wedge \texttt{read}(w) \wedge \tau' : \texttt{thread}(r) \wedge r \wedge \texttt{read}(r) \wedge \texttt{inth}(\kappa', \tau') \wedge \texttt{in}(\kappa, \kappa')$$

The rule for kell replacement is similar to the rule of procedure replacement. It allows the replacement of a silent (i.e. non running) kell by a new one while preserving the original kell name. A side effect of this replacement is to change the status of the replaced kell to active (`run`) again.

$$[\textsc{KRep}]\quad \frac{\texttt{kell}\{y\}\ S\ \textbf{end}\ |_{\kappa}}{\sigma \wedge \kappa' : \texttt{kell}(\pi, w) \models \phi} \,\Bigg\|\, \frac{\textbf{skip}\qquad \tau'\langle S\ \ \langle\rangle\rangle}{\sigma \wedge \sigma'}\ \text{ if } C$$

where

$$C \equiv \tau', r, s \notin \texttt{dom}(\sigma) \wedge \texttt{strict}_{\sigma}(S, \emptyset)$$
$$\phi \equiv y = \kappa' \wedge w = \texttt{packed} \wedge \texttt{in}(\kappa, \kappa')$$
$$\sigma' \equiv \kappa' : \texttt{kell}(\pi, s) \wedge s \wedge \texttt{read}(s) \wedge \tau' : \texttt{thread}(r) \wedge r \wedge \texttt{read}(r) \wedge \texttt{inth}(\kappa', \tau')$$

Packed values

Packed values can be modified by means of the `Mark` operation. The `Mark` operation takes as input a change instruction, in the form of a pair of names. The first name of the input pair specifies the name of the gate or procedure to replace in the packed value. The second name of the input pair specifies the name of the replacement gate or procedure. A side effect of the operation is that gates or procedures that have thus *marked* do not get renamed upon unpacking. The first rule concerns the replacement of gates.

$$[\textsc{MarkG}]\quad \frac{\{\texttt{Mark}\ z\ \texttt{gate}(x\ y)\ \ p\}}{\sigma \models \phi \wedge p = \bot} \,\Bigg\|\, \frac{\textbf{skip}}{\sigma \wedge p = \texttt{pack}(\omega, \mathcal{T}\theta, \sigma'\theta, \mu \cup \{\gamma\})}\ \text{ if } C$$

where

$$C \equiv \sigma' \models \gamma' : \texttt{gate}\ \wedge\ \theta = \{\gamma' \rightarrow \gamma\}$$
$$\phi \equiv z = \texttt{pack}(\omega, \mathcal{T}, \sigma', \mu) \wedge x = \gamma' \wedge \gamma' : \texttt{gate} \wedge y = \gamma \wedge \gamma : \texttt{gate}$$

The next rule deals with the replacement of a procedure inside a packed value. The operation is similar to the replacement of gates, and has an effect similar to the rule PREP that governs the replacement of procedures, except this replacement takes place inside a packed value.

$$[\textsc{MarkP}]\quad \frac{\{\texttt{Mark}\ z\ \texttt{prc}(x\ y)\ \ p\}}{\sigma \models \phi \wedge p = \bot} \,\Bigg\|\, \frac{\textbf{skip}}{\sigma \wedge p = \texttt{pack}(\omega, \mathcal{T}\theta, \sigma_2, \mu \cup \{\xi\})}\ \text{ if } C$$

where

$$\phi \equiv z = \texttt{pack}(\omega, \mathcal{T}, \sigma_1, \mu) \wedge x = \xi' \wedge \xi' : \texttt{proc}\{\$\ X_1 \ldots X_n\}S'\ \texttt{end} \wedge y = \xi \wedge \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\}S\ \texttt{end}$$
$$C \equiv \theta = \{\xi' \rightarrow \xi\} \wedge \texttt{strict}_{\sigma}(S, \emptyset)\ \wedge\ \sigma_2 \equiv \sigma_1\theta \wedge \xi : \texttt{proc}\{\$\ X_1 \ldots X_n\}S\ \texttt{end}$$

Packing

The rule for packing is given below. Notice that packing implies passivating the target kell, together with all of its subkells. Packing produces a *packed value*, which encapsulates the part of the current execution structure corresponding to the target kell. The set of marks of the resulting packed value is initially empty.

$$[\textsc{Pack}]\quad \frac{\{\texttt{Pack}\ x\ y\}\ |_{\kappa}\quad \tau_1 : T_1 \ldots \tau_n : T_n}{\sigma \models \bigwedge_{i=1}^{m} \phi_i \wedge y = \bot \wedge \phi} \,\Bigg\|\, \frac{\textbf{skip}\quad \emptyset}{\sigma \wedge \sigma'}\ \text{ if } C$$

where

$$C \equiv \texttt{subth}_{\sigma}(\kappa_0, \{\tau_1, \ldots, \tau_n\}) \wedge \texttt{subk}_{\sigma}(\kappa_0, \{\kappa_1, \ldots, \kappa_m\})$$
$$\phi \equiv x = \kappa_0 \wedge \kappa_0 : \texttt{kell}(\pi, z) \wedge z = \bot \wedge \texttt{in}(\kappa, \kappa_0) \qquad \phi_i \equiv \kappa_i : \texttt{kell}(\pi_i, w_i) \wedge w_i = \bot$$
$$\sigma' \equiv \bigwedge_{i=1}^{m} w_i = \texttt{packed} \wedge y = \texttt{pack}(\kappa_0, \mathcal{T}, \sigma, \emptyset) \wedge z = \texttt{packed} \qquad \mathcal{T} \equiv \tau_1 : T_1\ \ldots\ \tau_n : T_n$$

The rule for unpacking is given below. Unpacking creates an execution structure which is similar to the packed one, except all the variables and all the non-marked names in the packed structure are renamed to avoid any potential conflict between the current store, σ, and the unpacked one, σ'. In addition, unpacking returns a list of pairs, called the *name list*. The first elements ξ_i pairs in the name list are all the gate names that appear in the packed value. The second element $\xi_i\theta$ of a pair in the name list is the new name which has been substituted to the the first element of the pair during unpacking.

$$[\text{UNPACK}]\ \frac{\{\texttt{Unpack}\ y\ x\}\ |_\kappa \qquad \emptyset}{\sigma \models \kappa : \texttt{kell}(\pi, z) \wedge x = \bot \wedge y = \texttt{pack}(\kappa_0, \mathcal{T}, \sigma', \mu)} \left\| \frac{\textbf{skip} \quad \mathcal{T}\theta\theta'}{\sigma \wedge \sigma'''} \right. \text{ if } C_1 \wedge C_2$$

where

$$C_1 \equiv (\texttt{dom}(\theta) = \texttt{dom}(\sigma') \setminus \mu) \wedge (\sigma \wedge \sigma''' \not\equiv \bot) \wedge \forall l \in \texttt{ran}(\theta), l \notin \texttt{dom}(\sigma, \sigma')$$
$$C_2 \equiv (\sigma' \equiv \sigma'' \wedge \kappa_0 : \texttt{kell}(\pi', z')) \wedge \theta' = \{\kappa_0 \to \kappa\} \wedge \{\xi_1, \ldots, \xi_n\} = \texttt{gn}(\mathcal{T}, \sigma')$$
$$\sigma''' \equiv x = [(\xi_1\ \xi_1\theta) \ldots (\xi_n\ \xi_n\theta)] \wedge \sigma''\theta\theta' \wedge \bigwedge_{\kappa' \in \texttt{tkn}_{\sigma'}(\mathcal{T})} \texttt{in}(\kappa, \kappa'\theta)$$

5.2 OZ/K properties

This section gathers elementary properties of the OZ/K operational semantics. We let $\to^*$ denote the reflexive and transitive closure of the reduction relation $\to$. We say that an execution structure $(\sigma, \mathcal{T})$ results from the execution of an OZ/K statement, if there exists a OZ/K statement S such that $(\sigma_0, \tau\langle S\ \langle\rangle\rangle) \to^* (\sigma, \mathcal{T})$, where $\sigma_0 \equiv \tau : \texttt{thread}(w) \wedge w \wedge \texttt{read}(w) \wedge \texttt{inth}(\top, \tau)$.

We say that a task $\mathcal{T}$ belongs to a kell κ if for all names τ of threads in $\mathcal{T}$, we either have $\texttt{inth}(\kappa, \tau)$ or $\texttt{inth}(\kappa', \tau)$, where κ' is a descendant kell of the kell κ. The following proposition establishes the separation property for OZ/K computation. It asserts that two distinct kells in an execution structure cannot hold references to the same unbound variable (either directly, or indirectly, through cells, procedures, etc).

Proposition 1 *Assume $(\sigma, \mathcal{T})$, with $\mathcal{T} \equiv \mathcal{T}_1\ \mathcal{T}_2\ \mathcal{T}'$, is an execution structure that result from the execution of an* OZ/K *statement, where $\mathcal{T}_1$ belongs to kell κ_1, $\mathcal{T}_2$ belongs to kell κ_2, and $\kappa_1 \neq \kappa_2$. If $\sigma \models x = \bot$, and $x \in \texttt{v}(\mathcal{T}_1, \sigma)$, then $x \notin \texttt{v}(\mathcal{T}_2, \sigma)$.*

Proof: See Appendix C. □

The following proposition asserts a form of *perfect firewall* property for OZ/K, namely, that there exists an execution structure where a task can be completely isolated from the rest of the other tasks in the execution structure. Let $(\mathcal{T}, \sigma)$ be an execution structure. We say that κ *appears at the top level* in $(\mathcal{T}, \sigma)$, if $\sigma = \sigma' \wedge \texttt{in}(\top, \kappa)$, for some σ'. We also say that κ is *not referenced in* $\mathcal{T}$ if there exists no variable x such that $x \in \texttt{v}(\mathcal{T}, \sigma)$ and $\sigma = \sigma' \wedge x = \kappa$, for some σ'.

Proposition 2 *Let $(\mathcal{T}\ \ \mathcal{T}_\kappa, \sigma)$ be an execution structure that results from the execution of a* OZ/K *statement, where κ appears at the top level, $\mathcal{T}_\kappa$ is the set of all threads that belong to κ, κ is not referenced in $\mathcal{T}$, there is no thread τ such that $\sigma \models \texttt{inth}(\kappa, \tau)$, and $\sigma = \sigma_0 \wedge \kappa : \texttt{kell}(\emptyset, w)$, for some σ_0, w. The reductions possible from $\langle \sigma, \mathcal{T}\ \ \mathcal{T}_\kappa \rangle$ can only be of one of the following two forms:*

$$\frac{\mathcal{T}\ \ \mathcal{T}_\kappa}{\sigma} \left\| \frac{\mathcal{T}'\ \ \mathcal{T}_\kappa}{\sigma'} \right. \quad or \quad \frac{\mathcal{T}\ \ \mathcal{T}_\kappa}{\sigma} \left\| \frac{\mathcal{T}\ \ \mathcal{T}'_\kappa}{\sigma'} \right.$$

where $\mathcal{T}'_\kappa$ is the set of threads that belong to κ in execution structure $(\mathcal{T}\ \ \mathcal{T}'_\kappa, \sigma')$, and σ' is such that there is no τ such that $\sigma' \models \texttt{inth}(\kappa, \tau)$, and $\sigma' = \sigma'_0 \wedge \kappa : \texttt{kell}(\emptyset, w)$, for some σ'_0.

Proof: See Appendix C. □

Informally, if we denote by $\kappa[\mathcal{T}]$ a task $\mathcal{T}$ whose threads belong to κ, the proposition asserts that a kell structure of the form $\kappa[\kappa_1[\mathcal{T}_1] \ldots \kappa_n[\mathcal{T}_n]]$ at the top level, where κ is not referenced outside of $\kappa[\ldots]$ (and thus cannot be packed), constitutes a perfect firewall for the tasks $\mathcal{T}_1, \ldots, \mathcal{T}_n$. This can be understood intuitively since there is no thread in kell κ (condition *there is no thread named τ such that* $\sigma \models \texttt{inth}(\kappa, \tau)$) that can act as a relay of communication between threads in $\mathcal{T}_1, \ldots, \mathcal{T}_n$ and the outer environment, and since there is no gate opened in κ for such communication (condition $\sigma = \sigma_0 \wedge \kappa : \texttt{kell}(\emptyset, w)$).

6 Discussion

6.1 Component granularity in OZ/K

As currently designed, OZ/K supports different forms of components. Kells, of course, constitutes primitive components. Notions of components which can be built in standard OZ, such as e.g. port objects, active objects, and modules, and their variants (such as e.g. port objects sharing one thread) are also available to OZ/K programmers. This variety of component forms allows programmers to build component-based programs at different granularities. For instance, having multiple port objects sharing the same thread can reduce their cost to that of a standard object, whereas a kell can be as small as a single thread or an active object[6]. However, these different components do not share the same properties, e.g. with respect to passivation and objective reconfiguration, or isolation. It might be beneficial to study how to unfiy further the different notions involved. For instance, the **kell** and **thread** constructs share many characteristics, and one could think of applying the passivation operation to a single thread. The current OZ/K design distinguishes threads and kells because of their different communication capabilities. However, this distinction could be lifted if one allowed other forms of communication between kells, and a different passivation semantics.

6.2 Encapsulation and sharing in OZ/K

The kell construct in OZ/K enforces a strong encapsulation, exemplified by the firewall property (Proposition 2 in Section 5.2). This strong encapsulation prevents communications between subkells or between a subkell and its parent kell's environment, to bypass the parent kell, which can thus act as a sandbox. Sandboxing can be done without any knowledge of the behavior of en encapsulated kell, as exemplified by the sandbox examples in Section 4, and thus provides a simple way to enforce given protocols for establishing communication with an environment outside a sandbox. The encapsulation realized by the kell construct allows in particular to build wrappers as in the Boxed-π calculus [102].

The strong encapsulation provided by the kell construct can become a hindrance, however, when building software architectures with component sharing [61]. For instance, A logger might be used to provide a logging service to different components in a software structure, whose locations, in the component hierarchy, can be arbitrary. A component such as the logger, can be understood as being *shared* among all the composite components that encapsulate its client components. In OZ/K, component structures with sharing can be approximated using gate opening. For instance, a logger configuration, with two components `C1` and `C2` that use the logging service, placed inside composite components `CA1` and `CB1`, and `CA2`, respectively, can be defined as follows (with `LG` the gate at which the logging service is made available by the logger `Logger`):

```
kell{CA1}
  kell{CB1}
    kell{C1} ... end
    {Open C1 LG}
    ...
  end
  {Open CB1 LG}
end

kell{CA2}
  kell{C2} ... end
  {Open C2 LG}
  ...
end

kell{Logger} ... end
{Open all LG}
```

[6] Note that threads in OZ are extremely lightweight, which authorizes in OZ the liberal use of port objects as units of modularity. The cost of a kell at execution is no higher than that of an OZ port object.

In the above program sketch, sharing the `Logger` component amounts to establishing direct communication channels with it, by opening the `LG` gate for communication at all the required levels of the component structure. In the case of a logging service, the communication between a logger client and the logger is typically unidirectional, with the client just requesting that some information be logged in a single message. If a service such as a database management service is shared, then one would expect to have interactions between a client and the database take the for of requests with responses. We can accommodate simply this kind of interactions using subordinate gates. For instance, with a component structure similar to the one above, we would set:

```
kell{CA1}
  kell{CB1}
    kell{C1} ... end
    {Open C1 LG#all}
    ...
  end
  {Open CB1 LG#all}
end

kell{CA2}
  kell{C2} ... end
  {Open C2 LG#all}
  ...
end

kell{Database} ... end
{Open all LG#all}
```

Instructions of the form `{Open C LG#all}` open the gate `LG` for direct communication but also all of it subordinate gates, thus allowing, for instance, a subordinate gate to be used as a continuation in a request / response interaction style. If necessary, one can protect the different gates from tampering by the component involved by building e.g. the equivalent of FRACTAL interfaces, as shown in Section 6.7, which encapsulate gates and a particular interaction protocol. Unfortunately, we do not know how to avoid decorating the whole component structure with `Open` instructions in order to model component structures with sharing. It seems there is a basic tension between the need for strict encapsulation, as required e.g. for writing wrappers for untrusted components, and the definition of "natural" component architectures with sharing. One could of course imagine adopting a different stance for OZ/K, which would consist in turning by default all kells into transparent ones (i.e. ones which would allow direct communication on all gates), and in adding a new operation allowing to retrieve and monitor the set of gates used for communication by kell. Creating transparent kells is easy in OZ/K. The following procedure creates such kells:

```
proc{NewKell P K}
  kell{K} {P} {Open all all} end
  {Open K all}
end
```

The body of the kell is input to procedure `NewKell` in the form of a nullary procedure, `P`. The first `Open` statement allows all the subkells of the transparent kell `K` to have the same communication rights as threads in `K`. The second `Open` statement allows the content of `K` to have the same communication rights as threads in the current kell.

However we do not have the possibility to dynamically monitor the gates of a kell and preventing communication using only transparent kells. The problem with the alternate approach would thus be to devise an appropriate primitive for this gate monitoring and selective gate communication prevention.

A different approach to the issues of encapsulation and sharing would be to rely on a static type system, to ensure proper encapsulation in a context where sharing is the norm, as with object-based languages. The solutions devised for object-oriented languages to overcome the aliasing issues, would be relevant here. For instance, Clarke et al. [36, 37] introduce *ownership types* which attribute to each object *obj* an *owner* that controls the references to *obj*. Similar types are used in ArchJava [3] to ensure a form of component encapsulation called *communication* *integrity*. To what extent these ideas, together with the techniques for typing and the dynamic binding of modules developed for Alice and Acute can be exploited in our setting, remains for further study.

6.3 On network independence

The principle of network independence actually leads us to avoid introducing in the language any abstraction for remote communication or remote execution. This may seem paradoxical in a language intended for distributed programming, but this decision actually opens the way for the introduction of many different forms of abstractions for distributed programming. Let us explain this in more detail[7]. The only abstraction for distribution which we introduce is OZ/K, is that of a kell, a form of locality as can be found in distributed process calculi such as Dπ or Mobile Ambients. The presence of kells means that we can partition OZ/K computations into separate places. Now, these places can be realized either as data structures and programs executing on a given machine, or as whole machines, together with their software environment – including e.g. a virtual machine for running OZ/K programs. In other words we can view an OZ/K statement either as an *executable program*, to be run e.g. on an OZ/K virtual machine, or as a *model*, that specifies the behavior of some system. A distributed environment with two machines `M1` and `M2`, an interconnection network `N`, can be modelled by an OZ/K statement of the form:

```
kell{N} {NetBehavior} end
kell{M1} {MachineBehavior M1 Add1 Program1} end
kell{M2} {MachineBehavior M2 Add2 Program2} end
thread {TopLevelBehavior} end
```

where `Program1` and `Program2` correspond to OZ/K statements to execute on `M1` and `M2`, respectively, and where `Add1` and `Add2` correspond to addresses of `M1` and `M2` on the network, and the overall behavior of the network is modelled by the two statements `TopLevelBehavior` and `NetBehavior` (the statement `TopLevelBehavior` typically merely provides for the opening and closing of gates for a direct communication between the network kell `N` and the machine kells `M1` and `M2`). The procedure `MachineBehavior` can also be seen as having a structure of the form

```
proc{MachineBehavior M Add Program}
VM in
 thread {VirtualMachinery VM} end
 kell{VM} {Program M Add} end
end
```

which means it consists in running a `Program` – here described as a procedure that takes as argument the name of the local machine (e.g. a gate representing its IP domain name), and its address (e.g. its IP address) – together with some additional virtual machinery.

The important point to notice is this: because we have separated the computation space into different kells, these can be realized in different ways. For instance, the kell `VM` above can be realized as an OZ/K virtual machine, able to execute OZ/K statements, such as `{Program M Add}`, whereas kells `M1`, `M2`, `N` will model the actual machines and network, that support the execution of the two copies of the `VM` virtual machine in our distributed environment. The statements `NetBehavior`, and `{MachineBehavior M1 Add1 Program1}`, should then be considered not as executable OZ/K programs, but as models of the behavior of `N`, and `M1`, respectively. From the point of view of an OZ/K programmer, programming in a distributed setting means accessing, using the communication constructs in OZ/K (sending and receiving on gates), the services available in the realization of `MachineBehavior` and `NetBehiavor`, just as if they were ordinary OZ/K programs. In other terms, to program in a distributed environment, we just require that its basic services (typically, communication services) be made available as OZ/K abstractions. From a semantical point of view, the boundary between actual programs and models of the supporting environment, is clearly marked, thanks to the separation of computation in different kells.

