

Region analysis and
a π -calculus with groups

Silvano Dal Zilio and Andrew D. Gordon

August 2000
Revision of May 2001

Technical Report
MSR-TR-2000-57

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract

We show that the typed region calculus of Tofte and Talpin can be encoded in a typed π -calculus equipped with name groups and a novel effect analysis. In the region calculus, each boxed value has a statically determined region in which it is stored. Regions are allocated and de-allocated according to a stack discipline, thus improving memory management. The idea of name groups arose in the typed ambient calculus of Cardelli, Ghelli, and Gordon. There, and in our π -calculus, each name has a statically determined group to which it belongs. Groups allow for type-checking of certain mobility properties, as well as effect analyses. Our encoding makes precise the intuitive correspondence between regions and groups. We propose a new formulation of the type preservation property of the region calculus, which avoids Tofte and Talpin's rather elaborate co-inductive formulation. We prove the encoding preserves the static and dynamic semantics of the region calculus. Our proof of the correctness of region de-allocation shows it to be a specific instance of a general garbage collection principle for the π -calculus with effects. We propose new equational laws for *letregion*, analogous to scope mobility laws in the π -calculus, and show them sound in our semantics.

Contents

1	Motivation	1
2	A λ-Calculus with Regions	3
2.1	Syntax	3
2.2	Dynamic Semantics	4
2.3	Static Semantics	6
2.4	Relating the Static and Dynamic Semantics	8
3	A π-Calculus with Groups	10
3.1	Syntax	10
3.2	Dynamic Semantics	12
3.3	Static Semantics	14
3.4	Barbed Congruence	18
4	Encoding Regions as Groups	19
4.1	The Encoding	20
4.2	Two Garbage Collection Theorems	22
5	An Equational Theory	23
6	Extensions	26
6.1	An Extended λ -Calculus	27
6.2	An Extended π -Calculus	30
6.3	An Extended Encoding	34
7	Conclusions	35
	References	37
A	Review of the Untyped π-Calculus	40
A.1	Syntax	40
A.2	Dynamic Semantics	41
A.3	Labelled Transition Semantics	42
A.4	Barbed Congruence	43
B	Proofs	48
B.1	Subject Reduction for the λ -Calculus	49
B.2	Subject Reduction for the π -Calculus	53
B.3	Correctness of Type Erasure	59
B.4	Properties of Barbed Congruence	64
B.5	Garbage Collection for the π -Calculus	67
B.6	Properties of the Encoding	70
B.6.1	Proof of Static Adequacy	71
B.6.2	An Auxiliary Small Step Semantics	76
B.6.3	Proof of Dynamic Adequacy	79

B.6.4	Proof of Garbage Collection for the λ -Calculus	84
B.7	An Equational Theory	85

1 Motivation

This paper reports a new proof of correctness of region-based memory management (Tofte and Talpin 1997), and also proofs of new equational laws for the region calculus.

Tofte and Talpin’s region calculus is a compiler intermediate language that, remarkably, supports an implementation of Standard ML that has no garbage collector, the ML Kit compiler (Birkedal, Tofte, and Vejlstrup 1996). The basic idea of the region calculus is to partition heap memory into a stack of regions. Each boxed value (that is, a heap-allocated value such as a closure or a cons cell) is annotated with the particular region into which it is stored. The construct *letregion* ρ in b manages the allocation and de-allocation of regions. It means: “Allocate a fresh, empty region, denoted by the region variable ρ ; evaluate the expression b ; de-allocate ρ .” A type and effect system for the region calculus guarantees the safety of de-allocating the defunct region as the last step of *letregion*. The allocation and de-allocation of regions obeys a stack discipline determined by the nesting of the *letregion* constructs. A region inference algorithm compiles ML to the region calculus by computing suitable region annotations for boxed values, and inserting *letregion* constructs as necessary. In practice, space leaks, where a particular region grows without bound, are a problem. Still, they can practically always be detected by profiling and eliminated by simple modifications. The ML Kit efficiently executes an impressive range of benchmarks without a garbage collector and without space leaks. Region-based memory management facilitates interoperability with languages like C that have no garbage collector and helps enable realtime applications of functional programming.

Tofte and Talpin’s semantics of the region calculus is a structural operational semantics. A map from region names to their contents represents the heap. A fresh region name is invented on each evaluation of *letregion*. This semantics supports a co-inductive proof of type safety, including the safety of de-allocating the defunct region at the end of each *letregion*. The proof is complex and surprisingly subtle, in part because active regions may contain dangling pointers that refer to de-allocated regions.

The region calculus is a strikingly simple example of a language with type generativity. A language has type generativity when type equivalence is by name (that is, when types with different names but the same structure are not equivalent), and when type names can be generated at run-time. A prominent example is the core of Standard ML (Milner, Tofte, Harper, and MacQueen 1997), whose *datatype* construct generates a fresh algebraic type each time it is evaluated. (The ML module system admits type generativity also, but at link-time rather than run-time.) The region calculus has type generativity because the type of a boxed value includes the name of the region where it lives, and region names are dynamically generated by *letregion*. The semantics of Standard ML accounts operationally for type generativity by inventing a fresh type name on each elaboration of *datatype*. Various researchers have sought more abstract accounts of type generativity (Leroy 1996; Russo 1996).

This paper describes a new semantics for a form of the region calculus, obtained by translation to a typed π -calculus equipped with a novel effect system. The π -calculus (Milner 1999) is a rather parsimonious formalism for describing the essential semantics of concurrent systems. It serves as a foundation for describing a variety of imperative, functional, and object-oriented programming features (Sangiorgi and Walker 2000; Walker 1995), for the design of concurrent programming languages (Fournet and Gonthier 1996; Pierce and Turner 1997), and for the study of security protocols (Abadi and Gordon 1999), as well as other applications. The only data in the π -calculus are atomic names. Names can model a wide variety of identifiers: communication channels, machine addresses, pointers, object references, cryptographic keys, and so on. A new-name construct $(\nu x)P$ generates names dynamically in the standard π -calculus. It means: “Invent a fresh name, denoted by x ; run process P .” One might hope to model region names with π -calculus names but unfortunately typings would not be preserved: a region name may occur in a region-calculus type, but in standard typed π -calculi (Pierce and Sangiorgi 1996), names may not occur in types.

We solve the problem of modelling region names by defining a typed π -calculus equipped with named groups and a new-group construct (Cardelli, Ghelli, and Gordon 2000a). The idea is that each π -calculus name belongs to a group, G . The type of a name now includes its group. A new-group construct $(\nu G)P$ generates groups dynamically. It means: “Invent a fresh group, denoted by G ; run process P .” The basic ideas of the new semantics are that region names are groups, that pointers into a region ρ are names of group ρ , and that given a continuation channel k the continuation-passing semantics of *letregion* ρ in b is simply the process $(\nu \rho)[[b]]k$ where $[[b]]k$ is the semantics of expression b . The semantics of other expressions is much as in earlier π -calculus semantics of λ -calculi (Sangiorgi and Walker 2000). Parallelism allows us to explain a whole functional computation as an assembly of individual processes that represent components such as closures, continuations, and function invocations.

This new semantics for regions makes two main contributions.

- First, we give a new proof of the correctness of memory management in the region calculus. We begin by extending a standard encoding with the equation $[[\textit{letregion } \rho \textit{ in } b]]k = (\nu \rho)[[b]]k$. Then the rather subtle correctness property of de-allocation of defunct regions turns out to be a simple instance of a new abstract principle expressed in the π -calculus. Hence, an advantage of our π -calculus proof is that it is conceptually simpler than a direct proof.
- Second, the semantics provides a more abstract account of type generativity in the region calculus than the standard operational semantics. A specific benefit is that new equational laws for *letregion* are corollaries of its semantics in terms of the new-group construct.

The specific technical results of the paper are:

- A simple proof of type soundness of the region calculus (Theorem 2.1).

- A new semantics of the region calculus in terms of the π -calculus with groups. The translation preserves types and effects (Theorem 4.1) and operational behaviour (Theorem 4.2).
- A new garbage collection principle for the π -calculus (Theorem 4.3) whose corollary (Theorem 4.4) justifies de-allocation of defunct regions in the region calculus.
- A new equational theory for *letregion*, inspired and justified (Theorem 5.2) by the π -calculus model.

We organise the rest of the paper as follows. Section 2 introduces the region calculus. Section 3 describes the π -calculus with groups and effects. Section 4 gives our new π -calculus semantics for regions. Section 5 describes our new equations for manipulating *letregion*. Section 6 considers extensions. Section 7 concludes. Appendix A reviews the untyped π -calculus. Appendix B describes proofs of all properties stated without proof in the main text.

2 A λ -Calculus with Regions

To focus on the encoding of *letregion* with the new-group construct, we work with a simplified version of the region calculus of Tofte and Talpin (1997). Our calculus omits the recursive functions, type polymorphism, and region polymorphism present in Tofte and Talpin’s calculus. Section 6 extends our results to a version of the region calculus of this section extended with recursive functions, finite lists, and region polymorphism. Tofte and Talpin explain that type polymorphism is not essential for their results. Still, we conjecture that our framework could easily accommodate type polymorphism.