In a sense, this approach is comparable to a distributed computing extension to an object-based language that relies on special objects that wrap distributed services available from the supporting environment (communication

[7] A similar case has been made for a network independent abstract machine for the Kell calculus in [21].

libraries, machines and networks). Contrary to classical distributed extensions to object-oriented languages, such as RMI for Java or Network Objects for Modula 3 [22], however, we do not try to extend the semantics of the language local communication primitive — method invocation in the case of Java and Modula 3 — to cover the case of remote execution. Instead, all communications in OZ/K, including those with distributed services, retain their semantics, and remain strictly local. Also, note an important distinction: with our approach, distributed execution can be described within the semantics of the language, because of the introduction of localities; this is not the case with the network objects approach, where distributed execution is not captured by the language semantics. Having a primitive notion of locality in the language semantics allows to specify different forms of separation, much like having a notion of thread in the language semantics allows to formalize concurrent execution and synchronization, compared to an approach where concurrency is introduced as an external library to a sequential language, and thus is not part of the language semantics.

Overall, our approach to distribution is similar to that of Acute, which adheres to the belief that "a general-purpose distributed programming language should not have a built-in commitment to any particular means of interaction" [101, 99]. However, in contrast to Acute, we believe it is important to add a basic notion of separation in the language semantics, so as to be able to provide a model of a distributed environment in terms of the language semantics. In Acute, distribution is only manifest through the sending and receiving of marshalled modules.

Supporting different communication capabilities in a uniform fashion can be obtained via the `export/bind` design pattern, which has been used to good effect in different operating system and middleware developments [5, 45, 47, 68]. This architectural pattern can be summarized as follows:

- Communication between different sites first requires that these sites share a common *naming context*, within which communicating entities at each site can be unambiguously designated (named). The `export` operation allow an entity in a participating site to receive an unambiguous name within the common context.
- Communication between different sites then requires the existence of *binding factories*. A binding factory supports the `bind` operation, which establishes a communication path (called *binding*) between a set of named entities.

A simple example of the `export/bind` pattern is provided by the establishment of a remote operation channel between a client and a server located on two different machines. To enable clients to connect to the server requires first that the server's local name be exported with the naming context that is provided by the chosen remote operation protocol. The `export` operation returns a name that unambiguously designates the server in the context of the chosen remote operation protocol, and that can be communicated to potential clients. Once a client has obtained the exported name of the server, it can invoke the appropriate binding factory to establish a binding with the server via the `bind` operation. Once the binding is established, the client can invoke operations on the server. This behavior can be readily captured as follows. We model as above each site and the network by a distinct kell. A server component is modelled as a kell communicating on a dedicated half-gate (with the receive capability private to the server kell). Each request to the server takes the form of a message, i.e. a tuple with two fields: the first field contains the request arguments, the second field contains a subgate of the server gate, which can be used as a continuation to send back to the client the response to the request. Each site runs a `BindingFactory` component (typically as part of the `VirtualMachinery` behavior in the previous code snippet), which provides both an `Export` operation and a `Bind` operation. The `Export` operation takes as input the interface of a server (its dedicated half-gate) and returns a chunk that can be used as input to the `Bind` operation to create a communication channel between sites (e.g. similar in principle to a `Relay` process as defined in Section 4.2).

6.4 Gate semantics and implementation

Two important design choices have been made concerning gates and gate communication. First, gates are not located, i.e. a gate does not belong to a particular kell. This is possible because of the network independence assumption, which makes all gates local (to the top level kell in an OZ/K virtual machine). An advantage of this design choice is that gates can be used indifferently by all kells, subject to the opening and closing specifications that govern communication on gates. A disadvantage is that the usual notion of component interface, which implies some sort of ownership of interfaces by components, must be encoded, as demonstrated with the encoding of half-gates. Second, communication via gates is by atomic rendezvous. Again, the network independence assumption means this is reasonable since all gates are assumed to be local to an OZ/K virtual machine. A main advantage of this design choice is that there is no state associated with gates. This in turn simplifies considerably the semantics of packing and unpacking. In particular, both operations can take place at any point during an OZ/K computation. If a different communication semantics had been chosen, e.g. communication by means of bounded or unbounded buffers, then packing and unpacking would have had to deal with the state of communication buffers, as is the case e.g. in the implementation of the Kell calculus abstract machine reported in [21]. Another advantage is that it is possible to encapsulate different communication semantics in connectors, i.e. kells that act as communication channels between other kells, as illustrated in Section 4. In particular, it is possible to define connectors with explicit flow control, in contrast e.g. to ports or mailboxes in OZ, E, Erlang, or Sing#.

A disadvantage of this choice of communication semantics is that care should be exercised to obtain an efficient implementation. However one can note that communication by atomic rendezvous-vous can be implemented efficiently, as shown e.g. by its use as the interprocess communication primitive in the Minix 3 operating system [59]. Also, it is possible to devise efficient specific implementations for rendezvous-vous with certain forms of kells which can be assumed to always have a thread waiting for communication, e.g. when dealing with kells providing an asynchronous communication service based on buffers, or when dealing with kells that support server interfaces as in the FRACTAL model.

Overall, the current design choices represent a reasonable compromise, for they do allow different communication semantics to be defined as different forms of "connector" kells, while allowing a clear semantics for packing and unpacking. However, there are several questions pertaining to the definition of communication primitives for kells that remain open. For instance, programing control loops for self-manageable systems, as advocated by [110], could be facilitated by introducing *regulative superposition* [64], as supported in the IP formalism [49]. This in turn could require adopting some form of multicast guarded communication, with kell containment understood as superposition composition.

6.5 Dynamic reconfiguration in OZ/K

OZ/K provides basic support for dynamic reconfiguration through kells and packing. However, there are at least two issues that still need to be addressed with respect to dynamic reconfiguration: the granularity of possible reconfigurations, and automated support for state transfer.

The unit of dynamic reconfiguration in OZ/K, the kell, is coarse grained, for it corresponds roughly to that of an active object. On the one hand, this level of granularity provides a good isolation between components: components can fail independently, and failure handling can take place in separate monitoring components, as illustrated in Section 4. On the other hand, finer-grained component-based designs, illustrated e.g. in [111], do not get built-in support for on-line update and replacement. Supporting dynamic reconfiguration at this level, would require the ability to update executing code at the level of procedure frames, possibly exploiting ideas for dynamic software update as proposed e.g. in [60], and for updateability analysis, as presented in [106].

The second issue with dynamic reconfiguration has to do with the automation of state transfer to ensure safe component updates. The semantic issues associated with dynamic reconfiguration have been explored in some works in the past two decades [23, 55]. A key enabler is the capture of appropriate state information on the component to replace. In OZ/K, this information is made available in the form of a packed value. Providing support for automated state transfer would require having the ability to introspect the contents of a packed value, at a sufficient level of granularity. A fine-grained access to the contents of a packed value can easily be provided by a standardized record representation for tasks and stores. However, support for extracting higher-level state information from packed values record representation would still be required, typically exploiting meta-programming ideas and in particular template meta-programming, e.g. as described for Haskell in [104].

6.6 Failure and event handling

The failure handling facilities provided in OZ/K still remain fairly crude, and depend on predefined status information for threads and kells. Additional support is required in the OZ/K computation model for programming different forms of failure detectors, dealing e.g. with omission failures. Two alternatives seem to be worth exploring. The first one would imply introducing an explicit notion of time in OZ/K, relying on the large body of work dealing with timed transition systems, and the timed-π-calculus in particular (for instance, [17]). The second one would be to introduce reactive programming constructs as in the ULM programming model [25], which combines synchronous and functional programming. The second alternative is particularly appealing for it provides an elegant way to deal with multiple forms of timers and interrupts, and formalizes event-driven execution with no need to introduce an explicit notion of time. A crucial element in reactive programming is the ability to react to the *absence* of events. It would be interesting to see how this relates to the ability to test for the determinacy of logical variables, and whether streams in OZ and OZ/K could be used to model flows of reactive events.

6.7 On component-based programming in OZ/K

The kell concept provides a basis for component-based programming, with strictly enforced component encapsulation and isolation. As an illustration of this fact, we define in this section different constructs to support programming following the FRACTAL component model. The FRACTAL model is a language-independent reflective component model, which targets the construction of highly configurable systems. It has been used for the construction of several configurable systems, including operating system kernels [47], message-oriented middleware [68], for the instrumentation and automatic deployment of application server clusters [24], for building auto-adaptive systems [44], for building systems with integrated quality of service (QoS) management [109], or building adaptive multimedia applications [66]. The main elements of the model can be summarized as follows:

- Components are run-time entities, that are encapsulated, and that exhibit interfaces (access points) for communication with their environment. Interfaces can be of two kinds: server interfaces, which can receive operation invocations (either simple requests with no response, or requests with responses); client interfaces, which can emit operation invocations.
- Components can provide different meta-level interfaces for accessing their internal structure, and for controlling their behavior. In the general case, the internal structure of a component can be understood as comprising *membrane* and *contents*. The contents of a component consists in a set of other components. The membrane of a component embodies the specific behavior of the component, including meta-level behavior, e.g. for supporting introspection and intercession.

The FRACTAL model does not prescribe a given set of meta-level interfaces for components, however several useful such interfaces have been identified [28], including:

- *Component.* This interface provides access to the different interfaces of a component.
- *Content Controller.* This interface provides access to the contents of a component, and the ability to add or remove subcomponents.
- *Binding Controller.* This interface provides the ability to bind client interfaces of the component with server interfaces of other components in its environment.
- *Attribute Controller.* This interface provides the ability to access the attributes of a component, a set of named pieces of information associated with a component.
- *Lifecycle Controller.* This interface provides the ability to access and modify some macro-states of a component, such as `active`, `waiting`, `stopped`, etc.

We present in this section an interpretation in Oz/K of the FRACTAL specifications. This interpretation is analogous to the interpretation of objects in the Oz kernel computation model. Briefly, we interpret a FRACTAL component as an Oz/K kell, whose interfaces are mapped onto gates. The sub-components of a FRACTAL component are modelled as sub-kells, whereas the membrane of a component is modelled as a record (of shared attributes) and processes, which can comprise operation handlers, and meta-level interface controllers.

Component templates. We first define *component templates*, i.e., by analogy with object-oriented programming, classes for components. This allows to enforce certain programming conventions when building components. A component template `Temp` is a record that contains:

- A set of server interface templates, accessible via the feature `svIfs`. A server interface template contains an interface name, and a record of operations. The features of the operation record are the names of the operation. The fields of the operation record are operations. An operation is a procedure with three arguments: a message `M`, which is always a record, whose label denotes the name of the operation; a parameter `State` the represents the current state of the component owning the interface; and a reference `S` to the interface itself.
- A set of client interface templates, accessible via the feature `slIfs`. A client interface template contains an interface name, and a set of operation names to be used as labels of messages
- A set of names, that correspond to the names of the attributes of the component, accessible via the feature `atts`. Each attribute is a stateful cell that can be accessed by the attribute name, which is either an atom or a name. The record of all attributes of a component constitute its state.
- A template for a component controller, acessible via the feature `comp`. The component controller is the most basic form of controller for a component. It provides access to the other interfaces of the component.
- A set of controller templates, accessible via the feature `ctrls`. A controller provides a meta-level interface to implement. A controller template takes the form of a procedure. A call to the procedure instantiates a new controller for the current component.
- A set of component templates, accessible via the feature `subTemps`, that constitutes the templates required to create the subcomponents of the enclosing composite.
- An initialization procedure, accessible via the fature `init`. This procedure is responsible for initializing the internal structure of the composite component, including creating subcomponents, binding them together, binding subcomponent interfaces to interfaces of the enclosing composite.

Components. Components are created from component templates, via the following procedure (note that we make use of standard list and record operations as provided in the MOZART environment):

```
proc{NewComponent Template Gate}
K Gate = {NewGate $} in
  kell{K}
   State IST ICT CT CCT Component Meta = c(ist:IST ict:ICT ct:CT cct:CCT) in
    {List.forAll [IST ICT CT CCT] proc{$ T}{{NewDictionaryObject T}end}
    {MakeRecord s Template.atts State}
    {Record.forAll State proc{$ A}{NewCell _ A}end}
    {Template.comp K Component State Meta Gate}
    {Record.forAll Template.ctrls proc{$ I}{I K State Meta}end}
    {Record.forAll Template.svIfs proc{$ I}{Interface.newS I Component State IST _}end}
    {Record.forAll Template.clIfs proc{$ I}{Interface.newC I Component ICT _}end}
    {Template.init K State c(IST ICT CT)}
  end
end
```

Component creation proceeds as follows. First, dictionaries are created[8] that will hold meta data associated with the component: the table of (external) server interfaces of the component, called `IST`; the table of (external) client interfaces of the component, called `ICT`; the table of component controller interfaces, called `CCT`; the table of subcomponents, called `CT`. Then a record is created that will constitute the internal `State` of the component, in the form of a set of attributes. The component controller of the component is then created (and put in the componaent controller table `CCT` as a side effect). Other controllers are created, followed by server and client interfaces, and finally the initialization procedure is called (which typically can create subcomponents and configure them).

Interfaces. We define here the module `Interface` that allows to instantiate server and client interfaces from server and client interface templates, respectively. For server interfaces, it essentially creates a new gate which can be opened for communication with the owner component (the owner component is the current component executing the `NewServerIf` operation). For client interfaces, it essentially creates a cell that can be updated with a server interface, which corresponds to the establishment of a binding between the client interface and a server interface.

```
ServerIf TagS={NewName $} NewServerIf IsServerIf
ClientIf TagC={NewName $} NewClientIf IsClientIf
in
proc{IsServerIf I B} {HasFeature TagS I B} end
proc{NewServerIf Template Component State IfT Server}
  G = {NewGate $}
  Server = {NewChunk r(TagS:Request open:OpenS close:CloseS owner:Component) $}
  OpenS = proc{$ K} {Open K G#all} end
  CloseS = proc{$ K} {Close K G#all} end

  Request = proc{$ M}
    L = {Label M $} in
    if {HasFeature Template.ops L $}
    then case M of
              L(R unit) then {Send G M}
         []   L(R X) then Y RG in {NewGate G#RG}{Send G L(R RG)}{Receive RG Y} X = Y
         else raise invalidRequest(M) end
         end
    else raise invalidOperation(L) end
    end
  end

  Handle  = proc{$}
  M  L = {Label M $} in
    {Receive G M}
    case M of
       L(R unit) then
          try {Template.ops.L M State Server}
          catch _  then skip end
    [] L(R RG} then Y in
          try {Template.ops.L L(R Y) State Server}{Send RG Y}
          catch E  then {Send RG error(E)} end
    else skip end
    {Handle}
  end

  in
  thread {Handle} end
  {IfT put(Template.ifName Server)}
end
```

[8] Note that, to simplify the code, we postulate the existence of an operation `NewDictionaryObject` which returns not a standard MOZART dictionary but a dictionary *object*. Just like objects in MOZART, a dictionary object is a procedure which takes an operation request as argument. An operation request is a record whose label corresponds to the name of the operation.

```
\% Client interfaces

proc{IsClientIf I B} {HasFeature TagC I $} end

proc{NewClientIf Template Component IfT Client}
  Ch = {Newcell unit $}
  Client = {NewChunk r(TagC:Ch owner:Component bind:Bind unbind:Unbind invoke:Invoke) $}

  Bind = proc{$ S}
    if {IsServerIf S $} then Ch := S
    else raise notServerIf(S) end
    end
  end

  Unbind = proc{$} Ch := unit end

  Invoke = proc{$ M}
   L = {Label M $}
   S = @Ch in
    if {Member L Template.opLabels $}
    then case M of L(R X)
         then {S.TagS M}
         else raise invalidRequest(M) end
         end
    else raise invalidOperation(L) end
    end
  end

  in
  {IfT put(Template.ifName Client)}

end

Interface = ifMod(isServer:IsServerIf newS:NewServerIf isClientIf:IsClientIf newC:NewClientIf)
```

The procedure `NewServerIf` creates a new server interface (a chunk, with features `TagS`, `open`, `Close`, and `owner`), creates a thread dedicated to the handling (via procedure `Handle`) of operation requests on the newly created server interface, and registers the newly created server interface in the interface table `ifT` which is passed as a parameter to the procedure. An interface table is a dictionary, whose keys are the names of interfaces (represented as OZ/K atoms or names) it holds. The `Template` parameter of the procedure is a server interface template, the `Component` parameter corresponds to the owner of the newly created interface, and the `State` parameter corresponds to the record of attributes of the owner component. The `Handle` behavior is simple: it awaits a message on the gate `G` which is attached to the server interface. A message is essentially a record whose label is the name of the requested operation, and which contains two fields: the first one is a record of arguments of the operation, the second one is a continuation for the operation. If the continuation is **`unit`**, this indicates that the operation is a simple request with no response. Otherwise, the continuation is a gate on which the response to the operation request is returned. Note that `Handle` serves requests sequentially. If another behavior is required, such as e.g. handling requests concurrently with a pool of threads and a scheduler, then all that is required is to change the `Handle` procedure to implement the required behavior.

A server interface can be passed freely in communication to different components. The `OpenS` operation is present to allow communication on a server interface to cross component boundaries, i.e. for both operation requests and operation responses (via the statement `{Open K G#all}` in the body of procedure `OpenS`) to freely cross component boundaries. Notice that the gate which a server interface encapsulates cannot be used directly. This enforces communication on a server interface to obey the request/response protocol associated with its operation. Thus, it is not possible for a thrid-party to listen on the gate of a server interface and to intercept requests and responses coming sent on this gate. This construction is similar to the half-gate construct presented in section 4.

A client interface can only be known outside of its owner component via its interface name (this is because a client interface is chunk that holds a cell – a non-strict value) [9]. However it is possible to bind a client interface to a server interface from outside its owner component by using the binding controller (server) interface below, which will use to that effect the `Bind` operation provided by the client interface. The `Bind` operation merely updates the client interface internal cell with the server interface passed as argument. The `Invoke` operation on a client interface can be used to communicate via a client interface (note that only threads within its owner component can make use of a client interface). It takes a message as an argument, which is a record whose label corresponds to the name of an operation supported by the interface, and which contains two fields. The first one is the content of the message. The second one is the continuation of the message. If it is bound to **`unit`** upon invocation of the `Invoke` operation, this means the operation is a simple request with no expected response. Otherwise, it will be bound to the response to the operation request when the latter completes.

Component controller. We define here the component controller from the FRACTAL specification[10]. It allows to discover the different interfaces of a component. Note that, from an interface, it is possible to recover the `Component` controller interface, by accessing the interface's `owner` feature. The component controller provides operations with which it is possible to retrieve all the server interfaces (`GetSIfs`), all the client interfaces (`GetCIfs`), and all the controller interfaces (`GetCtrls`) of a component. In addition, operation (`GetHand`) retrieves the name of the kell that constitutes the component. The gate `Gate` provides a means to retrieve the component interface itself[11].

```
proc{ComponentCtrl K Component State Meta Gate}
  Temp = temp(ifname:component
              ops:m(getSIfs:GetSIfs getCIfs:GetCIfs getCtrls:GetCtrls getHand:GetHand))
  GetSIfs = proc{$ M}
     case M of getSIfs(_ R) then {Meta.ist toRecord(ist R)} else skip end
  end

  GetCIfs = proc{$ M}
     case M of getCIfs(_ R) then {Meta.ict toRecord(ict R)} else skip end
  end

  GetCtrls = proc{$ M}
     case M of getCtrls(_ R) then {Meta.cct toRecord(cct R)} else skip end
  end

  GetHand = proc{$ M}
     case M of getHand(_ R) then R = K else skip end
  end

in
  {Interface.newS Temp Component State Meta.cct Component}
  thread P = proc{$}{Send Gate Component}{P} end in {P}  end
end
```

Note that the creation of the component controller uses the `Interface` module and its operation for creating server interfaces, which puts in place, as a side effect, a `Handle` thread for dealing with operation requests. The component controller behavior is thus determined by the `Handle` procedure defined in the `Interface` module, and the operations associated with the `Component` server interface. This scheme is used for the other controllers defined below.

[9] Ensuring that an interface name is unambiguous within the context of a component is the responsibility of the procedures that update the component interface tables, i.e. procedures `NewS` and `NewC` in module `Interface`. However, for the sake of simplicity, we have not added these checks in the code of these two procedures.

[10] For the sake of simplicity, there are some slight differences between the controller operations we define in this section, and those in the FRACTAL specification. However the essential functionality of the FRACTAL default controllers is preserved.

[11] This gate can be seen also as a component identifier. To turn it into a true component identifier would require to wrap it in a half-gate, as illustrated in Section 4, so that it is only possible to receive on this gate. For the sake of simplicity, this is not shown here.