2.1 Syntax

Our region calculus is a typed call-by-value λ -calculus equipped with a *letregion* construct and an annotation on each function to indicate its storage region. We assume an infinite set of *names*, ranged over by p, q, x, y, z . For the sake of simplicity, names represent both program variables and memory pointers, and a subset of the names $L = \{\ell_1, \dots, \ell_n\}$ represents literals. The following table defines the syntax of λ -calculus expressions, a or b , as well as an auxiliary notion of boxed value, u or v .

Expressions and Values:

x, y, p, q, f, g	name: variable, pointer, literal
ρ	region variable
$a, b ::=$	expression
x	name
$v \text{ at } \rho$	allocation of v at ρ
$x(y)$	application

$let\ x = a\ in\ b$	sequencing
$letregion\ \rho\ in\ b$	region allocation, de-allocation
$u, v ::=$	boxed value
$\lambda(x:A)b$	function

We shall explain the type A later. In both $let\ x = a\ in\ b$ and $\lambda(x:A)b$, the name x is bound with scope b . Let $fn(a)$ be the set of names that occur free in the expression a . We identify expressions and values up to consistent renaming of bound names. We write $P\{x \leftarrow y\}$ for the outcome of renaming all free occurrences of x in P to the name y . Our syntax is in a reduced form, where an application $x(y)$ is of a name to a name. We can regard a conventional application $b(a)$ as an abbreviation for $let\ f = b\ in\ let\ x = a\ in\ f(x)$, where $f \neq x$ and f is not free in a .

We explain the intended meaning of the syntax by example. The following expression,

$$ex_1 \triangleq \begin{array}{l} letregion\ \rho'\ in \\ \quad let\ f = \lambda(x:Lit)x\ at\ \rho'\ in \\ \quad let\ g = \lambda(y:Lit)f(y)\ at\ \rho\ in\ g(5) \end{array}$$

means: “Allocate a fresh, empty region, and bind it to ρ' ; allocate $\lambda(x:Lit)x$ in region ρ' , and bind the pointer to f ; allocate $\lambda(y:Lit)f(y)$ in region ρ (an already existing region), and bind the pointer to g ; call the function at g with literal argument 5; finally, de-allocate ρ' .” The function call amounts to calling $\lambda(y:Lit)f(y)$ with argument 5. So we call $\lambda(x:Lit)x$ with argument 5, which immediately returns 5. Hence, the final outcome is the answer 5, and a heap containing a region ρ with g pointing to $\lambda(y:Lit)f(y)$. The intermediate region ρ' has gone. Any subsequent invocations of the function $\lambda(y:Lit)f(y)$ would go wrong, since the target of f has been de-allocated. The type and effect system of Section 2.3 guarantees there are no subsequent allocations or invocations on region ρ' , such as invoking $\lambda(y:Lit)f(y)$.

2.2 Dynamic Semantics

Like Tofte and Talpin, we formalize the intuitive semantics via a conventional structural operational semantics. A heap, h , is a map from region names to regions, and a region, r , is a map from pointers (names) to boxed values (function closures). In Tofte and Talpin’s semantics, defunct regions are erased from the heap when they are de-allocated. In our semantics, the heap consists of both live regions and defunct regions. Our semantics maintains a set S containing the region names for the live regions. This is the main difference between the two semantics. Side-conditions on the evaluation rules guarantee that only the live regions in S are accessed during evaluation. Retaining the defunct regions simplifies the proof of subject reduction. Semmelroth and Sabry (1999) adopt a similar technique for the same reason in their semantics of monadic encapsulation.

Regions, Heaps, and Stacks:

$r ::= (p_i \mapsto v_i)_{i \in 1..n}$	region, p_i distinct
$h ::= (\rho_i \mapsto r_i)_{i \in 1..n}$	heap, ρ_i distinct
$S ::= \{\rho_1, \dots, \rho_n\}$	stack of live regions

A region r is a finite map of the form $p_1 \mapsto v_1, \dots, p_n \mapsto v_n$, where the p_i are distinct, which we usually denote by $(p_i \mapsto v_i)_{i \in 1..n}$. An application, $r(p)$, of the map r to p denotes v_i , if p is p_i for some $i \in 1..n$. Otherwise, the application is undefined. The domain, $dom(r)$, of the map r is the set $\{p_1, \dots, p_n\}$. We write \emptyset for the empty map. If $r = (p_i \mapsto v_i)_{i \in 1..n}$, we define the notation $h - p$ to be $p_i \mapsto v_i$ $_{i \in (1..n) - \{j\}}$ if $p = p_j$ for some $j \in 1..n$, and otherwise to be simply r . Then we define the notation $r + (p \mapsto v)$ to mean $(r - p), p \mapsto v$.

We use finite maps to represent regions, but also heaps, and various other structures. The notational conventions defined above for regions apply also to other finite maps, such as heaps. Additionally, we define $dom_2(h)$ to be the set of all pointers defined in h , that is, $\bigcup_{\rho \in dom(h)} dom(h(\rho))$.

The evaluation relation, $S \cdot (a, h) \Downarrow (p, h')$, may be read: in an initial heap h , with live regions S , the expression a evaluates to the name p (a pointer or literal), leaving an updated heap h' , with the same live regions S .

Judgments:

$S \cdot (a, h) \Downarrow (p, h')$	evaluation
-------------------------------------	------------

Evaluation Rules:

(Eval Var)

$$\frac{}{S \cdot (p, h) \Downarrow (p, h)}$$

(Eval Alloc)

$$\frac{\rho \in S \quad p \notin dom_2(h)}{S \cdot (v \text{ at } \rho, h) \Downarrow (p, h + (\rho \mapsto (h(\rho) + (p \mapsto v))))}$$

(Eval Appl)

$$\frac{\rho \in S \quad h(\rho)(p) = \lambda(x:A)b \quad S \cdot (b\{x \leftarrow q\}, h) \Downarrow (p', h')}{S \cdot (p(q), h) \Downarrow (p', h')}$$

(Eval Let)

$$\frac{S \cdot (a, h) \Downarrow (p', h') \quad S \cdot (b\{x \leftarrow p'\}, h') \Downarrow (p'', h'')}{S \cdot (\text{let } x = a \text{ in } b, h) \Downarrow (p'', h'')}$$

(Eval Letregion)

$$\frac{\rho \notin dom(h) \quad S \cup \{\rho\} \cdot (a, h + \rho \mapsto \emptyset) \Downarrow (p', h')}{S \cdot (\text{letregion } \rho \text{ in } a, h) \Downarrow (p', h')}$$

Recall the example expression ex_1 from the previous section. Consider an initial heap $h = \rho \mapsto \emptyset$ and a region stack $S = \{\rho\}$, together representing a heap with a single region ρ that is live but empty. We can derive $S \cdot (ex_1, h) \Downarrow (5, h')$ where $h' = \rho \mapsto (g \mapsto \lambda(y:Lit)f(y)), \rho' \mapsto (f \mapsto \lambda(x:Lit)x)$. Since $\rho \in S$ but $\rho' \notin S$, ρ is live but ρ' is defunct.

2.3 Static Semantics

The static semantics of the region calculus is a simple type and effect system (Gifford and Lucassen 1986; Talpin and Jouvelot 1992; Wadler 1998). The central typing judgment of the static semantics is:

$$E \vdash a : \{\rho_1, \dots, \rho_n\} A$$

which means that in a typing environment E , the expression a may yield a result of type A , while allocating and invoking boxed values stored in regions ρ_1, \dots, ρ_n . The set of regions $\{\rho_1, \dots, \rho_n\}$ is the *effect* of the expression, a bound on the interactions between the expression and the store. For simplicity, we have dropped the distinction between allocations, $put(\rho)$, and invocations, $get(\rho)$, in Tofte and Talpin's effects. This is an inessential simplification; the distinction could easily be added to our work.

An expression type, A , is either *Lit*, a type of literal constants, or $(A \xrightarrow{e} B)$ at ρ , the type of a function stored in region ρ . The effect e is the latent effect: the effect unleashed by calling the function. An environment E has entries for the regions and names currently in scope.

Effects, Types, and Environments:

$e ::= \{\rho_1, \dots, \rho_n\}$	effect
$A, B ::=$	type of expressions
<i>Lit</i>	type of literals
$(A \xrightarrow{e} B)$ at ρ	type of functions stored in ρ
$E ::=$	environment
\emptyset	empty environment
E, ρ	entry for a region ρ
$E, x:A$	entry for a name x

Let $fr(A)$ be the set of region variables occurring in the type A . We define the domain, $dom(E)$, of an environment, E , by the equations $dom(\emptyset) = \emptyset$, $dom(E, \rho) = dom(E) \cup \{\rho\}$, and $dom(E, x:A) = dom(E) \cup \{x\}$.

The following tables present our type and effect system as a collection of typing judgments defined by a set of rules. Tofte and Talpin present their type and effect system in terms of constructing a region-annotated expression from an unannotated expression. Instead, our main judgment simply expresses the type and effect of a single region-annotated expression. Otherwise, our system is essentially the same as Tofte and Talpin's.

Type and Effect Judgments:

$E \vdash \diamond$	good environment
$E \vdash A$	good type
$E \vdash a :^e A$	good expression, with type A and effect e

Type and Effect Rules:

(Env \emptyset)	(Env x) (recall L is the set of literals)	(Env ρ)
$\emptyset \vdash \diamond$	$\frac{E \vdash A \quad x \notin \text{dom}(E) \cup L}{E, x:A \vdash \diamond}$	$\frac{E \vdash \diamond \quad \rho \notin \text{dom}(E)}{E, \rho \vdash \diamond}$
(Type <i>Lit</i>)	(Type \rightarrow)	
$E \vdash \text{Lit}$	$\frac{E \vdash A \quad \rho \cup \{e\} \subseteq \text{dom}(E) \quad E \vdash B}{E \vdash (A \xrightarrow{e} B) \text{ at } \rho}$	
(Exp x)	(Exp ℓ)	
$\frac{E, x:A, E' \vdash \diamond}{E, x:A, E' \vdash x :^\emptyset A}$	$\frac{E \vdash \diamond \quad \ell \in L}{E \vdash \ell :^\emptyset \text{Lit}}$	
(Exp Appl)	(Exp Let)	
$\frac{E \vdash x :^\emptyset (B \xrightarrow{e} A) \text{ at } \rho \quad E \vdash y :^\emptyset B}{E \vdash x(y) :^{\{\rho\} \cup e} A}$	$\frac{E \vdash a :^e A \quad E, x:A \vdash b :^{e'} B}{E \vdash \text{let } x = a \text{ in } b :^{e \cup e'} B}$	
(Exp Letregion)		
$\frac{E, \rho \vdash a :^e A \quad \rho \notin \text{fr}(A)}{E \vdash \text{letregion } \rho \text{ in } a :^{e - \{\rho\}} A}$		
(Exp Fun)		
$\frac{E, x:A \vdash b :^e B \quad e \subseteq e' \quad \{\rho\} \cup e' \subseteq \text{dom}(E)}{E \vdash \lambda(x:A)b \text{ at } \rho :^{\{\rho\}} (A \xrightarrow{e'} B) \text{ at } \rho}$		

The rules for good environments are standard; they assure that all the names and region variables in the environment are distinct, and that the type of each name is good. All the regions in a good type must be declared. The type of a good expression is checked much as in the simply typed λ -calculus. The effect of a good expression is the union of all the regions in which it allocates or from which it invokes a closure. In the rule (Exp Letregion), the condition $\rho \notin \text{fr}(A)$ ensures that no function with a latent effect on the region ρ may be returned. Calling such a function would be unsafe since ρ is de-allocated once the *letregion* terminates. In the rule (Exp Fun), the effect e of the body of a function must be contained in the latent effect e' of the function. For the sake of simplicity we have no rule of effect subsumption, but it would be sound to add it: if $E \vdash a :^e A$ and $e' \subseteq \text{dom}(E)$ then $E \vdash a :^{e \cup e'} A$. In the presence of effect subsumption we could simplify (Exp Fun) by taking $e = e'$.

Recall the expression ex_1 from Section 2.1. We can derive the following judgments:

$$\begin{aligned}
& \rho, \rho' \vdash (\lambda(x:Lit)x) \text{ at } \rho' : \{\rho'\} (Lit \xrightarrow{\varnothing} Lit) \text{ at } \rho' \\
& \rho, \rho', f : (Lit \xrightarrow{\varnothing} Lit) \text{ at } \rho' \\
& \vdash (\lambda(x:Lit)f(x)) \text{ at } \rho : \{\rho\} (Lit \xrightarrow{\{\rho'\}} Lit) \text{ at } \rho \\
& \rho, \rho', f : (Lit \xrightarrow{\varnothing} Lit) \text{ at } \rho', g : (Lit \xrightarrow{\{\rho'\}} Lit) \text{ at } \rho \\
& \vdash g(5) : \{\rho, \rho'\} Lit
\end{aligned}$$

Hence, we can derive $\rho \vdash ex_1 : \{\rho\} Lit$.

For an example of a type error, suppose we replace the application $g(5)$ in ex_1 simply with the identifier g . Then we cannot type-check the *letregion* ρ' construct, because ρ' is free in the type of its body. This is just as well, because otherwise we could invoke a function in a defunct region.

For an example of how a dangling pointer may be passed around harmlessly, but not invoked, consider the following. Let F abbreviate the type $(Lit \xrightarrow{\varnothing} Lit) \text{ at } \rho'$. Let ex_2 be the following expression:

$$\begin{aligned}
ex_2 & \triangleq \text{letregion } \rho' \text{ in} \\
& \quad \text{let } f = \lambda(x:Lit)x \text{ at } \rho' \text{ in} \\
& \quad \text{let } g = \lambda(f:F)5 \text{ at } \rho \text{ in} \\
& \quad \text{let } j = \lambda(z:Lit)g(f) \text{ at } \rho \text{ in } j
\end{aligned}$$

We have $\rho \vdash ex_2 : \{\rho\} (Lit \xrightarrow{\{\rho'\}} Lit) \text{ at } \rho$. If $S = \{\rho\}$ and $h = \rho \mapsto \varnothing$, then $S \cdot (b, h) \Downarrow (j, h')$ where the final heap h' is $\rho \mapsto (g \mapsto \lambda(f:F)5, j \mapsto \lambda(z:Lit)g(f)), \rho' \mapsto (f \mapsto \lambda(x:Lit)x)$. In the final heap, there is a pointer f from the live region ρ to the defunct region ρ' . Whenever j is invoked, this pointer will be passed to g , harmlessly, since g will not invoke it.

2.4 Relating the Static and Dynamic Semantics

To relate the static and dynamic semantics, we need to define when a configuration is well-typed. First, we need notions of region and heap typings. A region typing R tracks the types of boxed values in the region. A heap typing H tracks the region typings of all the regions in a heap. The environment $env(H)$ lists all the regions in H , followed by types for all the pointers in those regions.

Region and Heap Typings:

$R ::= (p_i:A_i)_{i \in 1..n}$	region typing
$H ::= (\rho_i \mapsto R_i)_{i \in 1..n}$	heap typing
$ptr(H) \triangleq R_1, \dots, R_n$	if $H = (\rho_i \mapsto R_i)_{i \in 1..n}$
$env(H) \triangleq dom(H), ptr(H)$	

The next tables describe the judgments and rules defining well-typed regions, heaps, and configurations. The main judgment $H \models S \cdot (a, h) : A$ means that

a configuration $S \cdot (a, h)$ is well-typed: the heap h conforms to H and the expression a returns a result of type A , and its effect is within the live regions S .

Region, Heap, and Configuration Judgments:

$E \vdash r \text{ at } \rho : R$	in E , region r , named ρ , has type R
$H \models \diamond$	the heap typing H is good
$H \models h$	in H , the heap h is good
$H \models S \cdot (a, h) : A$	in H , configuration $S \cdot (a, h)$ returns A

Region, Heap, and Configuration Rules:

(Region Good)	
$E \vdash v_i \text{ at } \rho : \{\rho\} A_i \quad \forall i \in 1..n$	
<hr/>	
$E \vdash (p_i \mapsto v_i)^{i \in 1..n} \text{ at } \rho : (p_i : A_i)^{i \in 1..n}$	
(Heap Typing Good)	
$env(H) \vdash \diamond$	
<hr/>	
$H \models \diamond$	
(Heap Good) (where $dom(H) = dom(h)$)	
$env(H) \vdash h(\rho) \text{ at } \rho : H(\rho) \quad \forall \rho \in dom(H)$	
<hr/>	
$H \models h$	
(Config Good) (where $S \subseteq dom(H)$)	
$env(H) \vdash a :^c A \quad e \cup fr(A) \subseteq S \quad H \models h$	
<hr/>	
$H \models S \cdot (a, h) : A$	

These predicates roughly correspond to the co-inductively defined consistency predicate of Tofte and Talpin. The retention of defunct regions in our semantics allows a simple inductive definition of these predicates, and a routine inductive proof of the subject reduction theorem stated below.

We now present a subject reduction result relating the static and dynamic semantics. Let $H \asymp H'$ if and only if the pointers defined by H and H' are disjoint, that is, $dom_2(H) \cap dom_2(H') = \emptyset$. Assuming that $H \asymp H'$, we write $H + H'$ for the heap consisting of all the regions in either H or H' ; if ρ is in both heaps, $(H + H')(\rho)$ is the concatenation of the two regions $H(\rho)$ and $H'(\rho)$.

Theorem 2.1 *If $H \models S \cdot (a, h) : A$ and $S \cdot (a, h) \Downarrow (p', h')$ there is H' such that $H \asymp H'$ and $H + H' \models S \cdot (p', h') : A$.*

Intuitively, the theorem asserts that evaluation of a well-typed configuration $S \cdot (a, h)$ leads to another well-typed configuration $S \cdot (p', h')$, where H' represents types for the new pointers and regions in h' .

The following proposition shows that well-typed configurations avoid the runtime errors of allocation or invocation of a closure in a defunct region.

Proposition 2.2

- (1) If $H \models S \cdot (v \text{ at } \rho, h) : A$ then $\rho \in S$.
- (2) If $H \models S \cdot (p(q), h) : A$ then there are ρ and v such that $\rho \in S$, $h(\rho)(p) = v$, and v is a function of the form $\lambda(x:B)b$ with $\text{env}(H), x:B \vdash b :^e A$.