Attribute controller. We define here the attribute controller from the FRACTAL specification. This controller allows to access the attributes of a component, through getter and setter operations.

```
proc{AttributeCtrl K State Meta}
  Temp = temp(ifname:attribute  ops:m(get:Get set:Set))
  Get = proc{$ M}
     case M of get(A R) then R = @(State.A)
     else skip
     end
  end

  Set = proc{$ M}
     case M of set((A V) unit) then (State.A) := V
     else skip
     end
  end

in
  {Interface.newS Temp Component State Meta.cct _}
end
```

Binding controller. We define here the binding controller from the FRACTAL specification. This controller allows to bind and unbind client interfaces of a component to server interfaces of another component.

```
proc{BindingCtrl K State Meta}
  Temp = temp(ifname:binding  ops:m(bind:Bind unbind:Unbind))
  Bind = proc{$ M}
     case M of bind((L S) unit)
     then C = {Meta.ict get(L $)} in {C.bind S}
     else skip
     end
  end

  Unbind = proc{$ M}
     case M of unbind(L unit)
     then C = {Meta.ict get(L $)} in {C.unbind}
     else skip
     end
  end

in
  {Interface.newS Temp Component State Meta.cct _}
end
```

The `Bind` operation takes as arguments the name of the client interface to bind (as registered in the component's client interface table), and a server interface. The name of a client interface can be an atom (as is the case here with controller interfaces), or an OZ/K name.

Content controller. We define here the content controller from the FRACTAL specification[12]. This controller allows to add and remove subcomponents, to and from a component.

```
proc{ContentCtrl K State Meta}
  Temp = temp(ifname:content  ops:m(add:Add remove:Remove))
  Add = proc{$ M}
     case M of add((V G) unit)
     then KK W Comp in
          kell{KK} {V.mark top(KK) W}{V.Mark gate(G:G)}{Unpack W _} end
```

[12]In the FRACTAL specification, the content controller supports an operation that returns the internal interfaces of a component, i.e. interfaces that are provided by interceptors for subcomponent interfaces. We have not defined this operation to keep things simple.

```
          {Receive G Comp}
          {Meta.ct put(G Comp)}
     else skip
     end
  end

  Remove = proc{$ M}
     case M of remove((Comp G) R)
     then K = {Comp getHand(_ $)} in
          if {Meta.ct member(G Comp) $}
          then {Meta.ct remove(G Comp)} {Pack K R}
          else R = notSubComponent(Comp G)
          end
     else skip
     end
  end

in
  {Interface.newS Temp Component State Meta.cct _}
end
```

The Add operation takes as argument a packed value V that contains the component to add, and a gate G, which corresponds to the gate on which to retrieve the component controller interface of the added component. The Add operation just unpacks the packed component in a new kell. The gate which G which identifies the component is preserved, and the component controller interface of the new sub-component is added to the sub-component table of the current component. The Remove operation takes as argument a component controller interface and a gate G that identify the sub-component of the current component to remove. The operation returns a packed value which contains the remove sub-component, after having removed the appropriate entry from the sub-component table of the current component.

Lifecycle controller. We define here the life cycle controller from the FRACTAL specification. This controller allows to start and stop the execution of a component (apart from its controllers).

```
proc{LifecycleCtrl K State Meta}
  Temp = temp(ifname:lifecycle  ops:m(start:Start stop:Stop getState:GetState))
  Status = {NewCell stopped $}

  Stop = proc{$ M}
   T = Meta.ct in
     case M of stop(_ unit)
     then O N in {Exchange Status O N}
          if O == started
          then
            R = {T toRecord($)}
            L = {Record.arity R $}
            P = proc{$ G}
              V = {T get(G $)} K = {V getHand(_ $)}
              W = {Pack K $} Z = {W.mark gate(G:G) $} in
              {T remove(G)} {T put(G Z)}
            end
            in
            {List.forAll L P} N = stopped
          else skip
          end
     else skip
     end
  end

  Start = proc{$ M}
   T = Meta.ct in
```

```
     case M of start(_ unit)
     then O N in {Exchange Status O N}
          if O == stopped
          then
            R = {T toRecord($)}
            L = {Record.arity R $}
            P = proc{$ G}
              V = {T get(G $)} K W Comp in
              kell{K} {V.mark top(K)}{Unpack V _} end
              {Receive G Comp}
              {T remove(G)} {T put(G Comp)}
            end
            in
            {List.forAll L P} N = started
          else skip
          end
     else skip
     end
  end

  GetState = proc{$ M}
    case M of getState(_ R) then R = @Status
    else skip
    end
  end


in
  {Interface.newS Temp Component State Meta.cct _}
end
```

7 Related work

OZ/K is related to several bodies of work, which we can classify in, roughly, the following categories: component-based programming models, architecture description languages, programming languages, process calculi.

Programming languages. The reference language for open programming is the Java language [12, 54], with its comprehensive programming environment. A number of open programming facilities are provided by Java and its associated environment (e.g. through class loading, remoting and security mechanisms), but they still exhibit important limitations, e.g. with respect to modularization and componentization (no native notion of component, except through the notions of Java Beans or EJBs, but without hierarchical components and control over component interconnections; limited form of modules through OSGI bundles), explicit marshalling and pickling (no generic pickling mechanism, serialization provides only limited marshalling – e.g. code cannot be serialized), dynamic linking, and isolation (dynamic linking and sandboxing available through class loaders and security managers, with complex APIs and no formal semantics). Overall, support for open programming in the Java environment appears complex, with crucial aspects dealt with in environment libraries and associated APIs, and with no formal semantics.

A few programming languages are built around a notion of locality, notably JoCaml [48], Nomadic Pict [114], O'Klaim [18], ULM [25]. None of these languages provide the ability to build sandboxes with strong isolation properties as OZ/K provides. Except for JoCaml (which supports hierarchical localities and strong mobility), localities in these languages essentially represent execution sites.

A number of works have considered recently open programming issues, dealing in particular with software configuration, modules and dynamic linking, such as e.g. [8, 30]. These works focus on basic formalisms and calculi dealing with specific issues. There have been comparatively less work on programming language designs taking open programming features into account. Recent ones include AchJava [3, 4], Assemblage [72], ComponentJ [97, 98], E [80], Jiazzi [78], Piccola [2, 75], Scala [83], Classages [71], Sing# [46], OZ [111], Alice [91, 90], Acute [99, 100, 101] and O-Klaim [19].

ArchJava, ComponentJ, Jiazzi, Assemblage and Classages focus on the notions of components and component composition. ArchJava, Assemblage, and Classages come closest to the notion of component as embodied in the kell notion in OZ/K. However, even though ArchJava and Classages components are units of encapsulation, and provide what ArchJava calls *communication integrity*, components in ArchJava and Classages are not units of fault isolation (multiple threads may traverse a given component at any point in time). Also, component configurations in ArchJava and Classages can evolve at run time, through the creation of new components and new connectors, but these evolutions are limited by what the behavior programmed in component classes and classages. ArchJava and Classages do not provide for the kind of unplanned reconfigurations and component mobility that can take place in OZ/K through the use of the `Pack` and `Unpack` primitives, and they do not support passivation, failure detection, and isolation, as OZ/K does. Assemblage has recently been extended to include explicit deployment [70], but still the language does not provide support for passivation and isolation.

Piccola is a scripting language developed on top of a formal kernel, the asynchronous π-calculus with extensible records (called *forms*). Piccola is intended as a composition language, which derives its expressive power from the combination of the π-calculus lexical scoping and name passing, together with extensible records, which allow e.g. the encoding of generic wrappers and higher-order composition schemas. Piccola's notion of component is that of a π-calculus process, and remains limited with respect to the handling of distribution, isolation, explicit marshalling and passivation, compared to OZ/K.

Scala combines object-oriented and functional programming in a statically typed programming language, which supports class mixins and views, with a very expressive type system. The notion of component and component composition in Scala is closer to the notion of module than to the run-time unit of isolation and reconfiguration that OZ/K provides with its notion of kell. In addition, Scala does not provide support for passivation, explicit marshalling and pickling, as available in OZ/K.

Alice can be understood as an extension of Standard ML [81] that offers higher-order modules, packages (essentially, an extension of the notion of dynamics [1], which combines a higher-module with its dynamic signature), pickles (marshalled forms of packages), components, and concurrency with futures and laziness. The Alice notion of component (or dynamic module) can be understood, following [90] as a function, taking packages as arguments (imports), and that evaluates to a package (containing the export module). The Alice notions of packages, pickles and components, formalize, in a strongly typed setting, similar notions of notions of functors and pickles that appear in OZ. Still, compared to OZ/K, Alice does not provide support for passivation, and the notion of sandbox in Alice, available through a notion of *component manager* that is part of the Alice library environment, is not accessible to programmers.

Acute is also a language in the ML family, with extensive support for open programming in a strongly typed setting, including explicit marshalling, dynamic linking, dynamic modules, support for versioning constraints, support for concurrency through threads, and even a form of passivation through the ability to thunkify running threads. Acute also introduces the notion of *mark* to control the extent of dynamic linking in module. The notion of mark in Acute is related to the `mark` operation on packed values in OZ/K, that can be exploited to obtain similar effects (e.g. shipping only the relevant portion of a module code, as illustrated in Section 4). Compared to OZ/K, Acute does not support sandboxing and isolation, and it supports open programming through a relative complex set of mechanisms that are subsumed in OZ/K by a smaller set of constructs (namely via *kells*, *gates*, and *packing*).

O-Klaim provides a Java-based, object-oriented programming language built on a formal kernel, the Klaim process calculus [82], that provides generative communication à la Linda [51, 52]. O-Klaim supports classes and a form of mixins with first-class status, that provide support for mobile code in a strongly-typed setting. In contrast to OZ/K, the notion of dynamic module provided by O-Klaim (also in comparison with Alice and Acute), through its *mobile mixins* appears limited (for instance, higher-order modules are not supported). In addition, O-Klaim does not provide support for sandboxing, isolation, and passivation.

The E programming language [80], whose aim is to be a secure language, combines object-oriented programming, capability-based access control, and concurrency control. Concurrency control in E is based on the notion of *vat*, which corresponds roughly to a thread communicating by its environment (other vats) via asynchronous remote method invocation with futures. E provides extensive support for capabilities, but fails to provide the sandboxing and passivation functionality that OZ/K supports.

The Sing# language [46], developed as part of the Microsoft Singularity operating system, that extends the C# programming language with isolated processes and asynchronous message passing communication. Processes in Sing# are isolated by virtue of being units of fault isolation, and of their code being unmodifiable, at run-time. However, Sing# does not provide the sandboxing and capabilities of OZ/K, and relies on standard C# notions and associated .Net capabilities for handling modules and code deployment.

OZ/K obviously builds upon OZ. The benefits brought by OZ/K have already been identified in the introduction. Our work on OZ/K is also related to recent proposals for extending OZ. A first attempt at exploiting a locality concept inspired by the Kell calculus was made in [63]. In this paper, localities (named "membranes") are finer grained than kells in OZ/K, but they are used only for communication control (confinement), and do not constitute units of failure isolation, or of passivation. The kell construct in OZ/K seems in line with the proposed design guidelines for a *secure* OZ, presented in [105].

Like OZ, OZ/K is an essentially untyped language. This is in contrast to most of the languages cited above (Acute, Alice, ArchJava, Classages, ComponentJ, O-Klaim, Scala), which are statically typed languages (with forms of *dynamics* for some, such as Acute and Alice). Static type checking has well-known advantages, and languages such as Acute and Alice provide stronger safety guarantees during execution than OZ/K does. However an untyped setting provides more flexibility when trying to combine different language features, as we are doing in OZ/K. It also allows for a simpler formal operational semantics (compare e.g. the operational semantics of OZ/K, and that of Acute). Devising a strongly-typed variant of OZ/Kis an item for further study.

Component-based programming models. Several component models have appeared during the last decade, in industry standards and specifications, such as Sun's Java Beans, and Enterprise Java Beans (EJBs) [107], Microsoft COM and .Net [73], the OMG CORBA Component Model (CCM) [84], the OSGI Bundle model [86], IBM and BEA's Service Component Architecture (SCA) [62], the Grid Forum Commmon Component Architecture (CCA) [11]. An overview and discussion of several of these models can be found in [108]. Some of these models (e.g. CCA, SCA) merely cater for interface specifications and managing inter component connection. Other, such as Java EJBs and .Net provide comprehensive programming environments. However, even the most complete ones fail to support all the open programming capabilities presented in this paper (i.e. support for components, dynamic modules, dynamic linking and binding, isolation, fault handling, and passivation) in an integrated fashion. Their programming support typically takes the form of complex and loosely integrated APIs, with no formal semantics, and the use of several different languages and formalisms to deal with open programming issues such as e.g. distributed deployment and configuration, and limited native support for dynamic reconfiguration. More experimental component models such as OpenCOM [38, 41] and Fractal [27] provide a stronger support for dynamic reconfiguration, but, as with other programming language independent models, such as CCM, their implementations are typically limited by the host programming environment. This is apparent e.g. with Java implementations, which suffer from the limitations of the language and its associated environment.

Architecture Description Languages. During the past fifteen years, several architecture description languages have been developed, that embody component models and linguistic support for component-based specification and programming (see e.g. [79] for a survey). Of particular interest are ADLs that provide the ability to specify or program dynamically reconfigurable architectures (see e.g. [26] for a survey). These include in particular Rapide [74], Darwin [76, 77], CommUnity [112, 113], Olan [13], Dynamic Wright [7, 6], π-ADL [85]. Of these, CommUnity and π-ADL provide the more expressive power, especially with respect to dynamic reconfigurations. CommUnity is based on the Unity [34] and IP [49] specification languages, and describes a component configuration as a graph with nodes labelled by programs and arcs labelled by morphisms. In CommUnity, a reconfiguration is thus specified by conditional graph rewriting rules. While CommUnity can describe complex reconfigurations, it does not appear possible to specify a situation where a component is to be replaced by an unknown one, received on a communication channel. In π-ADL, components and component specifications are specified as higher-order processes, with process specifications combining process descriptions as in the higher-order π-calculus [94], and behavioral properties expressed in a variant of the μ-calculus [65]. and reconfigurations correspond to higher-order actions effected by processes. Thanks to its higher-order communication, only π-ADL provides the ability to specify dynamic reconfigurations that depend on components received from the environment (e.g. as arguments of messages). However, π-ADL does not provide the equivalent of the packing capability of OZ/K, with its ability to suspend and marshall a running component. Overall, the architecture description languages that provide the more extensive support for dynamic reconfiguration, allow high-level behavioral specifications, but do not explicitly support un-planned dynamic reconfigurations that can be expressed in OZ/K, through a combination of higher-order communication and passivation.

Process calculi. The notion of kell in OZ/K is directly inspired by the Kell calculus [96], and the kell calculus with sharing developed in [61]. The OZ/K notion of packing and unpacking is close to the passivation operator of the M-calculus [95]. For a more in-depth discussion of the relations between kells and localities in other distributed process calculi, such as Mobile Ambients [31] and their many variants, the Distributed π-calculus [89] and its higher-order variant SafeDpi [58], the Seal calculus [32], and Klaim [18], the reader can refer to [96]. The approach advocated in OZ/K, inherited from the Kell calculus and the M-calculus, is the only one to combine a higher-order approach as can be found in SafeDpi, and the possibility to passivate a locality. The Seal calculus can approximate to some extent the effect of passivation, but at the expense of complex encodings in order to simulate simple Kell calculus moves.

Two recent process calculi that offer the possibility of dynamic reconfiguration are the γ-calculus, that embodies a higher-order chemical computation model [14, 15], and Homer [53], that provides for locality passivation as in the Kell calculus. As [16] illustrates, it is possible to program some forms of dynamic reconfigurations in the γ-calculus. However the calculus does not allow for passivating executing chemical solutions (only inert solutions can be matched by pattern matching), which would be the equivalent of passivating a kell. As a result, it is unclear how to support un-planned reconfigurations in the γ-calculus, where part of the system can be modified even though it was not programmed to account for such a reconfiguration. Homer is very close to the Kell calculus, but it allows the passivation of localities from an arbitrary ancestor in the locality tree, provided the path from ancestor to descendant is known. In contrast, OZ/K, as in the Kell calculus, only allows an immediate parent locality to passivate a given locality. This preserves the local aspect of all OZ/K reductions. In addition, it is possible if necessary to encode Homer-type passivation through some form of content controllers à la FRACTAL(see Section 6.7 for an encoding of simple content controllers).

Finally, it is worth noting that, although very close to the kell constructs in OZ/K are close to those of the M-calculus and of the Kell calculus, there are some important differences. The packing operation is very similar to the passivation operation of the M-calculus, but the M-calculus relied on located channels, whereas gates in OZ/K are not. The the `Open` and `Close` operations in OZ/K have no equivalent in the M-calculus and the Kell calculus. In these calculi, it is possible to model the opening of a given gate by the introduction of a relaying process, but it is not possible to model a statement of the form `{Open K#all}`. In fact, *transparent kells* introduced in Section 6, correspond to localities of the Kell calculus with sharing [61].

8 Conclusion

We have presented OZ/K, a kernel language for open distributed programming. The main contribution of OZ/K is the introduction of a notion of locality as a unit of modularity, isolation, and reconfiguration in the multi-paradigm OZ computation model. Localities in OZ/K can be used to model distributed sites, to construct sandboxes, or to program dynamic modules. We have presented a formal operational semantics for OZ/K, and given a number of programming examples illustrating how open distributed programming can be supported in OZ/K.

The language presented in this paper constitutes a first attempt at introducing localities in Oz/K. As mentioned above, several aspects of the language warrant further study: the granularity of localities, the semantics of gate communication, the handling of failures, improved support of component sharing, improved support for dynamic reconfiguration and state capture. In addition, several questions are worth investigating. First, it would be interesting to define a compositional semantics for (possibly a subset of) the Oz/K kernel language. This would allow the development of a behavioral theory for Oz/K, and would ease the definition of type systems for Oz/K. Defining type systems for Oz/K would also be of interest, especially targeting configuration errors as studied e.g. in [20], and dealing with evolving configurations and mobility scenarios as studied e.g. in [58, 115]. A third question would be the development of appropriate support for transactional behavior or recoverable actions, exploiting for instance recent studies on the subject such as [29, 42, 43].

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 13, no 2*, 1991.

[2] F. Acherman. *Forms, agents and channels*. PhD thesis, University of Bern, 2002.

[3] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings 16th European Conference on Object-Oriented Programming (ECOOP)*, 2002.

[4] J. Aldrich, V. Sazawal, C. Chambers, and David Notkin. Language Support for Connector Abstractions. In *Proceedings 17th European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[5] M. Alia, S. Chassande-Barrioz, P. Déchamboux, C. Hamon, and A. Lefebvre. A Middleware Framework for the Persistence and Querying of Java Objects. In *Proceedings ECOOP 2004*, number 3086 in Lecture Notes in Computer Science, 2004.

[6] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[7] R. Allen, D. Garlan, and R. Douence. Specifying Dynamism in Software Architectures. In *Proceedings of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Switzerland, September 1997.

[8] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2), 2002.

[9] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, Stockholm, Sweden, 2003.

[10] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[11] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. R. Kohn, L. C. McInnes, S. R. Parker, and B. A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*. IEEE Computer Society, 1999.

[12] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, 4th edition*. Addison-Wesley, 2005.

[13] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.Y. Vion-Dury. Architecturing and configuring distributed applications with olan. In *Proceedings IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, UK*, 1998.

[14] J.P. Banâtre, P. Fradet, and Y. Radenac. Higher-order chemical programming style. In *Unconventional Programming Paradigms, International Workshop UPP 2004, Revised Selected and Invited Papers*, volume 3566 of *Lecture Notes in Computer Science*. Springer, 2005.

[15] J.P. Banâtre, P. Fradet, and Y. Radenac. A generalized higher-order chemical computation model. *Electr. Notes Theor. Comput. Sci.*, 135(3), 2006.

[16] J.P. Banâtre, Y. Radenac, and P. Fradet. Chemical specification of autonomic systems. In *Proceedings of the ISCA 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE)*. ISCA, 2004.

[17] M. Berger. Basic theory of reduction congruence for two timed asynchronous pi-calculi. In *CONCUR 2004 - Concurrency Theory, 15th International Conference*, volume 3170 of *Lecture Notes in Computer Science*. Springer, 2004.

[18] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The KLAIM Project: Theory and Practice. In *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, number 2874 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

[19] L. Bettini, V. Bono, and B. Venneri. O'Klaim: a coordination language with mobile mixins. In *Proceedings of Coordination 2004*. Springer, 2004.

[20] P. Bidinger, M. Leclercq, V. Quéma, A. Schmitt, and J.B. Stefani. Dream Types - A Domain Specific Type System for Component-Based Message-Oriented Middleware. In */4th Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), in association with ESEC/FSE'05/*, 2005.

[21] P. Bidinger, A. Schmitt, and J.-B. Stefani. An abstract machine for the kell calculus. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS), 7th IFIP WG 6.1 International Conference*, volume 3535 of *Lecture Notes in Computer Science*. Springer, 2005.

[22] A.D. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *in Proceedings 14th Symp. on Operating Systems Principles (SOSP)*, 1993.

[23] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, USA, 1983.

[24] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, N. De Palma, V. Quéma, and J.B. Stefani. Architecture-based autonomous repair management: An application to j2ee clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005*. IEEE Computer Society, 2005.

[25] G. Boudol. Ulm: a core programming model for global computing. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, 2004.