Combining Theorem 2.1 and Proposition 2.2 we may conclude that such runtime errors never arise in any intermediate configuration reachable from an initial well-typed configuration. Implicitly, this amounts to asserting the safety of region-based memory management, that defunct regions make no difference to the behaviour of a well-typed configuration. Our π -calculus semantics of regions makes this explicit: we show equationally that direct deletion of defunct regions makes no difference to the semantics of a configuration.

3 A π -Calculus with Groups

In this section, we define a typed π -calculus with groups. In the next, we explain a semantics of our region calculus in this π -calculus. Exactly as in the ambient calculus with groups (Cardelli, Ghelli, and Gordon 2000a), each name x has a type that includes its group G , and groups may be generated dynamically by a new-group construct, $(\nu G)P$. So as to model the type and effect system of the region calculus, we equip our π -calculus with a novel group-based effect system. In other work (Cardelli, Ghelli, and Gordon 2000b), not concerned with the region calculus, we consider a simpler version of this π -calculus, with groups but without an effect system, and show that new-group helps keep names secret, in a certain formal sense.

3.1 Syntax

The following table gives the syntax of processes, P . The syntax depends on a set of atomic names, x, y, z, p, q , and a set of groups, G, H . For convenience, we assume that the sets of names and groups are identical to the sets of names and region names, respectively, of the region calculus. We impose a standard constraint (Fournet and Gonthier 1996; Merro and Sangiorgi 1998), usually known as locality, that received names may be used for output but not for input. This constraint confers a richer equational theory on the π -calculus and is needed for the results of Section 5. Except for the addition of type annotations and the new-group construct, and the locality constraint, the following syntax and semantics are the same as for the polyadic, choice-free, asynchronous π -calculus (Milner 1999).

Expressions and Processes:

x, y, p, q	name: variable, channel
$P, Q, R ::=$	process
$x(y_1:T_1, \dots, y_n:T_n).P$	input (no $y_i \in \text{inp}(P)$)

$\bar{x}\langle y_1, \dots, y_n \rangle$	output
$(\nu G)P$	new-group: group restriction
$(\nu x:T)P$	new-name: name restriction
$P \mid Q$	composition
$!P$	replication
$\mathbf{0}$	inactivity

We explain the set $inp(P)$ below, and the types T, T_1, \dots, T_n in Section 3.3.

In a process $x(y_1:T_1, \dots, y_n:T_n).P$, the names y_1, \dots, y_n are bound; their scope is P (we explain the types T below). In a group restriction $(\nu G)P$, the group G is bound; its scope is P . In a name restriction $(\nu x:T)P$, the name x is bound; its scope is P . We identify processes up to the consistent renaming of bound groups and names. We let $fn(P)$ and $fg(P)$ be the sets of free names and free groups, respectively, of a process P . We write $P\{x \leftarrow y\}$ for the outcome of a capture-avoiding substitution of the name y for each free occurrence of the name x in the process P .

Free names, $fn(P)$, of process P :

$$\begin{aligned}
fn(x(y_1:T_1, \dots, y_n:T_n).P) &\triangleq \{x\} \cup (fn(P) - \{y_1, \dots, y_n\}) \\
fn(\bar{x}\langle y_1, \dots, y_n \rangle) &\triangleq \{x, y_1, \dots, y_n\} \\
fn((\nu G)P) &\triangleq fn(P) \\
fn((\nu x:T)P) &\triangleq fn(P) - \{x\} \\
fn(P \mid Q) &\triangleq fn(P) \cup fn(Q) \\
fn(!P) &\triangleq fn(P) \\
fn(\mathbf{0}) &\triangleq \emptyset
\end{aligned}$$

Free groups, $fg(P)$, of process P :

$$\begin{aligned}
fg(x(y_1:T_1, \dots, y_n:T_n).P) &\triangleq fg(T_1) \cup \dots \cup fg(T_n) \cup fg(P) \\
fg(\bar{x}\langle y_1, \dots, y_n \rangle) &\triangleq \emptyset \\
fg((\nu G)P) &\triangleq fg(P) - \{G\} \\
fg((\nu x:T)P) &\triangleq fg(T) \cup fg(P) \\
fg(P \mid Q) &\triangleq fg(P) \cup fg(Q) \\
fg(!P) &\triangleq fg(P) \\
fg(\mathbf{0}) &\triangleq \emptyset
\end{aligned}$$

The set $inp(P)$ consists of each name x such that an input $x(y_1:T_1, \dots, y_n:T_n).P'$ occurs as a subprocess of P , with x not bound.

Names in input position, $inp(P)$, in process P :

$$\begin{aligned}
inp(x(y_1:T_1, \dots, y_n:T_n).P) &\triangleq \{x\} \cup (inp(P) - \{y_1, \dots, y_n\}) \\
inp(\bar{x}\langle y_1, \dots, y_n \rangle) &\triangleq \emptyset \\
inp((\nu G)P) &\triangleq inp(P)
\end{aligned}$$

$$\begin{aligned}
inp((\nu x:T)P) &\triangleq inp(P) - \{x\} \\
inp(P \mid Q) &\triangleq inp(P) \cup inp(Q) \\
inp(!P) &\triangleq inp(P) \\
inp(\mathbf{0}) &\triangleq \emptyset
\end{aligned}$$

Next, we explain the semantics of the calculus informally, by example. We omit type annotations and groups; we shall explain these later.

A process represents a particular state in a π -calculus computation. A state may reduce to a successor when two subprocesses interact by exchanging a tuple of names on a shared communication channel, itself identified by a name. For example, consider the following process:

$$f(x, k').\overline{k'}\langle x \rangle \mid g(y, k').\overline{f}\langle y, k' \rangle \mid \overline{g}\langle 5, k \rangle$$

This is the parallel composition (denoted by the \mid operator) of two input processes $g(y, k').\overline{f}\langle y, k' \rangle$ and $f(x, k').\overline{k'}\langle x \rangle$, and an output process $\overline{g}\langle 5, k \rangle$. The whole process performs two reductions. The first is to exchange the tuple $\langle 5, k \rangle$ on the channel g . The names 5 and k are bound to the input names y and k , leaving $f(x, k').\overline{k'}\langle x \rangle \mid \overline{f}\langle 5, k \rangle$ as the next state. This state itself may reduce to the final state $\overline{k}\langle 5 \rangle$ via an exchange of $\langle 5, k \rangle$ on the channel f .

The process above illustrates how functions may be encoded as processes. Specifically, it is a simple encoding of the example ex_1 from Section 2.1. The input processes correspond to λ -abstractions at addresses f and g ; the output processes correspond to function applications; the name k is a continuation for the whole expression. The reductions described above represent the semantics of the expression: a short internal computation returning the result 5 on the continuation k .

The following is a more accurate encoding:

$$(\nu f)(\nu g) \overbrace{(!f(x, k').\overline{k'}\langle x \rangle)}^{f \mapsto \lambda(x)x} \mid \overbrace{!g(y, k').\overline{f}\langle y, k' \rangle}^{g \mapsto \lambda(y)f(y)} \mid \overbrace{\overline{g}\langle 5, k \rangle}^{g(5)}$$

A replication $!P$ is like an infinite parallel array of replicas of P ; we replicate the inputs above so that they may be invoked arbitrarily often. A name restriction $(\nu x)P$ invents a fresh name x with scope P ; we restrict the addresses f and g above to indicate that they are dynamically generated, rather than being global constants.

The other π -calculus constructs are group restriction and inactivity. Group restriction $(\nu G)P$ invents a fresh group G with scope P ; it is the analogue of name restriction for groups. Finally, the $\mathbf{0}$ process represents inactivity.

3.2 Dynamic Semantics

We formalize the semantics of our π -calculus using standard techniques. A reduction relation, $P \rightarrow Q$, means that P evolves in one step to Q . It is defined in terms of an auxiliary structural congruence relation, $P \equiv Q$, that identifies processes we never wish to tell apart.

Structural Congruence: $P \equiv Q$

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow x(y_1:T_1, \dots, y_n:T_n).P \equiv x(y_1:T_1, \dots, y_n:T_n).Q$	(Struct Input)
$P \equiv Q \Rightarrow (\nu G)P \equiv (\nu G)Q$	(Struct GRes)
$P \equiv Q \Rightarrow (\nu x:T)P \equiv (\nu x:T)Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \mid \mathbf{0} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$x_1 \neq x_2 \Rightarrow (\nu x_1:T_1)(\nu x_2:T_2)P \equiv (\nu x_2:T_2)(\nu x_1:T_1)P$	(Struct Res Res)
$x \notin fn(P) \Rightarrow (\nu x:T)(P \mid Q) \equiv P \mid (\nu x:T)Q$	(Struct Res Par)
$(\nu G_1)(\nu G_2)P \equiv (\nu G_2)(\nu G_1)P$	(Struct GRes GRes)
$G \notin fg(T) \Rightarrow (\nu G)(\nu x:T)P \equiv (\nu x:T)(\nu G)P$	(Struct GRes Res)
$G \notin fg(P) \Rightarrow (\nu G)(P \mid Q) \equiv P \mid (\nu G)Q$	(Struct GRes Par)

Reduction: $P \rightarrow Q$

$\bar{x}\langle y_1, \dots, y_n \rangle \mid x(z_1:T_1, \dots, z_n:T_n).P \rightarrow P\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\}$	(Red Interact)
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	(Red Par)
$P \rightarrow Q \Rightarrow (\nu G)P \rightarrow (\nu G)Q$	(Red GRes)
$P \rightarrow Q \Rightarrow (\nu x:T)P \rightarrow (\nu x:T)Q$	(Red Res)
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red \equiv)

Groups help to type-check names statically but have no dynamic behaviour; groups are not themselves values. The following proposition demonstrates this precisely; it asserts that the reduction behaviour of a typed process is equivalent to the reduction behaviour of the untyped process obtained by erasing all type and group annotations. (Appendix A reviews the untyped π -calculus.)

Erasing type annotations and group restrictions:

$erase((\nu G)P) \triangleq erase(P)$
$erase((\nu x:T)P) \triangleq (\nu x)erase(P)$
$erase(\mathbf{0}) \triangleq \mathbf{0}$
$erase(P \mid Q) \triangleq erase(P) \mid erase(Q)$
$erase(!P) \triangleq !erase(P)$
$erase(x(y_1:T_1, \dots, y_n:T_n).P) \triangleq x(y_1, \dots, y_n).erase(P)$
$erase(\bar{x}\langle y_1, \dots, y_n \rangle) \triangleq \bar{x}\langle y_1, \dots, y_n \rangle$

Proposition 3.1 (Erasure) *For all typed processes P and Q , if $P \rightarrow Q$ then $\text{erase}(P) \rightarrow \text{erase}(Q)$. If $\text{erase}(P) \rightarrow R$ then there is a typed process Q such that $P \rightarrow Q$ and $R \equiv \text{erase}(Q)$.*

3.3 Static Semantics

The main judgment $E \vdash P : \{G_1, \dots, G_n\}$ of the effect system for the π -calculus means that the process P uses names according to their types and that all its external reads and writes are on channels in groups G_1, \dots, G_n . A channel type takes the form $G[T_1, \dots, T_n] \setminus \mathbf{H}$. This stipulates that the name is in group G and that it is a channel for the exchange of n -tuples of names with types T_1, \dots, T_n . The set of group names \mathbf{H} is the *hidden effect* of the channel. In the common case when $\mathbf{H} = \emptyset$, we abbreviate the type to $G[T_1, \dots, T_n]$.

As examples of groups, in our encoding of the region calculus we have groups Lit and K for literals and continuations, respectively, and each region ρ is a group. Names of type $Lit[]$ are in group Lit and exchange empty tuples, and names of type $K[Lit[]]$ are in group K and exchange names of type $Lit[]$. In our running example, we have $\mathfrak{s} : Lit[]$ and $k : K[Lit[]]$. A pointer to a function in a region ρ is a name in group ρ . In our example, we could have $f : \rho'[Lit[], K[Lit[]]]$ and $g : \rho[Lit[], K[Lit[]]]$.

Given these typings for names, we have $g(y, k').\bar{f}\langle y, k' \rangle : \{\rho, \rho'\}$ because the reads and writes of the process are on the channels g and f whose groups are ρ and ρ' . Similarly, we have $f(x, k').\bar{k}'\langle x \rangle : \{\rho', K\}$ and $\bar{g}\langle \mathfrak{s}, k \rangle : \{\rho\}$. The composition of these three processes has effect $\{\rho, \rho', K\}$, the union of the individual effects.

The idea motivating hidden effects is that an input process listening on a channel may represent a passive resource (for example, a function) that is only invoked if there is an output on the channel. The hidden effect of a channel is an effect that is masked in an input process, but incurred by an output process. In the context of our example, our formal translation makes the following type assignments: $f : \rho'[Lit[], K[Lit[]]] \setminus \{K\}$ and $g : \rho[Lit[], K[Lit[]]] \setminus \{K, \rho'\}$. We then have $f(x, k').\bar{k}'\langle x \rangle : \{\rho'\}$, $g(y, k').\bar{f}\langle y, k' \rangle : \{\rho\}$, and $\bar{g}\langle \mathfrak{s}, k \rangle : \{\rho, \rho', K\}$. The hidden effects are transferred from the function bodies to the process $\bar{g}\langle \mathfrak{s}, k \rangle$ that invokes the functions. This transfer is essential in the proof of our main garbage collection result, Theorem 4.4.

The effect of a replicated or name-restricted process is the same as the original process. For example, abbreviating the types for f and g , we have:

$$(\nu f:\rho')(\nu g:\rho)(!f(x, k').\bar{k}'\langle x \rangle \mid !g(y, k').\bar{f}\langle y, k' \rangle \mid \bar{g}\langle \mathfrak{s}, k \rangle) : \{\rho, \rho', K\}$$

On the other hand, the effect of a group-restriction $(\nu G)P$ is the same as that of P , except that G is deleted. This is because there can be no names free in P of group G ; any names of group G in P must be internally introduced by name-restrictions. Therefore, $(\nu G)P$ has no external reads or writes on channels

of group G . For example,

$$(\nu \rho')(\nu f)(\nu g)(!f(x, k').\overline{k'}\langle x \rangle \mid !g(y, k').\overline{f}\langle y, k' \rangle \mid \overline{g}\langle \mathfrak{s}, k \rangle) : \{\rho, K\}$$

The following tables describe the syntax of types and environments, the judgments and the rules defining our effect system. Let $fg(G[T_1, \dots, T_n] \setminus \mathbf{H}) \triangleq \{G\} \cup fg(T_1) \cup \dots \cup fg(T_n) \cup \mathbf{H}$.

Group sets, types, environments:

$\mathbf{G}, \mathbf{H} ::= \{G_1, \dots, G_n\}$	finite set of name groups
$T ::=$	channel type
$G[T_1, \dots, T_n] \setminus \mathbf{H}$	polyadic channel in group G with hidden effect \mathbf{H}
$E ::=$	environment
\emptyset	empty environment
E, G	entry for a group G
$E, x:T$	entry for a variable x

Judgments:

$E \vdash \diamond$	good environment
$E \vdash T$	good channel type T
$E \vdash x : T$	good name x of channel type T
$E \vdash P : \mathbf{H}$	good process P with effect \mathbf{H}

Good environments:

$(\text{Env } \emptyset)$	$(\text{Env } x)$	$(\text{Env } G)$
$\frac{}{\emptyset \vdash \diamond}$	$\frac{E \vdash T \quad x \notin \text{dom}(E)}{E, x:T \vdash \diamond}$	$\frac{E \vdash \diamond \quad G \notin \text{dom}(E)}{E, G \vdash \diamond}$

Good types:

(Type Chan)
$\frac{E \vdash \diamond \quad \{G\} \cup \mathbf{H} \subseteq \text{dom}(E) \quad E \vdash T_1 \quad \dots \quad E \vdash T_n}{E \vdash G[T_1, \dots, T_n] \setminus \mathbf{H}}$

Good names:

$(\text{Exp } x)$
$\frac{E', x:T, E'' \vdash \diamond}{E', x:T, E'' \vdash x : T}$

Good processes:

$$\begin{array}{c}
\text{(Proc Input)} \\
\frac{E \vdash x : G[T_1, \dots, T_n] \setminus \mathbf{H} \quad E, y_1:T_1, \dots, y_n:T_n \vdash P : \mathbf{G}}{E \vdash x(y_1:T_1, \dots, y_n:T_n).P : \{G\} \cup (\mathbf{G} - \mathbf{H})} \\
\\
\text{(Proc Output)} \\
\frac{E \vdash x : G[T_1, \dots, T_n] \setminus \mathbf{H} \quad E \vdash y_1 : T_1 \quad \dots \quad E \vdash y_n : T_n}{E \vdash \bar{x}\langle y_1, \dots, y_n \rangle : \{G\} \cup \mathbf{H}} \\
\\
\begin{array}{cc}
\text{(Proc GRes)} & \text{(Proc Res)} \\
\frac{E, G \vdash P : \mathbf{H}}{E \vdash (\nu G)P : \mathbf{H} - \{G\}} & \frac{E, x:T \vdash P : \mathbf{H}}{E \vdash (\nu x:T)P : \mathbf{H}}
\end{array} \\
\\
\begin{array}{ccc}
\text{(Proc Par)} & \text{(Proc Repl)} & \text{(Proc Zero)} \\
\frac{E \vdash P : \mathbf{G} \quad E \vdash Q : \mathbf{H}}{E \vdash P \mid Q : \mathbf{G} \cup \mathbf{H}} & \frac{E \vdash P : \mathbf{H}}{E \vdash !P : \mathbf{H}} & \frac{E \vdash \diamond}{E \vdash \mathbf{0} : \emptyset}
\end{array} \\
\\
\text{(Proc Subsum)} \\
\frac{E \vdash P : \mathbf{G} \quad \mathbf{G} \subseteq \mathbf{H} \subseteq \text{dom}(E)}{E \vdash P : \mathbf{H}}
\end{array}$$

The rules for good environments and good channel types ensure that declared names and groups are distinct, and that all the names and groups occurring in a type are declared. The rules for good processes ensure that names are used for input and output according to their types, and compute an effect that includes the groups of all the free names used for input and output.