[26] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004*. ACM, 2004.

[27] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[28] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Component Model, *The ObjectWeb Consortium*, `http://www.objectweb.org`, 2004.

[29] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL*. ACM, 2005.

[30] L. Cardelli. Program fragments, linking and modularization. In *24th Symp. on Principles of Programming Languages (POPL '97)*. ACM, 1997.

[31] L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science, vol. 240, no 1*, 2000.

[32] G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *Information and Computation*, 201(1), 2005.

[33] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems*, 17(5), 1995.

[34] K. Chandy and J. Misra. *Parallel Program Design - A Foundation*. Addison-Wesley, 1988.

[35] S.W. Cheng, A.C. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *1st International Conference on Autonomic Computing (ICAC 2004)*. IEEE Computer Society, 2004.

[36] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *Proceedings ECOOP*, 2001.

[37] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *Proceedings 17th European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[38] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01)*, pages 160–178, Heidelberg, Germany, November 2001.

[39] R. Collet and P. Van Roy. Failure Handling in a Network-Transparent Distributed Programming Language. In *Recent Advances in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*. Springer, 2006.

[40] Oz Consortium. Mozart Documentation, 2004. Available at the URL: `http://www.mozart-oz.org/documentation/`.

[41] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. OpenCOM v2: A Component Model for Building Systsms Software. In *Proceedings of IASTED Software Engineering and Applications (SEA '04)*, 2004.

[42] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*. Springer, 2004.

[43] Vincent Danos and Jean Krivine. Transactions in rccs. In *CONCUR 2005 - Concurrency Theory, 16th International Conference*, volume 3653 of *Lecture Notes in Computer Science*. Springer, 2005.

[44] P.C. David and T. Ledoux. Towards a Framework for Self-adaptive Component-Based Applications. In *DAIS*, volume 2893 of *LNCS*. Springer, 2003.

[45] B. Dumant, F. Dang Tran, F. Horn, and J.B. Stefani. Jonathan: an open distributed platform in Java. *Distributed Systems Engineering Journal, vol.6*, 1999.

[46] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *1st EuroSys Conference*. ACM, 2006.

[47] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *USENIX Annual Technical Conference*, 2002.

[48] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: a language for concurrent, distributed and mobile programming. In *Summer Schol Adv. Functional Programming*, volume 2638 of *LNCS*, 2003.

[49] N. Francez and I. Forman. *Interacting Processes : A multiparty approach to coordinated distributed programming*. Addison-Wesley, 1996.

[50] D. Garlan, R. Monroe, and D. Wile. *Acme: Architectural Description of Component-Based Systems*, chapter 3. In [67], 2000.

[51] D. Gelernter. Generative communication in Linda. *ACM Transactions on Prog. Lang. and Systems (TOPLAS)*, 7(1), 1985.

[52] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2), 1992.

[53] J.C. Godskesen and T. T. Hildebrandt. Extending Howe's Method to Early Bisimulations for Typed Mobile Embedded Resources with Local Names. In *Foundations of Software Technology and Theoretical Computer Science, 25th International Conference (FSTTCS)*, volume 3821 of *Lecture Notes in Computer Science*. Springer, 2005.

[54] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 4th edition*. Addison-Wesley, 2005.

[55] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Software Engineering*, 22(2), 1996.

[56] S. Haridi and N. Franzen. Tutorial of Oz, 2004. Available at the URL: `http://www.mozart-oz.org/documentation/tutorial/index.html`.

[57] S. Haridi, P. Van Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. Efficient logic variables for distributed computing. *ACM Trans. Program. Lang. Syst.*, 21(3), 1999.

[58] M. Hennessy, J. Rathke, and N. Yoshida. SafeDpi: a language for controlling mobile code. *Acta Informatica*, 42(4-5), 2005.

[59] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Reorganizing unix for reliability. In *Advances in Computer Systems Architecture, 11th Asia-Pacific Conference, ACSAC 2006*, volume 4186 of *Lecture Notes in Computer Science*. Springer, 2006.

[60] M. W. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6), 2005.

[61] D. Hirschkoff, T. Hirschowitz, D. Pous, A. Schmitt, and J.B. Stefani. Component-oriented programming with sharing: Containment is not ownership. In *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*, volume 3676 of *Lecture Notes in Computer Science*. Springer, 2005.

[62] IBM, BEA et al. Service Component Architecture - Assembly Model Specification, 2005. Specification available at URL: `http://www.ibm.com/developerworks/library/specification/ws-sca/`.

[63] Y. Jaradin, F. Spiessens, and P. Van Roy. Capability Confinement by Membranes. Technical Report Research Report RR2005-03, Dep. of Comp. Science and Eng., UniversitéCatholique de Louvain, Belgium, 2005.

[64] S. Katz. A superimposition control construct for distributed systems. *ACM Trans. on Programming Languages and Systems, vol.15, no 2*, 1993.

[65] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27, 1983.

[66] O. Layaida and D. Hagimont. Designing self-adaptive multimedia applications through hierarchical reconfiguration. In *DAIS*, volume 3543 of *LNCS*. Springer, 2005.

[67] G. Leavens and M. Sitaraman (eds). *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[68] M. Leclercq, V. Quema, and J.B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), 2005.

[69] E. Lee. The problem with threads. *IEEE Computer*, 39(5), 2006.

[70] Y. Liu and S. Smith. A Formal Framework for Component Deployment. In *20th ACM Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[71] Y.D. Liu and S. Smith. Interaction-Based Programming with Classages. In *Proceedings OOPSLA*, 2005.

[72] Y.D. Liu and S.F. Smith. Modules with interfaces for dynamic linking and communication. In *Proceedings European Conf. on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*. Springer, 2004.

[73] J. Lowy. *Programming .Net Components, 2nd edition*. O'Reilly, 2005.

[74] D. Luckham and J. Vera. An event-based architecture definition Language. *IEEE Transactions on Software Engineering, vol. 21, no 9*, 1995.

[75] M. Lumpe, F. Achermann, and O. Nierstrasz. *A Formal Language for Composition*, chapter 4. In [67], 2000.

[76] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings 5th European Software Engineering Conference, Lecture Notes in Computer Science 989, Springer-Verlag*, 1995.

[77] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings 4th ACM Symp. on Foundations of Soft. Eng. (FSE-4)*. ACM Press, 1995.

[78] S. McDirmid, . Flatt, and W.C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings OOPSLA '01, ACM Press*, 2001.

[79] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering, vol. 26, no. 1*, 2000.

[80] M. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, baltimor, Maryland, USA, 2006.

[81] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[82] R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering, Vol. 24, no 5*, 1998.

[83] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings OOPSLA*, 2005.

[84] OMG. CORBA Component Model Specification, version 4.0, formal/06-04-01, 2004.

[85] F. Oquendo. π-ADL: An Architecture Description Language based on the Higher-Order π-Calculus for Specifying Dynamic and Mobile Software Architectures. *ACM Software Engineering Notes*, 29(4), 2004.

[86] OSGI Alliance. OSGI Service Platform – Core Specification – Release 4, Version 4.0.1, 2006.

[87] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambdrige University Press, 2003.

[88] D. Rémy. Using, Understanding, and Unraveling the OCaml Language. In Gilles Barthe, editor, *Applied Semantics. Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[89] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1998.

[90] A. Rossberg. The Missing Link – Dynamic Components for ML. In *Int. Conf. Functional Programming (ICFP)*, 2006.

[91] A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*. Intellect, 2006.

[92] P. Van Roy. On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart. In *Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*. World Scientific, 1999.

[93] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed oz. *ACM Trans. Program. Lang. Syst.*, 19(5), 1997.

[94] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[95] A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[96] A. Schmitt and J.B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*. Springer, 2005.

[97] J. Costa Seco and L. Caires. A basic model of typed components. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2000.

[98] J. Costa Seco and L. Caires. ComponentJ: The Reference Manual. Technical report, TR UNL-DI-6-2002, Departamento de Informatica, U. Nova de Lisboa, 2002.

[99] P. Sewell, J. Leifer, K. Wansbrough, M. Allen-Willians, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation – Design rationale and language definition. Technical Report RR-5329, INRIA, 2004.

[100] P. Sewell, J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Willians, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. In *Int. Conf. Functional Programming*, 2005.

[101] P. Sewell, J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Willians, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *to appear Journal of Functional Programming*, 2006.

[102] P. Sewell and J. Vitek. Secure Composition of Untrusted Code: Box pi, Wrappers, and Causality. *Journal of Computer Security*, 11(2), 2003.

[103] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[104] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12), 2002.

[105] F. Spiessens and P. Van Roy. The oz-e project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/Oz, 2nd International Conference, MOZ 2004*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.

[106] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *32nd POPL*. ACM, 2005.

[107] Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements, 2006.

[108] C. Szyperski. *Component Software, 2nd edition*. Addison-Wesley, 2002.

[109] J.C. Tournier, J.P. Babau, and V. Olive. Qinna, a component-based qos architecture. In *CBSE*, volume 3489 of *LNCS*. Springer, 2005.

[110] P. van Roy. Self-Management and the Future of Software Design. In *FACS*, Electronic Notes in Theoretical Computer Science, URL: `http://www.elsevier.nl/locate/entcs`, 2006.

[111] P. van Roy and S. Haridi. *Concepts, Techniques and Models of Computer Programming*. MIT Press, 2004.

[112] M. Wermelinger and J. Fiadeiro. A graph transformation approach to run-time software architecture reconfiguration. In *Proceedings ESEC/FSE 1999, LNCS 1687*. Springer-Verlag, 1999.

[113] M. Wermelinger, A. Lopes, and J. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proceedings ESEC/FSE 2001, Software Engineering Notes 26(5)*. ACM Press, 2001.

[114] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.

[115] N. Yoshida. Channel dependent types for higher-order mobile processes. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 2004.

$$\mathtt{v}(u) \triangleq \emptyset \qquad \mathtt{v}(!\epsilon) \triangleq \mathtt{v}(\epsilon)$$
$$\mathtt{v}(\langle X \rangle) \triangleq \emptyset \qquad \mathtt{v}(\epsilon(\epsilon_1 : X_1 \ldots \epsilon_n : X_n)) \triangleq \mathtt{v}(\epsilon, \epsilon_1, \ldots, \epsilon_n)$$
$$\mathtt{v}(\mathbf{skip}\ ) = \emptyset \qquad \mathtt{v}(S_1 \ \ S_2) = \mathtt{v}(S_1, S_2)$$
$$\mathtt{v}(\mathbf{thread}\{\epsilon\}\ S\ \mathbf{end}) = \mathtt{v}(\epsilon, S) \qquad \mathtt{v}(\mathbf{local}\ X_1 \ldots X_n\ \mathbf{in}\ S\ \mathbf{end}) = \mathtt{v}(S)$$
$$\mathtt{v}(\epsilon = \epsilon') = \mathtt{v}(\epsilon, \epsilon') \qquad \mathtt{v}(\epsilon = !!\epsilon') = \mathtt{v}(\epsilon, \epsilon')$$
$$\mathtt{v}(\epsilon = u) = \mathtt{v}(\epsilon) \qquad \mathtt{v}(\epsilon = l(f_1 : \epsilon_1 \ldots f_n : \epsilon_n)) = \mathtt{v}(\epsilon, \epsilon_1, \ldots, \epsilon_n)$$
$$\mathtt{v}(\mathbf{if}\ \epsilon\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}) = \mathtt{v}(\epsilon, S_1, S_2) \qquad \mathtt{v}(\mathbf{case}\ \epsilon\ \mathbf{of}\ J\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2) = \mathtt{v}(\epsilon, J, S_1, S_2)$$
$$\mathtt{v}(\mathbf{proc}\{\epsilon\ X_1 \ldots X_n\}\ S\ \mathbf{end}) = \mathtt{v}(\epsilon, S) \qquad \mathtt{v}(\{\epsilon\ \epsilon_1 \ldots \epsilon_n\}) = \mathtt{v}(\epsilon, \epsilon_1, \ldots, \epsilon_n)$$
$$\mathtt{v}(\mathbf{try}\ S_1\ \mathbf{catch}\ X\ \mathbf{then}\ S_2\ \mathbf{end}) = \mathtt{v}(S_1, S_2) \qquad \mathtt{v}(\mathbf{raise}\ \epsilon\ \mathbf{end}) = \mathtt{v}(\epsilon)$$
$$\mathtt{v}(\mathbf{kell}\{\epsilon\}\ S\ \mathbf{end}) = \mathtt{v}(\epsilon, S) \qquad \mathtt{v}(\{\mathtt{P}\ \epsilon_1\ \ldots\ \epsilon_n\}) = \mathtt{v}(\epsilon_1, \ldots, \epsilon_n)$$

Figure 2: Variables of an extended statement

A Auxiliary relations and predicates

The definition of the reduction relation relies on a number of functions, predicates and relations which we define in this section.

Primitive operations. We call *primitive operations* the operations `Unify`, `NewName`, `IsDet`, `NewCell`, `Exchange`, `WaitNeeded`, `FailedValue`, `NewGate`, `Send`, `Receive`, `Open`, `Close`, `Pack`, `Unpack`, `Mark`, `Status`, that appear in Tables 1 and 3.

Variables. Notions of free and bound variable identifiers are classical. Variable identifier binders are the following statements, which bind variable identifiers `X1`, . . ., `Xn`, with scope the statement `S1`:

```
local X1 ... Xn in S1 end
proc{P X1 ... Xn} S1 end
case X of V(V1:X1 ... Vn:Xn) then S1 else S2 end
try S catch X1 then S1 end
```

The set of variables of an extended statement S, noted $\mathtt{v}(S)$, is defined inductively in Figure 2, where `P` denotes a primitive operation, where u denotes a base value (integer, atom or name), and where ϵ, δ, and their decorated variants, denote both variable identifiers and variables. By definition, we set $\mathtt{v}(\epsilon) = \{x\}$ if $\epsilon = x$ (i.e. ϵ is a variable), and $\mathtt{v}(\epsilon) = \emptyset$ if $\epsilon = X$ (i.e. ϵ is a variable identifier). Also, if $T_1, \ldots, T_n$ are terms, we set:

$$\mathtt{v}(T_1, \ldots, T_n) = \mathtt{v}(T_1) \cup \ldots \cup \mathtt{v}(T_n)$$

The set of variables of task $\mathcal{T}$, relative to store σ, noted $\mathtt{v}(\mathcal{T}, \sigma)$, is defined as the smallest set satisfying the inference rules in Figure 3.

The set of variables of a store σ, noted $\mathtt{v}(\sigma)$ is defined inductively in Figure 4.

We define the substitution of variable identifiers by variables in a statement. We write

$$\theta = \{X_1 \to x_1, \ldots, X_n \to x_n\}$$

for the substitution that substitutes variables x_i to identifiers X_i, and $S\theta$ for the application of substitution θ to the extended statement S. We define $\theta_{\{X_1,\ldots,X_n\}}$ to be the substitution that coincides with θ on $\mathtt{dom}(\theta) \setminus \{X_1, \ldots, X_n\}$, i.e.

$$\theta_{\{X_1,\ldots,X_n\}} \triangleq \{X \to x \in \theta \mid X \notin \{X_1, \ldots, X_n\}\}$$

$$\frac{x \in \mathtt{v}(S) \vee x \in \mathtt{v}(\tau : T, \sigma)}{x \in \mathtt{v}(\tau\langle S\ \ T\rangle, \sigma)} \qquad \frac{x \in \mathtt{v}(\mathcal{T}, \sigma) \vee x \in \mathtt{v}(\mathcal{U}, \sigma)}{x \in \mathtt{v}(\mathcal{T}\ \ \mathcal{U}, \sigma)} \qquad \frac{x \in \mathtt{v}(\mathcal{T}, \sigma)\ \ \sigma \models x = y}{y \in \mathtt{v}(\mathcal{T}, \sigma)}$$

$$\frac{x \in \mathtt{v}(\mathcal{T}, \sigma)\ \ \sigma \models x = \xi \wedge \xi : \mathtt{cell}(y)}{y \in \mathtt{v}(\mathcal{T}, \sigma)} \qquad \frac{x \in \mathtt{v}(\mathcal{T}, \sigma)\ \ \sigma \models x = \xi \wedge \xi : \mathtt{thread}(y)}{y \in \mathtt{v}(\mathcal{T}, \sigma)}$$

$$\frac{x \in \mathtt{v}(\mathcal{T}, \sigma)\ \ \sigma \models x = \xi \wedge \xi : \mathtt{kell}(\pi, y)}{y \in \mathtt{v}(\mathcal{T}, \sigma)} \qquad \frac{x \in \mathtt{v}(\mathcal{T}, \sigma)\ \ \sigma \models \mathtt{read}(x, y)}{y \in \mathtt{v}(\mathcal{T}, \sigma)} \qquad \frac{x \in \mathtt{v}(\mathcal{T}, \sigma)\ \ \sigma \models x = \mathtt{failed}(y)}{y \in \mathtt{v}(\mathcal{T}, \sigma)}$$

$$\frac{x \in \mathtt{v}(\mathcal{T}, \sigma)\ \ \sigma \models x = f(l_1 : x_1, \dots, l_n : x_n)_{\mathtt{m}}}{x_i \in \mathtt{v}(\mathcal{T}, \sigma)} \qquad \frac{x \in \mathtt{v}(\mathcal{T}, \sigma)\ \ \sigma \models x = \xi \wedge \xi : \mathtt{proc}\{\$\ X_1 \dots X_n\}S\ \mathtt{end}\ \ y \in \mathtt{v}(S)}{y \in \mathtt{v}(\mathcal{T}, \sigma)}$$

Figure 3: Variables of a task relative to a store

$$\begin{aligned}
\mathtt{v}(x) &= \{x\} & \mathtt{v}(x = l(f_1 : x_1 \dots f_n : x_n)_{\mathtt{m}}) &= \{x, x_1, \dots, x_n\} \\
\mathtt{v}(x = u) &= \{x\} & \mathtt{v}(x = y) &= \{x, y\} \\
\mathtt{v}(x = \mathtt{pack}(\xi, \mathcal{T}, \sigma', \mu)) &= \mathtt{v}(\mathcal{T}, \sigma') \cup \mathtt{v}(\sigma') & \mathtt{v}(x = \mathtt{failed}(y)) &= \{x, y\} \\
\mathtt{v}(\xi : \mathtt{proc}\{\$\ X_1 \dots X_n\}S\ \mathtt{end}) &= \mathtt{v}(S, \sigma) & \mathtt{v}(\xi : \mathtt{thread}(x, y)) &= \{x, y\} \\
\mathtt{v}(\xi : \mathtt{kell}(\pi, x)) &= \{x\} & \mathtt{v}(\xi : \mathtt{cell}(x)) &= \{x\} \\
\mathtt{v}(\xi : \mathtt{gate}) &= \emptyset & \mathtt{v}(\mathtt{need}(x)) &= \{x\} \\
\mathtt{v}(\mathtt{read}(x)) &= \{x\} & \mathtt{v}(\mathtt{read}(x, y)) &= \{x, y\} \\
\mathtt{v}(\mathtt{in}(\kappa, \kappa')) &= \emptyset & \mathtt{v}(\mathtt{subg}(\gamma, \gamma')) &= \emptyset \\
\mathtt{v}(\mathtt{inth}(\kappa, \tau)) &= \emptyset & \mathtt{v}(\sigma \wedge \sigma') &= \mathtt{v}(\sigma) \cup \mathtt{v}(\sigma')
\end{aligned}$$

Figure 4: Variables of a store

$$\epsilon\theta \triangleq \theta(\epsilon) \quad \text{if } \epsilon \in \mathtt{dom}(\theta) \qquad \epsilon\theta \triangleq \epsilon \quad \text{if } \epsilon \notin \mathtt{dom}(\theta) \qquad v\theta \triangleq v$$

$$(!\epsilon)\theta \triangleq !(\epsilon\theta) \qquad (\epsilon(\epsilon_1 : X_1 \dots \epsilon_n : X_n))\theta \triangleq \epsilon\theta(\epsilon_1\theta : X_1 \dots \epsilon_n\theta : X_n)$$

$$\begin{aligned}
\mathbf{skip}\ \theta &\triangleq \mathbf{skip} & (S_1\ \ S_2)\theta &\triangleq S_1\theta\ \ S_2\theta \\
(\mathbf{thread}\{\epsilon\}\ S\ \mathbf{end})\theta &\triangleq \mathbf{thread}\{\epsilon\theta\}\ S\theta\ \mathbf{end} & (\epsilon_1 = \epsilon_2)\theta &\triangleq \epsilon_1\theta = \epsilon_2\theta \\
(\epsilon = v)\theta &\triangleq \epsilon\theta = v & (\epsilon_1 = \epsilon_2.\epsilon_3)\theta &\triangleq \epsilon_1\theta = \epsilon_2\theta.\epsilon_3.\theta \\
\{\mathtt{P}\ \ \epsilon_1 \dots \epsilon_n\}\theta &\triangleq \{\mathtt{P}\ \ \epsilon_1\theta \dots \epsilon_n\theta\} & \{\epsilon\ \ \epsilon_1 \dots \epsilon_n\}\theta &\triangleq \{\epsilon\theta\ \ \epsilon_1\theta \dots \epsilon_n\theta\} \\
(\mathbf{kell}\{\epsilon_1\}\ S\ \mathbf{end})\theta &\triangleq \mathbf{kell}\{\epsilon_1\theta\}\ S\theta\ \mathbf{end} & (\epsilon_1 = !!\epsilon_2)\theta &\triangleq \epsilon_1\theta = !!\epsilon_2\theta
\end{aligned}$$

$$\begin{aligned}
(\mathbf{if}\ \epsilon\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end})\theta &\triangleq \mathbf{if}\ \epsilon\theta\ \mathbf{then}\ S_1\theta\ \mathbf{else}\ S_2\theta\ \mathbf{end} \\
(\mathbf{raise}\ \epsilon\ \mathbf{end})\theta &\triangleq \mathbf{raise}\ \epsilon\theta\ \mathbf{end} \\
(\mathbf{local}\ X_1 \dots X_n\ \mathbf{in}\ S\ \mathbf{end})\theta &\triangleq \mathbf{local}\ X_1 \dots X_n\ \mathbf{in}\ S\theta_{\{X_1, \dots, X_n\}}\ \mathbf{end} \\
(\mathbf{proc}\{\epsilon\ X_1 \dots X_n\}\ S\ \mathbf{end})\theta &\triangleq \mathbf{proc}\{\epsilon\theta\ X_1 \dots X_n\}\ S\theta_{\{X_1, \dots, X_n\}}\ \mathbf{end} \\
(\mathbf{try}\ S_1\ \mathbf{catch}\ X\ \mathbf{then}\ S_2\ \mathbf{end})\theta &\triangleq \mathbf{try}\ S_1\theta\ \mathbf{catch}\ X\ \mathbf{then}\ S_2\theta_{\{X\}}\ \mathbf{end} \\
(\mathbf{case}\ \epsilon\ \mathbf{of}\ J\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end})\theta &\triangleq \mathbf{case}\ \epsilon\theta\ \mathbf{of}\ J\theta\ \mathbf{then}\ S_1\theta_{\mathtt{bv}(J)}\ \mathbf{else}\ S_2\theta\ \mathbf{end}
\end{aligned}$$

Figure 5: Substitution on statements and patterns

Using ϵ and its decorated variants to stand for a variable or a variable identifier, and P to stand for any of the primitive operations, we define by induction in Figure 5 the application of a substitution θ to an extended statement S (and a pattern J). In Figure 5, we define $\mathtt{bv}(J)$ as follows:

$$\mathtt{bv}(v) \triangleq \emptyset \qquad \mathtt{bv}(!X) \triangleq \emptyset \qquad \mathtt{bv}(\epsilon(\epsilon_1 : X_1 \ldots \epsilon_n : X_n)) \triangleq \{X_1, \ldots, X_n\}$$

Names. We define here functions and predicate dealing with names. The set of gate names $\mathtt{gn}(\mathcal{T}, \sigma)$ of an execution structure $(\mathcal{T}, \sigma)$ is defined as follows:

$$\mathtt{gn}(\mathcal{T}, \sigma) = \{\gamma \in \mathtt{name} \mid \exists x,\ x \in \mathtt{v}(\mathcal{T}, \sigma),\ \sigma \models x = \gamma \wedge \gamma : \mathtt{gate}\}$$

The set of thread names $\mathtt{thn}(\mathcal{T}, \sigma)$ of an execution structure $(\sigma, \mathcal{T})$ is defined inductively as follows:

$$\mathtt{thn}(\tau : T, \sigma) = \{\tau\} \qquad \mathtt{thn}(\mathcal{T}_1\ \ \mathcal{T}_2, \sigma) = \mathtt{thn}(\mathcal{T}_1, \sigma) \cup \mathtt{thn}(\mathcal{T}_2, \sigma)$$

The set of kell names $\mathtt{kn}(\mathcal{T}, \sigma)$ of an execution structure $(\sigma, \mathcal{T})$ is defined as follows:

$$\mathtt{kn}(\mathcal{T}, \sigma) = \{\kappa \in \mathtt{name} \mid \exists x, y, \pi,\ x \in \mathtt{v}(\mathcal{T}, \sigma)\ \wedge\ \sigma \models x = \kappa \wedge \kappa : \mathtt{kell}(\pi, y)\}$$

The set of procedure names $\mathtt{pn}(\mathcal{T}, \sigma)$ of an execution structure $(\mathcal{T}, \sigma)$ is defined as follows:

$$\mathtt{pn}(\mathcal{T}, \sigma) = \{\xi \in \mathtt{name} \mid \exists x, y, X_i, S,\ x \in \mathtt{v}(\mathcal{T}, \sigma)\ \wedge\ \sigma \models x = \xi \wedge \xi : \mathtt{proc}\{\$\ X_1 \ldots X_n\}S\ \mathtt{end}\}$$

The function kgpn that returns the set of gate, procedure and kell names of an execution structure $(\mathcal{T}, \sigma)$ is defined as:

$$\mathtt{kgpn}(\mathcal{T}, \sigma) = \mathtt{gn}(\mathcal{T}, \sigma) \cup \mathtt{pn}(\mathcal{T}, \sigma) \cup \mathtt{kn}(\mathcal{T}, \sigma)$$

The function $\mathtt{tkn}_\sigma$ returns the names of top level kells in a task, relative to a store σ. It is defined as follows, where $\top$ denotes by convention the name of the top-level kell:

$$\mathtt{tkn}_\sigma(\mathcal{T}) = \{\eta \in \mathtt{kn}(\mathcal{T}, \sigma) \mid \sigma \models \mathtt{in}(\top, \eta))\}$$

Equivalence relation. The reduction relation makes use of an equivalence relation, noted $\equiv$, between statements, between tasks and between stores. The equivalence relation between statement, noted $\equiv$, is the smallest equivalence relation that obeys the rules given in Figure 6.

The equivalence relation between tasks, also noted $\equiv$, is the smallest equivalence relation that obeys the rules in Figure 7.

The equivalence relation between stores, also noted $\equiv$, is the smallest relation that obeys the rules in Figure 8, where δ denotes a variable or a value, and where P and its decorated variants denote a store predicate of the form $\mathtt{proc}\{\$\ X_1 \ldots X_n\}S\ \mathtt{end}$.

Entailment between stores. The rules in Figure 8 define also an entailment relation between stores and stores. The entailment relation $\models$ is defined as the smallest relation that obeys the rules in Figure 8.

The domain of a store σ, noted $\mathtt{dom}(\sigma)$, is the set of variables and names that occur in σ. It is defined as

$$\mathtt{dom}(\sigma) = \mathtt{v}(\sigma) \cup \{\xi \in \mathtt{Name} \mid \exists x \in \mathtt{dom}(\sigma),\ \sigma \models x = \xi \vee x = \xi(...)\}$$

$$[\textsc{S.}\alpha]\ \frac{S_1 =_\alpha S_2}{S_1 \equiv S_2} \qquad [\textsc{S.seq}]\ \frac{S_1 \equiv S_2 \quad S_1' \equiv S_2'}{S_1\ S_1' \equiv S_2\ S_2'} \qquad [\textsc{S.local}]\ \frac{S_1 \equiv S_2}{\texttt{local}\ X_1 \ldots X_n\ \texttt{in}\ S_1\ \texttt{end} \equiv \texttt{local}\ X_1 \ldots X_n\ \texttt{in}\ S_2\ \texttt{end}}$$

$$[\textsc{S.thread}]\ \frac{S_1 \equiv S_2}{\texttt{thread}\{X\}\ S_1\ \texttt{end} \equiv \texttt{thread}\{X\}\ S_2\ \texttt{end}}$$

$$[\textsc{S.if}]\ \frac{S_1 \equiv S_2 \quad S_3 \equiv S_4}{\texttt{if}\ X\ \texttt{then}\ S_1\ \texttt{else}\ S_3\ \texttt{end} \equiv \texttt{if}\ X\ \texttt{then}\ S_2\ \texttt{else}\ S_4\ \texttt{end}}$$

$$[\textsc{S.case}]\ \frac{S_1 \equiv S_2 \quad S_3 \equiv S_4}{\texttt{case}\ X\ \texttt{of}\ J\ \texttt{then}\ S_1\ \texttt{else}\ S_3 \equiv \texttt{case}\ X\ \texttt{of}\ J\ \texttt{then}\ S_2\ \texttt{else}\ S_4}$$

$$[\textsc{S.try}]\ \frac{S_1 \equiv S_2 \quad S_3 \equiv S_4}{\texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_3\ \texttt{end} \equiv \texttt{try}\ S_2\ \texttt{catch}\ X\ \texttt{then}\ S_4\ \texttt{end}}$$

$$[\textsc{S.proc}]\ \frac{S_1 \equiv S_2}{\texttt{proc}\{P\ X_1 \ldots X_n\}\ S_1\ \texttt{end} \equiv \texttt{proc}\{P\ X_1 \ldots X_n\}\ S_2\ \texttt{end}} \qquad [\textsc{S.kell}]\ \frac{S_1 \equiv S_2}{\texttt{kell}\{K\}\ S_1\ \texttt{end} \equiv \texttt{kell}\{K\}\ S_2\ \texttt{end}}$$

Figure 6: Equivalence between statements

$$[\textsc{T.thread}]\ \frac{S_1 \equiv S_2 \quad \mathcal{T}_1 \equiv \mathcal{T}_2}{\eta : \langle S_1\ \ \mathcal{T}_1\rangle \equiv \eta : \langle S_2\ \ \mathcal{T}_2\rangle} \qquad [\textsc{T.comm}]\ \mathcal{T}_1\ \ \mathcal{T}_2 \equiv \mathcal{T}_2\ \ \mathcal{T}_1 \qquad [\textsc{T.assoc}]\ \mathcal{T}_1\ \ (\mathcal{T}_2\ \ \mathcal{T}_3) \equiv (\mathcal{T}_1\ \ \mathcal{T}_2)\ \ \mathcal{T}_3$$

$$[\textsc{T.par}]\ \frac{\mathcal{T}_1 \equiv \mathcal{T}_2}{\mathcal{T}_1\ \ \mathcal{T} \equiv \mathcal{T}_2\ \ \mathcal{T}}$$

Figure 7: Equivalence between tasks

$$[\textsc{E.pack}]\ \frac{\mathcal{T}_1 \equiv \mathcal{T}_2 \quad \sigma_1 \equiv \sigma_2}{x = \mathtt{pack}(\zeta, \mathcal{T}_1, \sigma_1, \mu) \equiv x = \mathtt{pack}(\zeta, \mathcal{T}_2, \sigma_2, \mu)} \qquad [\textsc{E.proc}]\ \frac{\mathrm{P} \equiv \mathrm{P}'}{\xi : \mathrm{P} \equiv \xi : \mathrm{P}'} \qquad [\textsc{E.equal}]\ x = y \equiv y = x$$

$$[\textsc{E.elim1}]\ \frac{\sigma \equiv \sigma_1 \wedge \sigma_2}{\sigma \models \sigma_1} \qquad [\textsc{E.elim2}]\ \frac{\sigma \equiv \sigma_1 \wedge \sigma_2}{\sigma \models \sigma_2} \qquad [\textsc{E.intro}]\ \frac{\sigma \models \sigma_1 \quad \sigma \models \sigma_2}{\sigma \models \sigma_1 \wedge \sigma_2} \qquad [\textsc{E.equalt}]\ \frac{\sigma \models y = \delta \wedge x = y}{\sigma \models x = \delta}$$

$$[\textsc{E.entails}]\ \frac{\sigma \models \sigma' \quad \sigma' \models \sigma}{\sigma \equiv \sigma'} \qquad [\textsc{E.reflex}]\ \sigma \models \sigma \qquad [\textsc{E.trans}]\ \frac{\sigma_1 \models \sigma_2 \quad \sigma_2 \models \sigma_3}{\sigma_1 \models \sigma_3}$$

Figure 8: Equivalence between stores

$$[\text{EQ.BASE}]\ \frac{u = u'}{u \equiv_\sigma u'} \qquad [\text{EQ.EQUAL}]\ \frac{\sigma \models x = y}{x \equiv_\sigma y} \qquad [\text{EQ.VAR}]\ \frac{\sigma \models x = v \wedge y = v' \quad v \equiv_\sigma v'}{x \equiv_\sigma y}$$

$$[\text{EQ.PACK}]\ \frac{\mathcal{T}_1 \equiv \mathcal{T}_2 \quad \sigma_1 \equiv \sigma_2}{\texttt{pack}(\zeta, \mathcal{T}_1, \sigma_1, \mu) \equiv_\sigma \texttt{pack}(\zeta, \mathcal{T}_2, \sigma_2, \mu)} \qquad [\text{EQ.FAILED}]\ \frac{x \equiv_\sigma y}{\texttt{failed}(x) \equiv_\sigma \texttt{failed}(y)}$$

$$[\text{EQ.RECORD}]\ \frac{x_1 \equiv_\sigma y_1 \ \ldots \ x_n \equiv_\sigma y_n}{l(f_1 : x_1 \ldots f_n : x_n) \equiv_\sigma l(f_1 : y_1 \ldots f_n : y_n)}$$

Figure 9: Equality between values

$$[\text{DIS.BASE}]\ \frac{u \neq u'}{u \bowtie_\sigma u'} \qquad [\text{DIS.VAR}]\ \frac{\sigma \models x = v \wedge y = v' \quad v \bowtie_\sigma v'}{x \bowtie_\sigma y} \qquad [\text{DIS.PACK}]\ \frac{\neg\exists \mathcal{T}_2, \sigma_2,\ v = \texttt{pack}(\zeta, \mathcal{T}_2, \sigma_2, \mu)}{v \bowtie_\sigma \texttt{pack}(\zeta, \mathcal{T}_1, \sigma_1, \mu)}$$

$$[\text{DIS.PACKD}]\ \frac{v = \texttt{pack}(\zeta, \mathcal{T}_2, \sigma_2, \mu) \ \ \mathcal{T}_2 \not\equiv \mathcal{T}_1 \vee \sigma_2 \not\equiv \sigma_2}{v \bowtie_\sigma \texttt{pack}(\zeta, \mathcal{T}_1, \sigma_1, \mu)} \qquad [\text{DIS.FAIL}]\ \frac{\neg\exists x,\ v = \texttt{failed}(x)}{v \bowtie_\sigma \texttt{failed}(y)}$$

$$[\text{DIS.FAILD}]\ \frac{v = \texttt{failed}(x) \ \ x \bowtie_\sigma y}{v \bowtie_\sigma \texttt{failed}(y)} \qquad [\text{DIS.REC}]\ \frac{\neg\exists x_1, \ldots, x_n,\ v = l(f_1 : x_1 \ldots f_n : x_n)}{v \bowtie_\sigma l(f_1 : y_1 \ldots f_n : y_n)}$$

$$[\text{DIS.RECD}]\ \frac{v = l(f_1 : x_1 \ldots f_n : x_n) \quad \bigvee_{i=1}^{n} x_i \bowtie_\sigma y_i}{v \bowtie_\sigma l(f_1 : y_1 \ldots f_n : y_n)}$$

Figure 10: Inequality between values

The extension of the entailment relation to first-order formulas is classical and is given by the rules below, where ϵ denotes a variable x or a name ξ.

$$\begin{array}{rcr}
\sigma \models \neg\phi & \Longleftrightarrow & \neg(\sigma \models \phi) \\
\sigma \models \phi \vee \psi & \Longleftrightarrow & (\sigma \models \phi) \vee (\sigma \models \psi) \\
\sigma \models \forall\epsilon, \phi & \Longleftrightarrow & \forall\epsilon \in \texttt{dom}(\sigma), (\sigma \models \phi) \\
\sigma \models \exists\epsilon, \phi & \Longleftrightarrow & \exists\epsilon \in \texttt{dom}(\sigma), (\sigma \models \phi)
\end{array}$$

We note $\sigma \models x = \bot$ if $x \in \texttt{dom}(\sigma)$ and $\neg\exists v,\ \sigma \models x = v$.

Equality between values. The notion of equality between values during execution is captured by relations $\equiv_\sigma$ and $\bowtie_\sigma$. Intuitively, two values are equal, relative to a store σ if they correspond to the same base value, or to the same packed value, or to the same failed value, or to the same record. The relation $\equiv_\sigma$ is defined as the smallest equivalence relation that satisfies the rules in Figure 9, where u denotes a base value (integer, atom, or name), and v, v' denote arbitrary values ($v, v' \neq \bot$).

The relation $\bowtie_\sigma$ is defined as the smallest relation that satisfies the rules in Figure 10, where u denotes a base value (integer, atom, or name), and v, v' denote arbitrary values ($v, v' \neq \bot$). Intuitively, the relation $\bowtie_\sigma$ expresses the fact that two values can be proved to be non-equal in store σ, regardless of future bindings.

Invalid stores. Intuitively, a store is invalid if it binds different values to the same variable or to the same name, or if its does not obey structural invariants such as e.g. the fact that a kell can only have one parent kell. We note $\texttt{in}^+$ the transitive closure of the relation `in`, and $\texttt{subg}^+$ the transitive closure of the relation `subg`. We note $\sigma = \bot$ to indicate that store σ is invalid. We define the predicate $\sigma \equiv \bot$ in Figure 11, where `P`, `Q` denote store predicates of the forms $\texttt{cell}(x)$, `gate`, $\texttt{thread}(x)$, $\texttt{kell}(\pi, x)$, or $\texttt{proc}\{\$\ X_1 \ldots X_n\} S$ `end`.

$$
\begin{array}{lll}
\sigma \equiv \bot \triangleq & \exists x, v, v',\ \sigma \models x = v \wedge x = v' \ \wedge\ v \bowtie_\sigma v' & (1)\\
& \vee\, \exists \xi, \mathtt{P}, \mathtt{Q}, \sigma \models\ \xi : \mathtt{P} \wedge \xi : \mathtt{Q}\ \wedge\ P \not\equiv Q & (2)\\
& \vee\, \exists \tau, x,\ \sigma \models \tau : \mathtt{thread}(x)\ \wedge\ \neg\mathtt{read}(x) & (3)\\
& \vee\, \exists \kappa, \pi, x,\ \sigma \models \kappa : \mathtt{kell}(\pi, x)\ \wedge\ \neg\mathtt{read}(x) & (4)\\
& \vee\, \exists \kappa, \kappa', \kappa'',\ \kappa \neq \kappa' \wedge \sigma \models \mathtt{in}(\kappa, \kappa'') \wedge \mathtt{in}(\kappa', \kappa'') & (5)\\
& \vee\, \exists \kappa, \kappa', \tau,\ \kappa \neq \kappa' \wedge \sigma \models \mathtt{inth}(\kappa, \tau) \wedge \mathtt{in}(\kappa', \tau) & (6)\\
& \vee\, \exists \kappa, \kappa',\ \sigma \models \mathtt{in}(\kappa, \kappa') \wedge \neg\exists \pi, x, \pi', x',\ \sigma \models \kappa : \mathtt{kell}(\pi, x) \wedge \kappa' : \mathtt{kell}(\pi', x') & (7)\\
& \vee\, \exists \kappa, \tau,\ \sigma \models \mathtt{inth}(\kappa, \tau) \wedge \neg\exists \pi, x, y,\ \sigma \models \kappa : \mathtt{kell}(\pi, x) \wedge \tau : \mathtt{thread}(y) & (8)\\
& \vee\, \exists \kappa, \kappa',\ \sigma \models \mathtt{in}^+(\kappa, \kappa')\ \wedge\ \sigma \models \mathtt{in}^+(\kappa, \kappa') & (9)\\
& \vee\, \exists \gamma, \gamma',\ \sigma \models \mathtt{subg}(\gamma, \gamma')\ \wedge\ \neg\sigma \models \gamma : \mathtt{gate} \wedge \gamma' : \mathtt{gate} & (10)\\
& \vee\, \exists \gamma, \gamma',\ \sigma \models \mathtt{subg}^+(\gamma, \gamma')\ \wedge\ \sigma \models \mathtt{subg}^+(\gamma', \gamma) & (11)
\end{array}
$$

Figure 11: Invalid stores

$$
\frac{v \in \mathtt{int} \cup \mathtt{atom}}{\mathtt{strict}_\sigma(v)}
\qquad
\frac{\xi \in \mathtt{name} \quad \sigma \not\models \xi : \mathtt{p}(\ldots)}{\mathtt{strict}_\sigma(v)}
\qquad
\frac{\xi \in \mathtt{name} \quad \sigma \models \xi : \mathtt{gate}}{\mathtt{strict}_\sigma(\xi)}
\qquad
\frac{v = \mathtt{pack}(\zeta, \mathcal{T}, \sigma, \mu)}{\mathtt{strict}_\sigma(v)}
$$

$$
\frac{\mathtt{strict}_\sigma(v) \quad \sigma \models x = v}{\mathtt{strict}_\sigma(x)}
\qquad
\frac{\mathtt{strict}_\sigma(x)}{\mathtt{strict}_\sigma(\mathtt{failed}(x))}
\qquad
\frac{\mathtt{strict}_\sigma(v_1) \ldots \mathtt{strict}_\sigma(v_n) \quad \sigma \models x_1 = v_1 \wedge \ldots \wedge x_n = v_n}{\mathtt{strict}_\sigma(f(l_1 : x_1, \ldots, l_n : x_n))}
$$

$$
\frac{\xi \in \mathtt{name} \quad \sigma \models \xi : P \quad P = \mathtt{proc}\{\$\ X_1 \ldots X_n\} S\ \mathtt{end} \quad \forall x \in \mathtt{v}(S, \sigma)\ \mathtt{strict}_\sigma(x)}{\mathtt{strict}_\sigma(\xi)}
$$

Figure 12: Definition of strictness

Strictness. We say that a value v is *strict* relative to σ, noted $\mathtt{strict}_\sigma(v)$, if v is either an integer or an atom, if v is a pure name (i.e. not a gate name, a cell name, a thread name, a kell name, or a procedure name), if v is a record value which contains only variables bound to strict values, or if v is a name bound to a gate, or a procedure whose free variables are bound, recursively, to strict values. Formally, the predicate `strict` is defined as the smallest predicate verifying the rules given in Figure 12, where p ranges over the set of semantical predicates $\{\mathtt{proc}, \mathtt{cell}, \mathtt{thread}, \mathtt{kell}, \mathtt{gate}\}$. We extend the `strict` function into a predicate on pairs of the form $\langle$extended statement, set of variables$\rangle$ thus:

$$\mathtt{strict}_\sigma(S, V) \triangleq \forall x \in \mathtt{v}(S, \sigma) \setminus V, \mathtt{strict}_\sigma(x)$$

Matching. The reduction relation depends also on a function `match` that operates on lists of values and patterns. Function `match` is defined inductively by the table below ($\mathtt{match}_\sigma(x, J)$ is defined to be $\bot$, where $\bot$ denotes a match failure, in all other cases). We note `Id` the trivial substitution, i.e. the substitution whose domain is empty (and thus, for all terms S, $S\mathtt{Id} = S$).