In the special case when the hidden effect \mathbf{H} is \emptyset , (Proc Input) and (Proc Output) specialise to the following:

$$\begin{array}{c}
\frac{E \vdash x : G[T_1, \dots, T_n] \setminus \emptyset \quad E, y_1:T_1, \dots, y_n:T_n \vdash P : \mathbf{G}}{E \vdash x(y_1:T_1, \dots, y_n:T_n).P : \{G\} \cup \mathbf{G}} \\
\\
\frac{E \vdash x : G[T_1, \dots, T_n] \setminus \emptyset \quad E \vdash y_1 : T_1 \quad \dots \quad E \vdash y_n : T_n}{E \vdash \bar{x}\langle y_1, \dots, y_n \rangle : \{G\}}
\end{array}$$

In this situation, we attribute all the effect \mathbf{G} of the prefixed process P to the input process $x(y_1:T_1, \dots, y_n:T_n).P$. The effect \mathbf{G} of P is entirely excluded from the hidden effect, since $\mathbf{H} = \emptyset$.

A dual special case is when the effect of the prefixed process P is entirely included in the hidden effect \mathbf{H} . In this case, (Proc Input) and (Proc Output) specialise to the following:

$$\frac{E \vdash x : G[T_1, \dots, T_n] \setminus \mathbf{H} \quad E, y_1:T_1, \dots, y_n:T_n \vdash P : \mathbf{H}}{E \vdash x(y_1:T_1, \dots, y_n:T_n).P : \{G\}}$$

$$\frac{E \vdash x : G[T_1, \dots, T_n] \setminus \mathbf{H} \quad E \vdash y_1 : T_1 \quad \dots \quad E \vdash y_n : T_n}{E \vdash \bar{x}(y_1, \dots, y_n) : \{G\} \cup \mathbf{H}}$$

The effect of P is not attributed to the input $x(y_1:T_1, \dots, y_n:T_n).P$ but instead is transferred to any outputs in the same group as x . If there are no such outputs, the process P will remain blocked, so it is safe to discard its effects.

These two special cases of (Proc Input) and (Proc Output) are in fact sufficient for the encoding of the region calculus presented in Section 4.2; we need the first special case for typing channels representing continuations, and the second special case for typing channels representing function pointers. For simplicity, our actual rules (Proc Input) and (Proc Output) combine both special cases; an alternative would be to have two different kinds of channel types corresponding to the two special cases.

The rule (Proc GRes) discards G from the effect of a new-group process $(\nu G)P$, since, in P , there can be no free names of group G (though there may be restricted names of group G). The rule (Proc Subsum) is a rule of effect subsumption. We need this rule to model the effect subsumption in rule (Exp Fun) of the region calculus. The other rules for good processes simply compute the effect of a whole process in terms of the effects of its parts.

We can prove a standard subject reduction result.

Proposition 3.2 *If $E \vdash P : \mathbf{H}$ and $P \rightarrow Q$ then $E \vdash Q : \mathbf{H}$.*

Next, a standard definition of the barbs exhibited by a process formalizes the idea of the external reads and writes through which a process may interact with its environment. Let a *barb*, β , be either a name x or a co-name \bar{x} .

Exhibition of a barb:

(Barb Input)	(Barb Output)	(Barb GRes)
$x(y_1:T_1, \dots, y_n:T_n).P \downarrow x$	$\bar{x}(y_1, \dots, y_n) \downarrow \bar{x}$	$\frac{P \downarrow \beta}{(\nu G)P \downarrow \beta}$
(Barb Res)	(Barb Par)	(Barb \equiv)
$\frac{P \downarrow \beta \quad \beta \notin \{x, \bar{x}\}}{(\nu x:T)P \downarrow \beta}$	$\frac{P \downarrow \beta}{P \mid Q \downarrow \beta}$	$\frac{P \equiv Q \quad Q \downarrow \beta}{P \downarrow \beta}$

The following asserts the soundness of the effect system. The group of any barb of a process is included in its effect.

Proposition 3.3 (Effect Soundness) *If $E \vdash P : \mathbf{H}$ and $P \downarrow \beta$ with $\beta \in \{x, \bar{x}\}$ then there is a type $G[T_1, \dots, T_n] \setminus \mathbf{G}$ such that $E \vdash x : G[T_1, \dots, T_n] \setminus \mathbf{G}$ and $G \in \mathbf{H}$.*

3.4 Barbed Congruence

To state equational properties of our encoding of the region calculus in the π -calculus, we need a notion of operational equivalence. To this end, we use a typed form (Pierce and Sangiorgi 1996) of the barbed congruence of (Milner and Sangiorgi 1992), an equivalence with a uniform definition for a variety of process calculi. What follows is a series of definitions leading up to our definition of barbed congruence.

First, we state a simple predicate for processes well-defined in a specific environment:

- We write $E \vdash P$ to mean there is an effect \mathbf{G} such that $E \vdash P : \mathbf{G}$.

Since we are in a typed calculus, we only wish to ask whether two processes are equivalent when they are well-defined in the same environment. The notion of a relation on typed processes, given next, is that of a family of binary relations on processes, indexed by an environment. Barbed congruence is defined as a relation on typed processes.

- A *relation on typed processes*, \mathcal{S} , is a set of triples (E, P, Q) where E is an environment and P and Q are typed terms such that $E \vdash P$ and $E \vdash Q$. We write $E \vdash P \mathcal{S} Q$ to mean $(E, P, Q) \in \mathcal{S}$.
- A relation on typed processes, \mathcal{S} , is *reflexive* if and only if $E \vdash P \mathcal{S} P$ whenever $E \vdash P$. It is *symmetric* if and only if $E \vdash Q \mathcal{S} P$ whenever $E \vdash P \mathcal{S} Q$. It is *transitive* if and only if $E \vdash P \mathcal{S} R$ whenever $E \vdash P \mathcal{S} Q$ and $E \vdash Q \mathcal{S} R$.
- For any relation on typed processes, \mathcal{S} , let $E \vdash P \equiv \mathcal{S} \equiv Q$ mean there are processes P' and Q' such that $P \equiv P'$, $E \vdash P' \mathcal{S} Q'$, and $Q' \equiv Q$.

Next, as a standard step towards defining barbed congruence, we define an auxiliary relation, barbed bisimilarity. It is defined co-inductively as the greatest barbed bisimulation.

- We write $P \Downarrow \beta$ to mean there is a process P' such that $P \rightarrow^* P'$ and $P' \downarrow \beta$.
- A relation on typed processes, \mathcal{S} , is a *barbed bisimulation* if and only if it is *symmetric* and $E \vdash P \mathcal{S} Q$ implies:
 - (1) If $P \downarrow \bar{x}$ then $Q \downarrow \bar{x}$.
 - (2) If $P \rightarrow P'$ then there is Q' such that $Q \rightarrow^* Q'$ and $E \vdash P' \equiv \mathcal{S} \equiv Q'$.
- *Barbed bisimilarity*, $\dot{\approx}$, is the relation on typed processes such that $E \vdash P \dot{\approx} Q$ if and only if there is a barbed bisimulation \mathcal{S} such that $E \vdash P \mathcal{S} Q$.

By definition, $E \vdash P \dot{\approx} Q$, it follows that the operational behaviours of P and Q are related in that the reductions and the barbs of P are matched by Q ,

and vice versa. On the other hand, barbed bisimilarity, $\dot{\approx}$, is not a congruence relation, that is, it is not preserved by the syntax formers of our calculus. In particular, it is not even closed under parallel composition. To remedy this, we extract a congruence relation, barbed congruence, from barbed bisimilarity as follows.

- A *renaming*, σ , is a substitution $\{x_1 \leftarrow x'_1\} \cdots \{x_n \leftarrow x'_n\}$ of names for names where $n \geq 0$ and the names x_1, \dots, x_n are pairwise distinct. Let $\text{dom}(\sigma) = \{x_1, \dots, x_n\}$ and $\text{ran}(\sigma) = \{x'_1, \dots, x'_n\}$. If $x = x_j$ for some $j \in 1..n$, let $\sigma(x) = x'_j$. Otherwise, if $x \notin \text{dom}(\sigma)$, let $\sigma(x) = x$. A renaming, σ , is an *E-renaming* if and only if for all names x, y , if $\sigma(x) = \sigma(y)$ and $E \vdash x : T$ and $E \vdash y : T'$ then $T = T'$. For any *E-renaming*, σ , the environment $E\sigma$ is defined as follows: $\emptyset\sigma \triangleq \emptyset$; $(E', G)\sigma \triangleq E'\sigma, G$; $(E', x:T)\sigma \triangleq E'\sigma, \sigma(x):T$ if $\sigma(x) \notin \text{dom}(E'\sigma)$, and $E'\sigma$ if not.
- *Barbed congruence*, \approx , is the relation on typed processes such that $E \vdash P \approx Q$ if and only if for all processes R , all *E-renamings* σ and all type environments E' , if $E\sigma, E' \vdash R$ then $E\sigma, E' \vdash P\sigma \mid R \dot{\approx} Q\sigma \mid R$.