σ	J	$\mathtt{match}_\sigma(x, J)$
$\sigma \models x = v$	v	Id
$\sigma \models x = v \wedge y = v$	$!y$	Id
$\sigma \models x = v_0(v_1 : x_1 \ldots v_n : x_n)_{\mathtt{r}} \wedge \epsilon_i = v_i$	$\epsilon_0(\epsilon_1 : X_1 \ldots \epsilon_n : X_n)$	$\{X_1 \rightarrow x_1, \ldots, X_n \rightarrow x_n\}$

Unification The function `Unify` is defined by

$$\mathtt{Unify}(x, y, \sigma) = \mathtt{fst}(\mathtt{U}(x, y, \sigma, \emptyset))$$

with the function U defined inductively by the following rules, where the function fst returns the first element of a pair, the function snd returns the second element of a pair, v denotes an arbitrary value, and u, u' denote arbitrary base values, i.e. integers, names or atoms (note that a failed value $\mathtt{failed}(z)$ is considered for the purpose of the algorithm as a unary tuple):

$$
\begin{array}{ll}
\mathtt{U}(x,y,\sigma,B) = \langle \sigma \wedge x = y, B \cup \{\{x,y\}\}\rangle & \text{if } \{x,y\} \notin B \wedge \sigma \models x = \bot \wedge y = v \wedge \neg\mathtt{rread}(x) \\
\mathtt{U}(x,y,\sigma,B) = \langle \sigma \wedge x = y, B \cup \{\{x,y\}\}\rangle & \text{if } \{x,y\} \notin B \wedge \sigma \models x = v \wedge y = \bot \wedge \neg\mathtt{rread}(y) \\
\mathtt{U}(x,y,\sigma,B) = \langle \sigma \wedge x = y, B \cup \{\{x,y\}\}\rangle & \text{if } \{x,y\} \notin B \wedge \sigma \models x = \bot \wedge y = \bot \wedge \neg\mathtt{rread}(x) \wedge \neg\mathtt{rread}(y) \\
\mathtt{U}(x,y,\sigma,B) = \langle \sigma \wedge x = y \wedge \mathtt{read}(y), B \cup \{\{x,y\}\}\rangle & \text{if } \{x,y\} \notin B \wedge \sigma \models x = \bot \wedge y = \bot \wedge \mathtt{rread}(x) \wedge \neg\mathtt{rread}(y) \\
\mathtt{U}(x,y,\sigma,B) = \langle \sigma \wedge x = y \wedge \mathtt{read}(x), B \cup \{\{x,y\}\}\rangle & \text{if } \{x,y\} \notin B \wedge \sigma \models x = \bot \wedge y = \bot \wedge \neg\mathtt{rread}(x) \wedge \mathtt{rread}(y) \\
\mathtt{U}(x,y,\sigma,B) = \langle \sigma, B \cup \{\{x,y\}\}\rangle & \text{if } \{x,y\} \notin B \wedge \sigma \models x = u \wedge y = u' \ \wedge\ u = u' \\
\mathtt{U}(x,y,\sigma,B) = \langle \sigma \wedge \sigma', B \cup B'\rangle & \text{if } \{x,y\} \notin B \wedge \sigma \models x = f(a_1 : x_1 \ldots a_n : x_n) \wedge y = f(a_1 : y_1 \ldots a_n : y_n) \\
 & \text{with } \quad \sigma' = \bigwedge_{i=1}^{n} \mathtt{fst}(U_i) \quad B' = \bigcup_{i=1}^{n} \mathtt{snd}(U_i) \quad U_i = \mathtt{U}(x_i, y_i, \sigma, B \cup \{\{x,y\}\}) \\
\mathtt{U}(x,y,\sigma,B) = \langle \sigma, B\rangle & \text{if } \{x,y\} \in B \\
\mathtt{U}(x,y,\sigma,B) = \langle \bot, B\rangle & \text{otherwise}
\end{array}
$$

Subkells. The assertion $\mathtt{subk}_\sigma(\zeta, \{\zeta_1, \ldots, \zeta_n\})$ indicates that the set $\{\zeta_1, \ldots, \zeta_n\}$ corresponds to the set of all *active* subkells of kell ζ, in store σ. The assertion $\mathtt{subth}_\sigma(\zeta, \{\zeta_1, \ldots, \zeta_n\})$ indicates that the set $\{\zeta_1, \ldots, \zeta_n\}$ corresponds to the set of all threads executing in *active* subkells of kell ζ, in store σ.

The predicates subth and subk are defined as follows:

$$
\begin{aligned}
\mathtt{child}_\sigma(\xi, \eta) &\equiv \sigma \models \xi : \mathtt{kell}(\varpi, w) \wedge \eta : \mathtt{kell}(\pi, z) \wedge z = \bot \wedge \mathtt{in}(\xi, \eta) \\
\mathtt{desc}_\sigma(\xi, \eta) &\equiv \mathtt{child}_\sigma(\xi, \eta) \ \vee\ \exists \zeta,\ \mathtt{desc}_\sigma(\xi, \zeta) \wedge \mathtt{child}_\sigma(\zeta, \eta) \\
\mathtt{subk}_\sigma(\xi, \{\xi_1, \ldots, \xi_n\}) &\equiv \{\xi_1, \ldots, \xi_n\} = \{\xi \mid \mathtt{desc}_\sigma(\xi, \eta)\} \\
\mathtt{subth}_\sigma(\xi, \{\zeta_1, \ldots, \zeta_n\}) &\equiv \{\zeta_1, \ldots, \zeta_n\} = \{\zeta \mid \exists \eta,\ \mathtt{desc}_\sigma(\xi, \eta) \wedge \sigma \models \mathtt{inth}(\eta, \zeta)\}
\end{aligned}
$$

Gate access. The assertion $\mathtt{access}_\sigma(\gamma, \kappa, \kappa')$ means that gate γ is accessible for communication between the threads in κ and threads in κ'. The predicate access is defined by cases as follows. In the first case, $\mathtt{access}_\sigma(\gamma, \kappa, \kappa)$. In the second case, where $\sigma \models \mathtt{in}(\kappa, \kappa')$ or $\sigma \models \mathtt{in}(\kappa', \kappa)$, we have $\mathtt{access}_\sigma(\gamma, \kappa, \kappa')$, and $\mathtt{access}_\sigma(\gamma, \kappa', \kappa)$. In the third case, let $\kappa_0, \kappa_1, \ldots, \kappa_{n+1}$, $n \geq 1$, be the smallest sequence such that $\kappa_0 = \kappa$, $\kappa_{n+1} = \kappa'$, $\sigma \models \mathtt{in}(\kappa_i, \kappa_{i+1})$ or $\sigma \models \mathtt{in}(\kappa_{i+1}, \kappa_i)$ for all $i \in I = \{0, \ldots, n\}$ (i.e. $\kappa_1, \ldots, \kappa_{n+1}$ is the minimal path from τ to τ' in the kell tree, where we assume a top-level kell named $\top$). Let $\pi_1, \ldots, \pi_{n+1}$ be such that $\sigma \models \kappa_i : \mathtt{kell}(\pi_i, x_i)$. We define $\mathtt{access}_\sigma(\gamma, \kappa, \kappa')$ by:

$$
\mathtt{access}_\sigma(\gamma, \kappa, \kappa') \triangleq \bigvee_{i \in I} \bigwedge_{j \in I \setminus \{i\}} \mathtt{auth}_\sigma(\gamma, \kappa_j, \kappa_{j+1})
$$

$$
\begin{array}{ll}
\mathtt{auth}_\sigma(\gamma, \kappa_i, \kappa_{i+1}) \triangleq \kappa_i \cdot \gamma \in \pi_{i+1} & \text{if } \sigma \models \mathtt{in}(\kappa_{i+1}, \kappa_i) \\
\mathtt{auth}_\sigma(\gamma, \kappa_i, \kappa_{i+1}) \triangleq \kappa_{i+1} \cdot \gamma \in \pi_i & \text{if } \sigma \models \mathtt{in}(\kappa_i, \kappa_{i+1})
\end{array}
$$

The truth value of the assertion $\kappa \cdot \gamma \in \pi$ in store σ is defined inductively by the table below (the truth value of the assertion is that of the table cell predicate, depending on the form of π), where $\mathtt{subg}^r$ is the reflexive closure of the relation subg, and $\mathtt{subg}^*$ is the reflexive and transitive closure of the relation subg:

π	$\mathtt{K}\cdot\mathtt{G}$	$\mathtt{K}\cdot\gamma'$	$\kappa'\cdot\mathtt{G}$	$\{\kappa'\cdot\gamma'\}$	$\{\kappa'\cdot\eta^r\}$	$\{\kappa'\cdot\eta^*\}$	$\pi_1\cup\pi_2$	$\pi_1\setminus\pi_2$
$\kappa\cdot\gamma\in\pi$	**true**	$\gamma=\gamma'$	$\kappa=\kappa'$	$\kappa=\kappa'$ $\wedge\,\gamma=\gamma'$	$\kappa=\kappa'$ $\wedge\,\sigma\models\mathtt{subg}^r(\eta,\gamma)$	$\kappa=\kappa'$ $\wedge\,\sigma\models\mathtt{subg}^*(\eta,\gamma)$	$\kappa\cdot\gamma\in\pi_1$ $\vee\,\kappa\cdot\gamma\in\pi_2$	$\kappa\cdot\gamma\in\pi_1$ $\wedge\,\kappa\cdot\gamma\notin\pi_2$

Grants. Granting access of a gate to a kell is specified using the function `grant`, which is defined below:

$$
\begin{array}{lr}
\mathtt{grant}(\sigma,k,g)\triangleq\kappa\cdot\gamma & \text{if } \sigma\models k=\kappa\wedge\kappa:\mathtt{kell}(...)\wedge g=\gamma\wedge\gamma:\mathtt{gate}\\
\mathtt{grant}(\sigma,k,g)\triangleq\kappa\cdot\gamma^r & \text{if } \sigma\models k=\kappa\wedge\kappa:\mathtt{kell}(...)\wedge g=\gamma\#\mathtt{all}\wedge\gamma:\mathtt{gate}\\
\mathtt{grant}(\sigma,k,g)\triangleq\kappa\cdot\gamma^* & \text{if } \sigma\models k=\kappa\wedge\kappa:\mathtt{kell}(...)\wedge g=\gamma\#\mathtt{allrec}\wedge\gamma:\mathtt{gate}\\
\mathtt{grant}(\sigma,k,g)\triangleq\kappa\cdot\mathtt{G} & \text{if } \sigma\models k=\kappa\wedge\kappa:\mathtt{kell}(...)\wedge g=\mathtt{all}\\
\mathtt{grant}(\sigma,k,g)\triangleq\mathtt{K}\cdot\gamma & \text{if } \sigma\models k=\mathtt{all}\wedge g=\gamma\wedge\gamma:\mathtt{gate}\\
\mathtt{grant}(\sigma,k,g)\triangleq\mathtt{K}\cdot\gamma^r & \text{if } \sigma\models k=\mathtt{all}\wedge g=\gamma\#\mathtt{all}\wedge\gamma:\mathtt{gate}\\
\mathtt{grant}(\sigma,k,g)\triangleq\mathtt{K}\cdot\gamma^* & \text{if } \sigma\models k=\mathtt{all}\wedge g=\gamma\#\mathtt{allrec}\wedge\gamma:\mathtt{gate}\\
\mathtt{grant}(\sigma,k,g)\triangleq\mathtt{K}\cdot\mathtt{G} & \text{if } \sigma\models k=\mathtt{all}\wedge g=\mathtt{all}\\
\mathtt{grant}(\sigma,k,g)\triangleq\emptyset & \text{otherwise}
\end{array}
$$

B Failure rules

We gather in this section the failure rules of the OZ/K operational semantics.

Thread creation failure

Thread creation fails when the thread name parameter is already bound. This is captured by the following rule.

$$[\textsc{ThreadF}]\ \frac{\texttt{\textbf{thread}}\{x\}\ S\ \texttt{\textbf{end}}}{\sigma \models x \neq \bot} \Bigg\| \frac{\texttt{\textbf{raise}}\ \texttt{error}(\texttt{thread}(\texttt{x}))\ \texttt{\textbf{end}}}{\sigma}$$

Read-only variables

$$[\textsc{ReadF}]\ \frac{x =!!y}{\sigma \models x \neq \bot} \Bigg\| \frac{\texttt{\textbf{raise}}\ \texttt{error}(\texttt{read}(x\ y))\ \texttt{\textbf{end}}}{\sigma}$$

Binding

A binding statement $x = v$ fails if variable x is already bound, or is read-only. The following rule captures this.

$$[\textsc{BindVF}]\ \frac{x = v}{\sigma \models x \neq \bot \vee \texttt{rread}(x)} \Bigg\| \frac{\texttt{\textbf{raise}}\ \texttt{error}(\texttt{bindV}(x))\ \texttt{\textbf{end}}}{\sigma}$$

A binding statement $x = y$ fails if both variables x and y are already bound, or both of them are read-only. The following rule captures this.

$$[\textsc{BindXYF}]\ \frac{x = y}{\sigma \models (x \neq \bot \wedge y \neq \bot) \vee (\texttt{rread}(x) \wedge \texttt{rread}(y))} \Bigg\| \frac{\texttt{\textbf{raise}}\ \texttt{error}(\texttt{bindXY}(x\ y))\ \texttt{\textbf{end}}}{\sigma}$$

A binding statement $x = y.z$ fails if x is already bound, y is not a record or a chunk, or if z is not an integer, an atom or a name. The following rule captures this.

$$[\textsc{BindRF}]\ \frac{x = y.z}{\sigma} \Bigg\| \frac{\texttt{\textbf{raise}}\ \texttt{error}(\texttt{bindR}(x\ y\ z))\ \texttt{\textbf{end}}}{\sigma}\ \text{if } C$$

where

$$C \equiv (\sigma \models x \neq \bot) \vee (\sigma \models y \neq l(f_1 : w_1 \ldots f_n : w_n)_{\mathtt{m}}) \vee (\sigma \models z = v \wedge v \notin \texttt{Int} \cup \texttt{Atom} \cup \texttt{Name})$$

Values

The operation `Equal` fails if the thrid parameter is already bound. This is captured by the following rule.

$$[\textsc{EqF}]\ \frac{\{\texttt{Equal}\ \ x\ \ y\ \ r\}}{\sigma \models r \neq \bot} \Bigg\| \frac{\texttt{\textbf{raise}}\ \texttt{error}(\texttt{equal}(r))\texttt{\textbf{end}}}{\sigma}$$

The operation `Status` fails if the first argument is not a thread, or is not a thread of the current kell.

$$[\textsc{StatusF}]\ \frac{\{\texttt{Status}\ \ x\ \ y\}\ |_\kappa}{\sigma} \Bigg\| \frac{\texttt{\textbf{raise}}\ \texttt{error}(\texttt{status}(x\ y))\ \texttt{\textbf{end}}}{\sigma}\ \text{if } C$$

where

$$C \equiv (\sigma \models y \neq \bot) \vee (\sigma \models x \neq \bot \wedge \sigma \not\models \exists\, \xi,\, x = \tau \wedge \tau : \texttt{thread}(w) \wedge \texttt{in}(\kappa, \tau))$$

If statement

An if statement fails if its condition evaluates to a non boolean value. This is captured by the following rule.

$$[\textsc{IfF}]\ \frac{\textbf{if}\ x\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}}{\sigma \models x = v} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{if}(x))\ \textbf{end}}{\sigma}\ \text{ if } v \notin \{\texttt{true}, \texttt{false}\}$$

Names

Name creation fails if its argument is already bound. This is captured by the following rule.

$$[\textsc{NewNameF}]\ \frac{\{\texttt{NewName}\ x\}}{\sigma \models x \neq \bot} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{newName}(x))\ \textbf{end}}{\sigma}$$

Procedure abstraction

Introducing a new procedure fails if the procedure name argument is already bound. This is captured by the following rule.

$$[\textsc{PNewF}]\ \frac{\textbf{proc}\{x\ X_1 \ldots X_n\}\ S\ \textbf{end}}{\sigma \models x \neq \bot} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{pNew}(x))\ \textbf{end}}{\sigma}$$

Calling a procedure fails if the first argument of the call is not a procedure name, or if the number of arguments provided does not match that of the called procedure. This is captured by the following rule.

$$[\textsc{PCallF}]\ \frac{\{x\ x_1 \ldots x_n\}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{pCall}(x\ [x_1 \ldots x_n]))\ \textbf{end}}{\sigma}\ \text{ if } C$$

where

$$C \equiv (\sigma \models x = v \wedge (v \notin \texttt{Name} \vee (v \in \texttt{Name} \wedge \sigma \not\models v : \texttt{proc}\{\$\ \ldots\}\ \texttt{end}))) \vee (\sigma \models x = \xi \wedge \xi : \texttt{P} \wedge \texttt{P.arity} \neq n)$$

Replacing a procedure fails if the first argument is not an existing procedure, or if the replacement closure does not have the same arity as the replaced one. This is captured by the following rule.

$$[\textsc{PRepBF}]\ \frac{\textbf{proc}\{x\ X_1 \ldots X_n\}\ S\ \textbf{end}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{pRep}(x))\ \textbf{end}}{\sigma}\ \text{ if } C$$

where

$$C \equiv (\sigma \models x = v \wedge (v \notin \texttt{Name} \vee (v \in \texttt{Name} \wedge \sigma \not\models v : \texttt{proc}\{\$\ \ldots\}\ \texttt{end}))) \vee (\sigma \models x = \xi \wedge \xi : \texttt{P} \wedge \texttt{P.arity} \neq n)$$

Checking determinacy

Operation `IsDet` fails if its second argument is already bound. This is captured by the following rule.

$$[\textsc{DetF}]\ \frac{\{\texttt{IsDet}\ x\ y\}}{\sigma \models y \neq \bot} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{isDet}(y))\ \textbf{end}}{\sigma}$$

Cells

Cell creation fails if its second argument is already bound. This is captured by the following rule.

$$[\textsc{NCellF}]\ \frac{\{\texttt{NewCell}\ x\ y\ z\}}{\sigma \models y \neq \bot} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{nCell}(y))\ \textbf{end}}{\sigma}$$

Operation `Exchange` fails if its first argument is not a cell, or if its second argument is already bound. This is captured by the following rule.

$$[\textsc{ECellF}]\ \frac{\{\texttt{Exchange}\ x\ y\ z\}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{eCell}(x))\ \textbf{end}}{\sigma}\ \text{if}\ C$$

where

$$C \equiv (\sigma \models x = v \wedge (v \notin \texttt{Name} \vee (v \in \texttt{Name} \wedge \neg\exists w, \sigma \models v : \texttt{cell}(w)))) \vee (\sigma \models y \neq \bot)$$