The following are basic properties of barbed congruence needed for equational reasoning. It is a congruence relation that is preserved by well-typed renamings, includes structural congruence, and satisfies a weakening principle.

Proposition 3.4

- (1) *Barbed congruence is reflexive, transitive, and symmetric.*
- (2) *Barbed congruence satisfies the congruence properties:*
 - If $E, y_1:T_1, \dots, y_n:T_n \vdash P \approx Q$ then $E \vdash x(y_1:T_1, \dots, y_n:T_n).P \approx x(y_1:T_1, \dots, y_n:T_n).Q$.
 - If $E \vdash P \approx Q$ and $E \vdash R$ then $E \vdash P \mid R \approx Q \mid R$.
 - If $E, x:T \vdash P \approx Q$ then $E \vdash (\nu x:T)P \approx (\nu x:T)Q$.
 - If $E, G \vdash P \approx Q$ then $E \vdash (\nu G)P \approx (\nu G)Q$.
 - If $E \vdash P \approx Q$ then $E \vdash !P \approx !Q$.
- (3) *If $E \vdash P \approx Q$ and σ is an *E-renaming* then $E\sigma \vdash P\sigma \approx Q\sigma$.*
- (4) *If $P \equiv Q$ and $E \vdash P$ then $E \vdash P \approx Q$.*
- (5) *If $E \vdash P \approx Q$ and $E, E' \vdash \diamond$ then $E, E' \vdash P \approx Q$.*

4 Encoding Regions as Groups

This section interprets the region calculus in terms of our π -calculus.

4.1 The Encoding

Most of the ideas of the translation are standard, and have already been illustrated by example. A function value in the heap is represented by a replicated input process, awaiting the argument and a continuation on which to return a result. A function is invoked by sending it an argument and a continuation. Region names and *letregion* ρ are translated to groups and $(\nu\rho)$, respectively.

The remaining construct of our region calculus is sequencing: *let* $x = a$ *in* b . Assuming a continuation k , we translate this to the following:

$$(\nu k')(\llbracket a \rrbracket k' \mid k'(x).\llbracket b \rrbracket k)$$

This invents a fresh, intermediate continuation k' . The process $\llbracket a \rrbracket k'$ evaluates a returning a result on k' . The process $k'(x).\llbracket b \rrbracket k$ blocks until the result x is returned on k' , then evaluates b , returning its result on k .

The following tables interpret the types, environments, expressions, regions, and configurations of the region calculus in the π -calculus. In particular, if $S \cdot (a, h)$ is a configuration, then $\llbracket S \cdot (a, h) \rrbracket k$ is its translation, a process that returns any eventual result on the continuation k . In typing the translation, we assume two global groups: a group, K , of continuations and a group, Lit , of literals. The environment $\llbracket \emptyset \rrbracket$ declares these groups and also a typing $\ell_i:Lit$ for each of the literals ℓ_1, \dots, ℓ_n .

Translating of the region calculus to the π -calculus:

$\llbracket A \rrbracket$	type modelling the type A
$\llbracket E \rrbracket$	environment modelling environment E
$\llbracket a \rrbracket k$	process modelling term a , answer on k
$\llbracket p \mapsto v \rrbracket$	process modelling value v at pointer p
$\llbracket r \rrbracket$	process modelling region r
$\llbracket S \cdot (a, h) \rrbracket k$	process modelling configuration $S \cdot (a, h)$

In the following equations, where necessary to construct type annotations in the π -calculus, we have added type subscripts to the syntax of the region calculus. The notation $\prod_{i \in I} P_i$ for some finite indexing set $I = \{i_1, \dots, i_n\}$ is short for the composition $P_{i_1} \mid \dots \mid P_{i_n} \mid \mathbf{0}$.

Translation rules:

$$\begin{aligned} \llbracket Lit \rrbracket &\triangleq Lit \\ \llbracket (A \xrightarrow{e} B) \text{ at } \rho \rrbracket &\triangleq \rho[\llbracket A \rrbracket, K[\llbracket B \rrbracket]] \setminus (e \cup \{K\}) \\ \llbracket \emptyset \rrbracket &\triangleq K, Lit, \ell_1:Lit, \dots, \ell_n:Lit \\ \llbracket E, \rho \rrbracket &\triangleq \llbracket E \rrbracket, \rho \\ \llbracket E, x:A \rrbracket &\triangleq \llbracket E \rrbracket, x:\llbracket A \rrbracket \\ \llbracket x \rrbracket k &\triangleq \bar{k}(x) \\ \llbracket \text{let } x = a_A \text{ in } b \rrbracket k &\triangleq (\nu k':K[\llbracket A \rrbracket])(\llbracket a \rrbracket k' \mid k'(x:\llbracket A \rrbracket).\llbracket b \rrbracket k) \end{aligned}$$

$$\begin{aligned}
\llbracket p(q) \rrbracket k &\triangleq \bar{p}\langle q, k \rangle \\
\llbracket \text{let region } \rho \text{ in } a \rrbracket k &\triangleq (\nu \rho) \llbracket a \rrbracket k \\
\llbracket (v \text{ at } \rho)_A \rrbracket k &\triangleq (\nu p: \llbracket A \rrbracket) (\llbracket p \mapsto v \rrbracket \mid \bar{k}\langle p \rangle) \\
\llbracket p \mapsto \lambda(x:A) b_B \rrbracket &\triangleq !p(x: \llbracket A \rrbracket, k: K \llbracket \llbracket B \rrbracket \rrbracket) . \llbracket b \rrbracket k \\
\llbracket (p_i \mapsto v_i)^{i \in 1..n} \rrbracket &\triangleq \prod_{i \in 1..n} \llbracket p_i \mapsto v_i \rrbracket \\
\llbracket (\rho_i \mapsto r_i)^{i \in 1..n} \rrbracket &\triangleq \prod_{i \in 1..n} \llbracket r_i \rrbracket \\
\llbracket S \cdot (a, h_H) \rrbracket k &\triangleq (\nu \vec{\rho}_{\text{defunct}}) (\nu \llbracket ptr(H) \rrbracket) (\llbracket a \rrbracket k \mid \llbracket h \rrbracket) \\
&\quad \text{where } \{\vec{\rho}_{\text{defunct}}\} = \text{dom}(H) - S
\end{aligned}$$

The following theorem asserts that the translation preserves the static semantics of the region calculus.

Theorem 4.1 (Static Adequacy)

- (1) If $E \vdash \diamond$ then $\llbracket E \rrbracket \vdash \diamond$.
- (2) If $E \vdash A$ then $\llbracket E \rrbracket \vdash \llbracket A \rrbracket$.
- (3) If $E \vdash a :^e A$ and $k \notin \text{dom}(\llbracket E \rrbracket)$ then

$$\llbracket E \rrbracket, k: K \llbracket \llbracket A \rrbracket \rrbracket \vdash \llbracket a \rrbracket k : e \cup \{K\}$$

- (4) If $H \models h$ and $\rho \in \text{dom}(H)$ then

$$\llbracket \text{env}(H) \rrbracket \vdash \llbracket h(\rho) \rrbracket : \{\rho\}$$

- (5) If $H \models S \cdot (a, h) : A$ and $k \notin \llbracket \text{env}(H) \rrbracket$ then

$$\llbracket \text{env}(H) \rrbracket, k: K \llbracket \llbracket A \rrbracket \rrbracket \vdash \llbracket a \rrbracket k \mid \llbracket h \rrbracket : \text{dom}(H) \cup \{K\}$$

and also

$$\llbracket \emptyset \rrbracket, S, k: K \llbracket \llbracket A \rrbracket \rrbracket \vdash \llbracket S \cdot (a, h) \rrbracket k : S \cup \{K\}$$

Next we state that the translation preserves the dynamic semantics. Our theorem states that if one region calculus configuration evaluates to another, their π -calculus interpretations are barbed congruent:

Theorem 4.2 (Dynamic Adequacy) *If $H \models S \cdot (a, h) : A$ and $S \cdot (a, h) \Downarrow (p', h')$ then there is H' such that $H \asymp H'$ and $H + H' \models S \cdot (p', h') : A$ and for all $k \notin \text{dom}_2(H + H') \cup L$, $\llbracket \emptyset \rrbracket, S, k: K \llbracket \llbracket A \rrbracket \rrbracket \vdash \llbracket S \cdot (a, h) \rrbracket k \approx \llbracket S \cdot (p', h') \rrbracket k$.*

Recall the evaluations of the examples ex_1 and ex_2 given previously. From Theorem 4.2 we obtain the following equations (in which we abbreviate environments and types for the sake of clarity):

$$\begin{aligned}
\llbracket \{\rho\} \cdot (ex_1, h) \rrbracket k &\approx \\
&(\nu \rho') (\nu f: \rho') (\nu g: \rho) (\llbracket f \mapsto \lambda(x)x \rrbracket \mid \llbracket g \mapsto \lambda(y)f(y) \rrbracket \mid \bar{k}\langle 5 \rangle)
\end{aligned}$$

$$\begin{aligned}
\llbracket \{\rho\} \cdot (ex_2, h) \rrbracket k &\approx \\
&(\nu \rho') (\nu f: \rho') (\nu g: \rho) (\nu j: \rho) \\
&(\llbracket f \mapsto \lambda(x)x \rrbracket \mid \llbracket g \mapsto \lambda(f)5 \rrbracket \mid \llbracket j \mapsto \lambda(z)g(f) \rrbracket \mid \bar{k}\langle j \rangle)
\end{aligned}$$

4.2 Two Garbage Collection Theorems

We present a general π -calculus theorem that has as a corollary a theorem asserting that defunct regions may be deleted without affecting the meaning of a configuration.