Failed values

The creation of a failed value fails if is second argument is already bound. This is captured by the following rule.

$$[\textsc{FailF}]\ \frac{\{\texttt{FailedValue}\ x\ y\}}{\sigma \models y \neq \bot} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{failC}(y))\ \textbf{end}}{\sigma}\ \ z \notin \texttt{dom}(\sigma)$$

Strictness check

$$[\textsc{StrictF}]\ \frac{\{\texttt{IsStrict}\ x\ y\}}{\sigma \models y \neq \bot} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{strict}(y))\ \textbf{end}}{\sigma}\ \text{if}\ \texttt{strict}_\sigma(x)$$

Gate abstraction

Creating a gate fails if the argument is already bound. This is captured by the following rule.

$$[\textsc{NewGF}]\ \frac{\{\texttt{NewGate}\ \ x\}}{\sigma \models x \neq \bot} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{newG}(x))\ \textbf{end}}{\sigma}$$

Creating a subordinate gate fails if the first element of the argument pair is not a gate or if the second element is already bound. This is captured by the following rule.

$$[\textsc{NewGSF}]\ \frac{\{\texttt{NewGate}\ \ x\#z\}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{newGS}(x\ y))}{\sigma}\ \text{if}\ C$$

where

$$C \equiv (\sigma \models x = v \wedge \sigma \not\models v : \texttt{gate}) \vee (z \neq \bot)$$

Sending a message fails if the first argument of the `Send` operation is not a gate. This is captured by the following rule.

$$[\textsc{SendF}]\ \frac{\{\texttt{Send}\ g\ x\}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{send}(g))\ \textbf{end}}{\sigma}\ \text{if}\ C$$

where

$$C \equiv (\sigma \models g = v \wedge \sigma \not\models v : \texttt{gate}) \vee ()$$

Receiving a message fails if the first argument of the `Receive` operation is not a gate, or if the second argument is already bound. This is captured by he following rule.

$$[\textsc{ReceiveF}]\ \frac{\{\texttt{Receive}\ g\ x\}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{receive}(g\ x))\ \textbf{end}}{\sigma}\ \text{if}\ C$$

where

$$C \equiv (\sigma \models g = v \wedge \sigma \not\models v : \texttt{gate}) \vee (x \neq \bot)$$

Opening and closing

The `Open` and `Close` operation fail if their arguments are not of the correct type. This is captured by the rules below.

$$[\textsc{OpenF}]\ \frac{\{\texttt{Open}\ k\ g\}\ |_\kappa}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{open}(k\ g))\ \textbf{end}}{\sigma}\ \text{if}\ C$$

where

$$\begin{aligned} C \equiv\ & \sigma \not\models \exists\kappa', \pi, w,\ k = \kappa' \wedge \kappa' : \texttt{kell}(\pi, w) \\ & \vee \sigma \not\models \exists\gamma,\ g = \gamma \wedge \gamma : \texttt{gate} \\ & \vee \sigma \models k = \kappa' \wedge \kappa' : \texttt{kell}(\pi, w) \wedge \neg\texttt{in}(\kappa, \kappa') \end{aligned}$$

$$[\textsc{CloseF}]\ \frac{\{\texttt{Close}\ k\ g\}\ |_\eta}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{close}(k\ g))\ \textbf{end}}{\sigma}\ \text{if}\ C$$

where

$$\begin{aligned} C \equiv\ & \sigma \not\models \exists\kappa', \pi, w,\ k = \kappa' \wedge \kappa' : \texttt{kell}(\pi, w) \\ & \vee \sigma \not\models \exists\gamma,\ g = \gamma \wedge \gamma : \texttt{gate} \\ & \vee \sigma \models k = \kappa' \wedge \kappa' : \texttt{kell}(\pi, w) \wedge \neg\texttt{in}(\kappa, \kappa') \end{aligned}$$

Kell abstraction

Kell creation fails if the kell name argument is already bound. This is captured by the rule below.

$$[\textsc{KNewF}]\ \frac{\textbf{kell}\{y\}\ S\ \textbf{end}}{\sigma \models y \neq \bot} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{kNew}(y))\ \textbf{end}}{\sigma}$$

Kell replacement fails if the kell name argument does not denote a packed subkell of the current kell. This is captured by the rule below.

$$[\textsc{KRepF}]\ \frac{\textbf{kell}\{y\}\ S\ \textbf{end}\ |_\kappa}{\sigma \models \neg\phi} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{kRep}(y))\ \textbf{end}}{\sigma}$$

where

$$\phi \equiv \exists\kappa', w,\ y = \kappa' \wedge w = \texttt{packed} \wedge \texttt{in}(\kappa, \kappa')$$

Packed values

The failure rule for gate replacement is given below.

$$[\textsc{MarkGF}]\ \frac{\{\texttt{Mark}\ z\ \texttt{gate}(x\ y)\ \ p\}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{markG}(z\ x\ y\ p))\ \textbf{end}}{\sigma}\ \text{if}\ \neg C$$

where

$$C \equiv (\sigma \models p = \bot \wedge \exists \kappa, \mathcal{T}, \sigma', \mu,\ z = \texttt{pack}(\kappa, \mathcal{T}, \sigma', \mu)) \wedge \phi$$

$$\phi \equiv \exists \gamma, \gamma',\ \sigma \models x = \gamma \wedge \gamma : \texttt{gate}\ \wedge\ y = \gamma' \wedge \gamma' : \texttt{gate} \wedge \sigma' \models \gamma : \texttt{gate}$$

The failure rule for procedure replacement is given below.

$$[\textsc{MarkPF}]\ \frac{\{\texttt{Mark}\ z\ \texttt{prc}(x\ y)\ \ p\}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{markP}(z\ x\ y\ p))\ \textbf{end}}{\sigma}\ \text{ if } \neg C$$

where

$$C \equiv (\sigma \models p = \bot \wedge \exists \kappa, \mathcal{T}, \sigma', \mu,\ z = \texttt{pack}(\kappa, \mathcal{T}, \sigma', \mu)) \wedge \phi$$

$$\phi \equiv \exists \eta, \zeta, \tilde{X}, S, S'\ x = \eta \wedge \eta : \texttt{proc}\{\$\ \tilde{X}\}S\ \texttt{end} \wedge y = \zeta \wedge \zeta : \texttt{proc}\{\$\ \tilde{X}\}S'\ \texttt{end} \wedge \sigma' \models \eta : \texttt{proc}\{\$\ \tilde{X}\}S\ \texttt{end}$$

Packing

The failure rule for packing is given below.

$$[\textsc{PackF}]\ \frac{\{\texttt{Pack}\ x\ y\}\ |_\kappa \quad \mathcal{T}}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{pack}(x\ y))\ \textbf{end}\ \ \mathcal{T}}{\sigma}\ \text{ if } \neg C$$

where

$$\mathcal{T} \equiv \tau_1 : \mathcal{T}_1\ \ldots\ \tau_n : \mathcal{T}_n$$

$$C \equiv \exists \kappa_0, \pi_0, w_0, \ldots, \kappa_m, \pi_m, w_m\ (\sigma \models \bigwedge_{i=0}^{m} \phi_i \wedge \phi) \wedge \texttt{subth}_\sigma(\kappa_0, \{\tau_1, \ldots, \tau_n\}) \wedge \texttt{subk}_\sigma(\kappa_0, \{\kappa_1, \ldots, \kappa_m\})$$

$$\phi \equiv x = \kappa_0 \wedge y = \bot \wedge \texttt{in}(\kappa, \kappa_0)$$

$$\phi_i \equiv \kappa_i : \texttt{kell}(\pi_i, w_i) \wedge w_i = \bot$$

The failure rule for unpacking is given below.

$$[\textsc{UnpackF}]\ \frac{\{\texttt{Unpack}\ y\ x\}\ |_\kappa}{\sigma} \Bigg\| \frac{\textbf{raise}\ \texttt{error}(\texttt{pack}(y\ x))\ \textbf{end}}{\sigma}\ \text{ if } \neg C$$

where

$$C \equiv \exists \kappa', \mathcal{T}, \sigma', \mu, \pi, z, \pi', z',\ \sigma \models \kappa : \texttt{kell}(\pi, z) \wedge x = \bot \wedge y = \texttt{pack}(\kappa', \mathcal{T}, \sigma', \mu) \wedge \sigma' \models \kappa' : \texttt{kell}(\pi', z')$$

C Proofs

We gather in this section proofs of the properties in Section 5.2. We first have a few auxiliary lemmas.

Lemma 1 *Let σ be a store, S and S' be statements such that $S \equiv S'$, and θ a substitution on variables and names such that $\mathtt{ran}(\theta) \cap \mathtt{v}(S, \sigma) = \emptyset$. Then, $S\theta \equiv S'\theta$.*

Proof: By induction on the derivation of $S \equiv S'$. In the case of rule S.α, $S =_\alpha S'$, then $S\theta =_\alpha S'\theta$, and hence $S\theta \equiv S'\theta$. In the case of rule S.SEQ $S = S_1\ ; S_2$ and $S' = S_1'\ S_2'$, with $S_1 \equiv S_1'$ and $S_2 \equiv S_2'$, then by induction $S_1\theta \equiv S_1'\theta$ and $S_2\theta \equiv S_2'\theta$, and thus $S\theta \equiv S'\theta$. Other cases are similar to that of rule S.SEQ. □

Lemma 2 *Let σ be a store, $\mathcal{T}$ and $\mathcal{T}'$ be tasks such that $\mathcal{T} \equiv \mathcal{T}'$, and θ a substitution on variables and names such that $\mathtt{ran}(\theta) \cap \mathtt{v}(\mathcal{T}, \sigma) = \emptyset$. Then, $\mathcal{T}\theta \equiv \mathcal{T}'\theta$.*

Proof: By induction on the derivation of $\mathcal{T} \equiv \mathcal{T}'$, using Lemma 1 for the case of rule T.THREAD. □

Lemma 3 *Let σ and σ' be stores such that $\sigma \equiv \sigma'$, and θ a substitution on variables and names such that $\mathtt{ran}(\theta) \cap \mathtt{dom}(\sigma) = \emptyset$. Then, $\sigma\theta \equiv \sigma'\theta$.*

Proof: By induction on the derivation of $\sigma \equiv \sigma'$, using Lemma 1 and Lemma 2 for the case of rule E.PACK, and Lemma 1 for the case of rule E.PROC. □

Lemma 4 *Let σ be a store, v, v' be values such that $v \equiv_\sigma v'$, and θ be a substitution on variables and names such that $\mathtt{ran}(\theta) \cap \mathtt{dom}(\sigma) = \emptyset$. Then, $v\theta \equiv_{\sigma\theta} v'\theta$.*

Proof: By induction on the derivation of $v \equiv_\sigma v'$, using Lemma 3 and Lemma 2 for the case of rule EQ.PACK. □

Lemma 5 *Let σ be a store, v, v' be values such that $v \bowtie_\sigma v'$, and θ be a substitution on variables and names such that $\mathtt{ran}(\theta) \cap \mathtt{dom}(\sigma) = \emptyset$. Then, $v\theta \bowtie_{\sigma\theta} v'\theta$.*

Proof: By induction on the derivation of $v \bowtie_\sigma v'$, using Lemma 3 and Lemma 2 for the case of rule DIS.PACKD. □

Lemma 6 *Let σ be a valid store, and let θ be a substitution on variables and names such that $\mathtt{ran}(\theta) \cap \mathtt{dom}(\sigma) = \emptyset$. Then $\sigma\theta$ is a valid store.*

Proof: The proof proceeds by contradiction. Assume that $\sigma\theta$ is invalid. Then one of the properties in the definition of store invalidity must hold. Assume for instance (dealing with other properties is similar) that the first property holds, i.e. $\exists x, v, v',\ v \bowtie_\sigma v' \wedge \sigma \models x = v \wedge x = v'$. Since $\mathtt{ran}(\theta) \cap \mathtt{dom}(\sigma) = \emptyset$, there exists θ' such that $\mathtt{ran}(\theta') \cap \mathtt{dom}(\sigma\theta) = \emptyset$ and $\sigma\theta\theta' = \sigma$. Now $\sigma\theta \models x = v \wedge x = v'$ implies $\sigma\theta \equiv \sigma' \wedge x = v \wedge x = v'$ for some σ', and thus $\sigma \equiv \sigma'\theta' \wedge y = w \wedge y = w'$, where $y = x\theta'$, $w = v\theta'$, and $w' = v'\theta'$. Now, by Lemma 5, $w \bowtie_\sigma w'$, and hence σ is not valid, a contradiction. □

Lemma 7 *Let σ, σ' be stores such that $\sigma \equiv \sigma'$. If σ is valid, then σ' is valid.*

Proof: By induction on the derivation of $\sigma \equiv \sigma'$. For the cases E.PACK, E.PROC, and E.EQUAL, the result is immediate, by definition of store validity. For the case E.ENTAILS, the result is obtained reasoning by contradiction and using E.TRANS. □

Lemma 8 *Let σ_1 and σ_2 be valid stores, and θ be a substitution such that $\mathtt{ran}(\theta) \cap \mathtt{dom}(\sigma_1, \sigma_2) = \emptyset$. Then $\sigma_1 \wedge \sigma_2\theta$ is a valid store.*

Proof: Immediate since $\sigma_2\theta$ is valid by Lemma 6, and $\mathtt{dom}(\sigma_1) \cap \mathtt{dom}(\sigma_2\theta) = \emptyset$. □

Lemma 9 *Let $\langle \sigma, \mathcal{T} \rangle$ execution structure that results from the execution of an* OZ/K *statement. Then σ is valid.*

Proof: By induction on the length of the reduction that leads to $\langle \sigma, \mathcal{T} \rangle$. In the base case, we have $\sigma_0 \equiv \sigma$, where $\sigma = \tau : \mathtt{thread}(w) \wedge w \wedge \mathtt{read}(w) \wedge \mathtt{inth}(\top, \tau)$. Since σ is trivially valid, so is σ_0 by Lemma 7. Assume that $(\sigma_0, \mathcal{T}_0) \rightarrow \ldots \rightarrow (\sigma_n, \mathcal{T}_n) \rightarrow (\sigma, \mathcal{T})$, with σ_j valid for $0 \leq j \leq n$. We reason by induction on the derivation that has been used to obtain $(\sigma_n, \mathcal{T}_n) \rightarrow (\sigma, \mathcal{T})$:

- [PAR] In this case, $\mathcal{T}_n = \mathcal{U}\ \mathcal{V}$, $\mathcal{T} = \mathcal{U}'\ \mathcal{V}$, and $(\sigma_n, \mathcal{U}) \rightarrow (\sigma, \mathcal{U}')$. By induction, we have σ valid, as required.
- [EQUIV] In this case, $\mathcal{T}_n \equiv \mathcal{U}_n$, $\mathcal{T} \equiv \mathcal{U}$, $\sigma_n \equiv \sigma'_n$, $\sigma \equiv \sigma'$, and $(\sigma'_n, \mathcal{U}_n) \rightarrow (\sigma', \mathcal{U})$. By induction, σ' is valid, and hence σ is valid by Lemma 7.
- Failure rules [THREADF] to [UNPACKF]: in these cases, we have $\sigma_n = \sigma$, hence σ is valid, as required.
- [SKIP], [SEQTH], [UNIF], [IFTRUE], [IFFALSE], [CASE], [CASEU], [PCALL], [TRYU], [TRYC], [RAISEW], [RAISE], [WAITN], [FAILW] In these cases, we have $\sigma_n = \sigma$, hence σ is valid, as required.
- [NIL] In this case, $\sigma_n \models x = \perp$, for some x, and $\sigma = \sigma_n \wedge x = \mathtt{terminated}$. Since σ_n is valid, by induction, the only possibility to make σ invalid would be if (1) $\sigma \models x = v \wedge x = v'$, with $v \bowtie_\sigma v'$. But since $\sigma_n \models x = \perp$, we have only $\sigma \models x = \mathtt{terminated}$, and hence property (1) does not hold. Hence σ is valid, as required.
- [NEWTH] In this case, $\sigma_n \models x = \perp \wedge \mathtt{inth}(\kappa, \tau)$, for some x, κ, τ, and $\sigma = \sigma_n \wedge x = \tau' \wedge \tau' : \mathtt{thread}(w) \wedge w \wedge \mathtt{read}(w) \wedge \mathtt{inth}(\kappa, \tau')$, with $\tau', w \notin \mathtt{dom}(\sigma_n)$. By induction, σ_n is valid. For σ to be invalid one of the following properties from the definition of store invalidity must hold: (1) with x, τ', (3) with τ', w, (6) with κ, τ', or (8) with κ, τ'. Now: (1) does not hold since $\sigma_n \models x = \perp$, σ_n valid; (3) does not hold since σ_n valid and $\sigma \models \tau' : \mathtt{thread}(w) \wedge \mathtt{read}(w)$; (6) does not hold since σ_n valid and τ' fresh; (8) does not hold since σ_n valid (and thus $\sigma_n \models \kappa : \mathtt{kell}(\pi, z)$ for some π, z), and $\sigma \models \tau' : \mathtt{thread}(w)$. Hence σ is valid, as required.
- [VAR] Immediate since in this case $\sigma = \sigma_n \wedge x_1 \wedge \ldots \wedge x_n$ with x_i fresh, and σ_n is valid by induction.
- [READ] Immediate since in this case $\sigma = \sigma_n \wedge \mathtt{read}(x, y)$, for some y, σ_n valid by induction, and the added assertion $\mathtt{read}(x, y)$ does not change the properties required for store invalidity.
- [READU] In this case, $\sigma_n = \sigma' \wedge \mathtt{read}(z, y)$, $\sigma = \sigma' \wedge z = y$, and $\sigma_n \models y \neq \perp$, for some y, z. Now σ_n is valid by induction, and the added store assertion $z = y$ cannot invalidate stor validity. Indeed, if σ were invalid, this would be because $\sigma_n \models x = z \wedge x \neq \perp$ for some x (property (1) in the definition of store invalidity). But this is only possible if the rule [BINDXY], and the rule [BINDV] or the rule [BINDR] have been used in the chain of reduction leading to $(\sigma_n, \mathcal{T}_n)$, with statements of the form $x = z$, $x = v$ or $x = r.f$, for some value v, and variables r, f, respectively. Assume that rule [BINDXY] has been used at step j with statement $x = z$, followed by rule [BINDV] at step $k \geq j$ with statement $x = v$. In this case we would have $\sigma_j \models \mathtt{read}(x)$ following the conditions with [BINDXY]. But then, since the assertion $\mathtt{read}(x)$ is not erased by any rule, rule [BINDV] could not be applied at step k in the reduction chain. The other cases are likewise impossible, which leads to a contradiction. Hence σ is valid, as required.
- [UNI] In this case, we have $\sigma = \sigma_n \wedge \mathtt{Unify}(x, y, \sigma_n)$ for some x, y. This case is handled similarly to the case [READU] above, noting that the added store assertions in $\mathtt{Unify}(x, y, \sigma_n)$ do not invalidate store validity.
- [BINDV] In this case, we have $\sigma = \sigma_n \wedge x = v$, $\sigma_n \models x = \perp \wedge \neg \mathtt{rread}(x)$, for some variable x and value v. Now σ_n is valid by induction, and since $\sigma_n \models x = \perp$, the addition of assertion $x = v$ does not make property (1) of store invalidity true. Hence σ is valid, as required.
- [BINDXY] In this case, we have $\sigma = \sigma_n \wedge \sigma'$, where σ' can be any of five possiblities. We handle the first one, the other ones are handled similarly. We have in this case $\sigma' \equiv y = v$ and $\sigma_n \models x = v \wedge y = \perp \wedge \neg \mathtt{rread}(y)$, for some x, y, v. No, since σ_n is calid by induction, and $\sigma_n \models x = v \wedge y = \perp$, the assertion $y = v$ does not make property (1) of store invalidity true. Hence σ is valid, as required.
- [BINDXY] [EQTRUE] [EQFALSE] [STATUS] [NEWNAME] These cases are similar to case [BINDV].
- [PNEW] In this case, we have $\sigma = \sigma_n \wedge x = \xi \wedge \xi : \mathtt{proc}\{\$\ X_1 \ldots X_n\}S\ \mathtt{end}$, $\sigma_n \models x = \perp$, for some x, X_i, S, with $\xi \notin \mathtt{dom}(\sigma_n)$. Since σ_n is valid by induction and $\sigma_n \models x = \perp$, the addition of assertion $x = \xi$ does not make property (1) of store invalidity true. Furthermore, since ξ is fresh and only the assertion $\xi : \mathtt{proc}\{\$\ X_1 \ldots X_n\}S\ \mathtt{end}$ is added, property (2) of store invalidity does not hold either. Hence σ is valid, as required.
- [PREP] In this case, we have $\sigma_n = \sigma' \wedge \xi : \mathtt{Q}$ and $\sigma = \sigma' \wedge \xi : \mathtt{P}$, for some closures $\mathtt{Q}, \mathtt{P}$. Since σ_n is valid by induction, property (2) of store invalidity does not hold for σ_n, nor for σ. Hence, σ is valid, as required.

- [DETTRUE] [DETFALSE] These cases are similar to case [BINDV].
- [NCELL] This case is similar to case [PNEW].
- [ECELL] In this case, we have $\sigma_n = \sigma' \wedge \xi : \texttt{cell}(t)$, $\sigma = \sigma' \wedge \xi : \texttt{cell}(z) \wedge y = t$, $\sigma_n \models y = \bot$, for some y, z, t. By induction, σ_n is valid, hence $\sigma' \wedge \xi : \texttt{cell}(z)$ is valid. And since $\sigma_n \models y = \bot$, the addition of the assertion $y = t$ does not make property (1) of store invalidity true. Hence σ is valid, as required.
- [RAISES] This case is similar to case [BINDV].
- [NEED] [NEEDD] Immediate since σ_n is valid by induction, and $\sigma = \sigma_n \wedge \texttt{need}(x)$ for some x, and the addition of assertion $\texttt{need}(x)$ dot not make any property of store invalidity true.
- [FAILC] [STRICTTRUE] [STRICTFALSE] [NEWG] [NEWGS] [COM] These cases are similar to case [BINDV].
- [OPEN] [CLOSE] In this cases, we have $\sigma_n = \sigma' \wedge \kappa : \texttt{kell}(\pi, x)$, $\sigma = \sigma' \wedge \kappa : \texttt{kell}(\pi', x)$, for some κ, π, π', x. Since σ_n is valid by induction, the change from π to π' does not make any of the properties (2), (4), (6)-(8) of store invalidity true. Hence, σ is valid, as required.
- [NEWKELL] In this case, we have $\sigma = \sigma_n \wedge y = \kappa \wedge \kappa : \texttt{kell}(\emptyset, w) \wedge w \wedge \texttt{read}(w) \wedge \tau : \texttt{thread}(r) \wedge r \wedge \texttt{read}(r) \wedge \texttt{inth}(\kappa, \tau) \wedge \texttt{in}(\kappa', \kappa)$, $\sigma_n \models y = \bot \wedge \kappa' : \texttt{kell}(\pi, z)$, for some $\kappa, \kappa', \tau, y, w, r, \pi, z$, with $\kappa, \tau, w, r \notin \texttt{dom}(\sigma_n)$. Now, σ_n is valid by induction. Since $\sigma_n \models y = \bot$, then the addition of the assertion $y = \kappa$ does not make property (1) of store invalidity true. Likewise, since $\kappa, \tau, w, r \notin \texttt{dom}(\sigma_n)$, property (2) of store invalidity remains false; since $\sigma \models \tau : \texttt{thread}(r) \wedge r \wedge \texttt{read}(r)$, property (3) of store invalidity remains false; since $\sigma \models \kappa : \texttt{kell}(\emptyset, w) \wedge w \wedge \texttt{read}(w)$, property (4) remains false; since $\kappa, \tau \notin \texttt{dom}(\sigma_n)$ and only the assertions $\texttt{inth}(\kappa, \tau) \wedge \texttt{in}(\kappa', \kappa)$ are added, properties (5), (6) and (9) of store invalidity remain false; since $\sigma \models \kappa : \texttt{kell}(\emptyset, w) \wedge \tau : \texttt{thread}(r) \wedge \texttt{inth}(\kappa, \tau)$, property (8) of store invalidity remains false. Hence σ is valid, as required; finally, since $\sigma \models \kappa' : \texttt{kell}(\pi, z) \wedge \kappa : \texttt{kell}(\emptyset, w) \wedge \texttt{in}(\kappa', \kappa)$, property (7) of store invalidity also remains false. Hence σ is valid, as required.
- [KREP] Similar to the case [NEWKELL].
- [MARKG] [MARKP] [PACK] Similar to the case [BINDV].
- [UNPACK] Immediate, since σ valid is a condition for the application of the rule.

□

Lemma 10 *The set of kells in an execution structure that results from the execution of an* OZ/K *statement forms a tree, with root* $\top$.

Proof: By Lemma 9 and the definition of store validity. □

Lemma 11 *Let* $(\sigma, \mathcal{T})$ *and* $(\sigma', \mathcal{T}')$ *be execution structures such that* $\sigma \equiv \sigma'$ *and* $\mathcal{T} \equiv \mathcal{T}'$*. Then* $\texttt{v}(\mathcal{T}, \sigma) = \texttt{v}(\mathcal{T}', \sigma')$.

Proof: We first show by induction on the derivation of the statement $\mathcal{T} \equiv \mathcal{T}'$ that $\texttt{v}(\mathcal{T}, \sigma) = \texttt{v}(\mathcal{T}', \sigma)$. We then show by induction on the derivation of the statement $\sigma \equiv \sigma'$ that $\texttt{v}(\mathcal{T}', \sigma) = \texttt{v}(\mathcal{T}', \sigma')$. □

Proposition 1 *Assume* $(\sigma, \mathcal{T})$*, with* $\mathcal{T} \equiv \mathcal{T}_1\ \mathcal{T}_2\ \mathcal{T}'$*, is an execution structure that result from the execution of an* OZ/K *statement, where* $\mathcal{T}_1$ *belongs to kell* κ_1*,* $\mathcal{T}_2$ *belongs to kell* κ_2*, and* $\kappa_1 \neq \kappa_2$*. If* $\sigma \models x = \bot$*, and* $x \in \texttt{v}(\mathcal{T}_1, \sigma)$*, then* $x \notin \texttt{v}(\mathcal{T}_2, \sigma)$.

Proof: We reason by induction on the length of the reduction to $(\sigma, \mathcal{T})$. We actually prove a stronger property, (P), which is the conjunction of the following properties:

1. if $\sigma \models x = \bot$, and $x \in \texttt{v}(\mathcal{T}_1, \sigma)$, then $x \notin \texttt{v}(\mathcal{T}_2, \sigma)$.
2. if $\sigma \models x = \xi \wedge \xi : \texttt{cell}(t)$, and $x \in \texttt{v}((\mathcal{T}_1, \sigma)$, then there exists no y such that $\sigma \models y = \xi$ and $y \in \texttt{v}(\mathcal{T}_2, \sigma)$.
3. if $\sigma \models x = \tau \wedge \tau : \texttt{thread}(t)$, and $x \in \texttt{v}((\mathcal{T}_1, \sigma)$, then there exists no y such that $\sigma \models y = \tau$ and $y \in \texttt{v}(\mathcal{T}_2, \sigma)$.
4. if $\sigma \models x = \kappa \wedge \kappa : \texttt{kell}(\pi, t)$, and $x \in \texttt{v}((\mathcal{T}_1, \sigma)$, then there exists no y such that $\sigma \models y = \kappa$ and $y \in \texttt{v}(\mathcal{T}_2, \sigma)$.

5. if $\sigma \models x = \xi \wedge \xi : \texttt{proc}\{\$\ \widetilde{X}\}S\ \texttt{end}$, and $x \in \texttt{v}(\mathcal{T}_1, \sigma)$, then if there exists y such that $\sigma \models y = \xi$ and $y \in \texttt{v}(\mathcal{T}_2, \sigma)$, we have $\texttt{strict}_\sigma(S, \emptyset)$.

We first note that we can replace the condition "there exists no y such that $\sigma \models y = \eta$ and $y \in \texttt{v}(\mathcal{T}_2, \sigma)$" in the definition of the property above by the condition "$x \notin \texttt{v}(\mathcal{T}_2, \sigma)$". Indeed, cells, threads, kells, and procedures are objects created through explicit creation operations (given by rules [NCELL], [NEWTH], [PNEW], [NEWKELL], respectively), that bind the fresh name η of the newly created object to a single variable. Now, all relevant binding operations in the language, given by rules [READU], [BINDXY], [BINDR], [UNI], and [COM] proceed by adding bindings of the form $z = z'$ to the store, where z, z' are variables. Thus a simple induction, using rule [E.EQUALT] shows that to obtain $\sigma \models x = \eta \wedge y = \eta$, where x and y are two distinct variables, one must have $\sigma \models x = y$. Thus, if $y \in \texttt{v}(\mathcal{T}_2, \sigma)$, we must have, by definition of variables of a task relative to a store, $x \in \texttt{v}(\mathcal{T}_2, \sigma)$.

The base case is immediate since there only one thread in the initial execution structure. Assume the property holds for n, and let $(\sigma_n, \mathcal{T}_n) \rightarrow (\sigma, \mathcal{T})$. Without loss of generality, we can consider that the derivation of the reduction $(\sigma_n, \mathcal{T}_n) \rightarrow (\sigma, \mathcal{T})$ has been obtained through an application of one of the rules in Section 5.1, except [PAR] and [EQUIV], or one of the rules in Appendix B (base rules); an application of [PAR]; and an application of [EQUIV]. The application of rule [EQUIV] is handled immediately thanks to the remark above and Lemma 11. Hence we can consider without loss of generality that $(\sigma_n, \mathcal{T}_n) \rightarrow (\sigma, \mathcal{T})$ is obtained by an application of a base rule followed by an application of [PAR]. We consider the different base rules in turn.

Failure rules are easily handled since they leave the store unchanged and modify only a single thread. Since (P) holds for $(\sigma_n, \mathcal{T}_n)$ by induction assumption, then (P) holds for $(\sigma, \mathcal{T})$. Base rules that leave the store unchanged and modify only a single thread are handled similarly: they are [SKIP], [SEQTH], [UNIF], [IFTRUE], [IFFALSE], [CASE], [CASEU], [PCALL], [TRYU], [TRYC], [RAISEW], [RAISE], [WAIT], [FAILW]. We now consider the remaining base rules:

- [NIL] We have $\sigma_n \models \tau : \texttt{thread}(x) \wedge x = \bot$, $\mathcal{T}_n = \tau\langle\rangle\ \ \mathcal{U}$, $\sigma = \sigma_n \wedge x = \texttt{terminated}$, $\mathcal{T} = \mathcal{U}$. Since (P) holds of $(\sigma_n, \tau\langle\rangle\ \ \mathcal{U})$ by induction assumption, (P) holds also of $(\sigma_n \wedge x = \texttt{terminated}, \mathcal{U})$ for $\texttt{v}(\tau\langle\rangle, \sigma_n) = \emptyset$. Hence (P) holds of $(\sigma, \mathcal{T})$, as required.
- [NEWTH] We have $\sigma_n \models x = \bot \wedge \texttt{inth}(\kappa, \tau)$, $\mathcal{T}_n = \tau\langle\textbf{thread}\{x\}\ S\ \textbf{end}\ \ T\rangle\ \ \mathcal{U}$, $\sigma = \sigma_n \wedge x = \tau' \wedge \tau' : \texttt{thread}(w) \wedge w \wedge \texttt{read}(w) \wedge \texttt{inth}(\kappa, \tau')$, $\mathcal{T} = \tau : T\ \ \ \tau' : \langle S'\ \ \langle\rangle\rangle\ \ \mathcal{U}$, with w fresh. Now, we have $\texttt{v}(\tau : T\ \ \tau' : \langle S'\ \ \langle\rangle, \sigma\rangle = \texttt{v}(\tau\langle\textbf{thread}\{x\}\ S\ \textbf{end}\ \ T\rangle, \sigma_n) \cup \{w\}$. Since (P) holds of $(\sigma_n, \tau\langle\rangle\ \ \mathcal{U})$ by induction assumption, we need only check whether clause 1 and clause 3 of (P) hold of $(\sigma, \mathcal{T})$. Since $\sigma_n \models x = \bot$, x does not belong to the variables of any thread in $\mathcal{U}$ that is not in kell κ, hence clause 3 of (P) holds $(\sigma, \mathcal{T})$. Also, w is fresh, and is only reachable through x, hence w does not belong to the variables of any thread in $\mathcal{U}$ that is not κ. Hence (P)) holds of $(\sigma, \mathcal{T})$, as required.
- [VAR] We have $\sigma = \sigma_n \wedge x_1 \wedge \ldots \wedge x_n$, $\mathcal{T}_n = \tau : T\ \ \mathcal{U}$, $\mathcal{T} = \tau : T'\ \ \mathcal{U}$, with x_i fresh, and x_i reachable only from $\tau : T'$. Hence, since (P) holds of $(\sigma_n, \mathcal{T}_n)$ by induction, it holds also of $(\sigma, \mathcal{T})$, as required.
- [READ] This case is imilar to [VAR].
- [BINDV] We have $\sigma_n \models x = \bot \wedge \neg\texttt{read}(x)$, $\mathcal{T}_n = \tau\langle x = v\ \ T\rangle\ \ \mathcal{U}$, $\mathcal{T} = \tau : T\ \ \mathcal{U}$, $\sigma = \sigma_n \wedge x = v$. We have $\texttt{v}(\tau : T, \sigma) \subseteq \texttt{v}(\tau\langle x = v\ \ T\rangle, \sigma_n)$, and (P) holds of $(\sigma_n, \mathcal{T}_n)$ by induction. Hence (P) holds of $(\sigma, \mathcal{T})$, as required.
- [BINDXY], [BINDR], [UNI], [EQTRUE], [EQFALSE], [STATUS] These cases are similar to [BINDV].
- [NEWNAME], [PNEW] These cases are similar to [NEWTH].
- [PREP] We have $\sigma_n = \sigma' \wedge \xi : Q$, $\sigma' \models x = \xi$, $\mathcal{T}_n = \tau : T\ \ \mathcal{U}$, $\mathcal{T} = \tau : T'\ \ \mathcal{U}$, $\sigma = \sigma' \wedge \xi : P$, with $\texttt{strict}_{\sigma'}(P, \emptyset)$. By induction, (P) holds of $(\sigma_n, \mathcal{T}_n)$, and in particular clause 5 of (P) in relation with x and ξ. Since $\texttt{strict}_{\sigma'}(P, \emptyset)$, we have $\texttt{strict}_\sigma(P, \emptyset)$, and hence (P) holds of $(\sigma, \mathcal{T})$, as required.
- [DETTRUE], [DETFALSE] Similar to [BINDV].
- [NCELL] Similar to [NEWTH].
- [ECELL], [RAISES] Similar to [BINDV].
- [NEED] We have $\sigma_n \not\models \texttt{need}(x)$, $\sigma = \sigma_n \wedge \texttt{need}(x)$, $\mathcal{T}_n = \mathcal{T}$. Since (P) holds of $(\sigma_n, \mathcal{T}_n)$ by induction, it holds of $(\sigma, \mathcal{T})$, as required.
- [NEEDD] Similar to [NEED].
- [FAILC], [STRICTTRUE], [STRICTFALSE] Similar to [BINDV].
- [NEWG], [NEWGS] Similar to [NEWTH].

- [COM] We have $\sigma_n \models y = \bot \wedge g = \gamma \wedge h = \gamma \wedge \gamma : \texttt{gate} \wedge \texttt{inth}(\kappa, \tau) \wedge \texttt{inth}(\kappa', \tau')$, $\mathcal{T}_n = \tau : T \ \ \tau' : T' \ \ \mathcal{U}$, $T = \langle\{\texttt{Send}\ g\ x\}\ \ U\rangle, T' = \langle\{\texttt{Receive}\ h\ y\}\ \ U'\rangle, \sigma = \sigma_n \wedge y = x, \mathcal{T} = \tau : U \ \ \tau' : U' \ \ \mathcal{U}$, with $\texttt{strict}_{\sigma_n}(x)$. By induction (P) holds of $(\sigma_n, \mathcal{T}_n)$. We thus need only check whether (P) holds with $\mathcal{T}_1 = \tau : U$ and $\mathcal{T}_2 = \tau : U'$. But this is immediate since $\texttt{strict}_{\sigma_n}(x)$. Hence (P) holds of $(\sigma, \mathcal{T})$, as required.
- [OPEN], [CLOSE] Similar to [NEED].
- [NEWKELL] Similar to [NEWTH], thanks to the condition $\texttt{strict}_\sigma(S, \{y\})$.
- [KREP] Similar to [NEWKELL].
- [MARKG], [MARKP] Similar to [BINDV].
- [PACK] We have $\sigma_n \models x = \kappa_0 \wedge y = \bot \wedge \texttt{inth}(\kappa, \tau), \sigma = \sigma_n \wedge y = \texttt{pack}(\kappa_0, \mathcal{U}, \sigma_n, \emptyset), \mathcal{T}_n = \tau\langle\{\texttt{Pack}\ x\ y\}\ \ T\rangle\ \ \mathcal{U}\ \ \mathcal{V}$, $\mathcal{T} = \tau : T \ \ \mathcal{V}$. By the definition of variables of a task relative to a store, we have $\texttt{v}(\tau : U, \sigma) \subseteq \texttt{v}(\tau : T, \sigma_n)$. Since, (P) holds of $(\sigma_n, \mathcal{T}_n)$, we can conclude immediately that it holds for $(\sigma, \mathcal{T})$, as required. Also, note that (P) holds for $(\sigma_n, \mathcal{U})$.
- [UNPACK] We have $\sigma_n \models \kappa : \texttt{kell}(\pi, z) \wedge x = \bot \wedge \texttt{inth}(\kappa, \tau) \wedge y = \texttt{pack}(\kappa_0, \mathcal{U}, \sigma', \mu), \sigma' = \sigma'' \wedge \kappa_0 : \texttt{kell}(\pi', z')$, $\sigma = \sigma_n \wedge \sigma''\theta \wedge \bigwedge_{\kappa' \in \texttt{tkn}_{\sigma'}(\mathcal{U})} \texttt{in}(\kappa, \kappa'\theta) \wedge x = l, \mathcal{T}_n = \tau\langle\{\texttt{Unpack}\ y\ x\}\ \ T\rangle\ \ \mathcal{V}, \mathcal{T} = \tau : T\ \ \mathcal{U}\theta\ \ \mathcal{V}$, where l is the name list, θ is a substitution that replaces κ_0 with κ, and that renames all variables and names appearing in σ' with fresh variables and fresh names, respectively. By induction, we have that (P) holds of $(\sigma_n, \mathcal{T}_n)$ and of $(\sigma', \mathcal{U})$. Since all the names and variables in $\sigma'\theta$ and $\mathcal{U}\theta$ are distinct from those in σ_n and $T, \mathcal{V}$, and since x is bound to a list of pairs of gate names (which are strict values), (P) holds of $(\sigma, \mathcal{T})$, as required.

□

Proposition 2 *Let $(\mathcal{T}\ \ \mathcal{T}_\kappa, \sigma)$ be an execution structure that results from the execution of a* OZ/K *statement, where κ appears at the top level, $\mathcal{T}_\kappa$ is the set of all threads that belong to κ, κ is not referenced in $\mathcal{T}$, there is no thread τ such that $\sigma \models \texttt{inth}(\kappa, \tau)$, and $\sigma \equiv \sigma_0 \wedge \kappa : \texttt{kell}(\emptyset, w)$, for some σ_0, w. The reductions possible from $\langle\sigma, \mathcal{T}\ \ \mathcal{T}_\kappa\rangle$ can only be of one of the following two forms:*

$$\frac{\mathcal{T}\ \ \mathcal{T}_\kappa}{\sigma} \Bigg\| \frac{\mathcal{T}'\ \ \mathcal{T}_\kappa}{\sigma'} \quad or \quad \frac{\mathcal{T}\ \ \mathcal{T}_\kappa}{\sigma} \Bigg\| \frac{\mathcal{T}\ \ \mathcal{T}'_\kappa}{\sigma'}$$

where $\mathcal{T}'_\kappa$ is the set of threads that belong to κ in execution structure $(\mathcal{T}\ \ \mathcal{T}'_\kappa, \sigma')$, and σ' is such that there is no τ such that $\sigma' \models \texttt{inth}(\kappa, \tau)$, and $\sigma' \equiv \sigma'_0 \wedge \kappa : \texttt{kell}(\emptyset, w)$, for some σ'_0.

Proof: Because of the assumption $\sigma \models \kappa : \texttt{kell}(\emptyset, w)$, we have $\texttt{access}_\sigma(\gamma, \kappa_1, \kappa_2) = \texttt{false}$ for κ_i such that κ_i is a descendant kell of κ, and $\kappa_{i\oplus 1}$ is not a descendant of κ. This implies that we cannot apply rule [COM] between a thread in $\mathcal{T}$ and a thread in $\mathcal{T}_\kappa$. Because of the assumption κ not referenced in $\mathcal{T}$, we cannot apply rule [PACK] with κ as the target from any thread in $\mathcal{T}\ \mathcal{T}_\kappa$. Because of the assumption that there is no thread τ such that $\sigma \models \texttt{inth}(\kappa, \tau)$, we cannot apply any of the rules [OPEN] and [CLOSE] from within κ, and hence $\sigma' \models \kappa : \texttt{kell}(\emptyset, w)$. □

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399