Suppose there are processes P and R such R has effect $\{G\}$ but G is not in the effect of P . So R only interacts on names in group G , but P never interacts on names in group G , and therefore there can be no interaction between P and R . Moreover, if P and R are the only sources of inputs or outputs in the scope of G , then R has no external interactions, and therefore makes no difference to the behaviour of the whole process. The following makes this idea precise equationally. We state the theorem in terms of the notation $(\nu E)P$ defined by the equations: $(\nu \emptyset)P \triangleq P$, $(\nu E, x:T)P \triangleq (\nu E)(\nu x:T)P$, and $(\nu E, G)P \triangleq (\nu E)(\nu G)P$. The proof proceeds by constructing a suitable bisimulation relation.

Theorem 4.3 *If $E, G, E' \vdash P : \mathbf{H}$ and $E, G, E' \vdash R : \{G\}$ with $G \notin \mathbf{H}$, then $E \vdash (\nu G)(\nu E')(P \mid R) \approx (\nu G)(\nu E')P$.*

Now, by applying this theorem, we can delete the defunct region ρ' from our two examples. We obtain:

$$\begin{aligned} & (\nu \rho')(\nu f:\rho')(\nu g:\rho)(\llbracket f \mapsto \lambda(x)x \rrbracket \mid \llbracket g \mapsto \lambda(y)f(y) \rrbracket \mid \bar{k}\langle 5 \rangle) \\ & \approx (\nu \rho')(\nu f:\rho')(\nu g:\rho)(\llbracket g \mapsto \lambda(y)f(y) \rrbracket \mid \bar{k}\langle 5 \rangle) \\ & (\nu \rho')(\nu f:\rho')(\nu g:\rho)(\nu j:\rho) \\ & \quad (\llbracket f \mapsto \lambda(x)x \rrbracket \mid \llbracket g \mapsto \lambda(f)5 \rrbracket \mid \llbracket j \mapsto \lambda(z)g(f) \rrbracket \mid \bar{k}\langle j \rangle) \\ & \approx (\nu \rho')(\nu f:\rho')(\nu g:\rho)(\nu j:\rho) \\ & \quad (\llbracket g \mapsto \lambda(f)5 \rrbracket \mid \llbracket j \mapsto \lambda(z)g(f) \rrbracket \mid \bar{k}\langle j \rangle) \end{aligned}$$

The first equation illustrates the need for hidden effects. The hidden effect of g is $\{K, \rho'\}$, and so the overall effect of the process $\llbracket g \mapsto \lambda(y)f(y) \rrbracket \mid \bar{k}\langle 5 \rangle$ is simply $\{\rho, K\}$. This effect does not contain ρ' and so the theorem justifies deletion of the process $\llbracket f \mapsto \lambda(x)x \rrbracket$, whose effect is $\{\rho'\}$. In an effect system for the π -calculus without hidden effects, the effect of $\llbracket g \mapsto \lambda(y)f(y) \rrbracket \mid \bar{k}\langle 5 \rangle$ would include ρ' , and so the theorem would not be applicable.

A standard garbage collection principle in the π -calculus is that if f does not occur free in P , then $(\nu f)(!f(x, k).R \mid P) \approx P$. One might hope that this principle alone would justify de-allocation of defunct regions. But neither of our example equations is justified by this principle; in both cases, the name f occurs in the remainder of the process. We need an effect system to determine that f is not actually invoked by the remainder of the process.

The two equations displayed above are instances of our final theorem, a corollary of Theorem 4.3. It asserts that deleting defunct regions makes no difference to the behaviour of a configuration:

Theorem 4.4 *Suppose $H \models S \cdot (a, h) : A$ and $k \notin \text{dom}_2(H) \cup L$. Let $\vec{\rho}_{\text{defunct}}$ be the sequence of groups in $\text{dom}(H) - S$. Then:*

$$\llbracket \emptyset \rrbracket, S, k:K \llbracket [A] \rrbracket \vdash \llbracket S \cdot (a, h) \rrbracket k \approx (\nu \vec{\rho}_{\text{defunct}})(\nu \llbracket \text{ptr}(H) \rrbracket)(\llbracket [a] \rrbracket k \mid \prod_{\rho \in S} \llbracket H(\rho) \rrbracket)$$

5 An Equational Theory

The new-group construct enjoys various equational properties, such as our laws of structural congruence. On the other hand, equational properties of *letregion* do not appear to have been previously studied. This section proposes an equational theory for the region calculus, including equations for *letregion* inspired in part by equations for new-group. We prove that the equational theory is sound with respect to the semantics of the previous section. The equational transformations of Benton and Kennedy (1999) for their ML intermediate language (without regions) appear to be the only prior work on an explicit equational theory for a typed calculus with effects.

In the following, recall that the conventional syntax for application, $b(a)$, where either b or a is not a name, abbreviates $b(a) \triangleq \text{let } f = b \text{ in let } x = a \text{ in } f(x)$ where $f \notin \{x\} \cup \text{fn}(a)$. Given this abbreviation, we can define in the standard way the substitution $b\{x \leftarrow a\}$ to be the expression obtained by replacing each free occurrence of x in b with the expression a .

Substitution of a term for a name:

$$\begin{array}{l}
 x\{z \leftarrow c\} \triangleq \begin{cases} c & \text{if } x = z \\ x & \text{otherwise} \end{cases} \\
 x(y)\{z \leftarrow c\} \triangleq x\{z \leftarrow c\}(y\{z \leftarrow c\}) \\
 (\text{let } x = a \text{ in } b)\{z \leftarrow c\} \triangleq \text{let } x = a\{z \leftarrow c\} \text{ in } (b\{z \leftarrow c\}) & \text{for } x \notin \{z\} \cup \text{fn}(c) \\
 (\lambda(x:A)b)\{z \leftarrow c\} \triangleq \lambda(x:A)(b\{z \leftarrow c\}) & \text{for } x \notin \{z\} \cup \text{fn}(c)
 \end{array}$$

The rules in the following tables inductively define the judgment $E \vdash a \leftrightarrow b : A$ intended to mean that the terms a and b have the same type, A , and equivalent observable behaviour, although they may have different effects.

The first set of rules is essentially the call-by-value λ -calculus (Plotkin 1975). As usual in an equational theory for call-by-value, we restrict the argument a in the rule (Eq Fun β) to be fully evaluated, either a name, x , or an allocation, $\lambda(x:A)b$ at ρ . This restriction is actually unnecessary for the present calculus, since there are no non-terminating computations, but we include it so that the equational theory remains valid when we extend our calculus with recursion. In rule (Eq Fun β), we also ask for $(\lambda(x:A)b \text{ at } \rho)(a)$ and $b\{x \leftarrow a\}$ to share the same type, B . This is because the type of $b\{x \leftarrow a\}$ can sometimes differ from the type of $(\lambda(x:A)b \text{ at } \rho)(a)$.

Equational Theory: The Call-by-Value λ -Calculus

$$\begin{array}{c}
 \begin{array}{ccc}
 \text{(Eq Refl)} & \text{(Eq Symm)} & \text{(Eq Trans)} \\
 \frac{E \vdash a :^e A}{E \vdash a \leftrightarrow a : A} & \frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A} & \frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}
 \end{array}
 \end{array}$$

(Eq Fun)

$$\frac{E \vdash (A \xrightarrow{e} B) \text{ at } \rho \quad E, x:A \vdash b_1 \leftrightarrow b_2 : B \quad E, x:A \vdash b_i :^{e_i} B \quad e_i \subseteq e \quad \forall i \in 1..2}{E \vdash (\lambda(x:A)b_1) \text{ at } \rho \leftrightarrow (\lambda(x:A)b_2) \text{ at } \rho : (A \xrightarrow{e} B) \text{ at } \rho}$$

(Eq Fun β) (where a is a name or an allocation)

$$\frac{E \vdash a :^{e_1} A \quad E, x:A \vdash b :^{e_2} B \quad E \vdash b\{x \leftarrow a\} :^{e_3} B \quad \rho \in \text{dom}(E)}{E \vdash (\lambda(x:A)b \text{ at } \rho)(a) \leftrightarrow b\{x \leftarrow a\} : B}$$

Next, we have rules for *let*, inspired by the computational λ -calculus (Moggi 1989).

Equational Theory: *let*

(Eq Let)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E, x:A \vdash b \leftrightarrow b' : B}{E \vdash \text{let } x = a \text{ in } b \leftrightarrow \text{let } x = a' \text{ in } b' : B}$$

(Eq Let Assoc)

$$\frac{E \vdash a :^{e_1} A \quad E, x:A \vdash b :^{e_2} B \quad E, y:B \vdash c :^{e_3} C}{E \vdash \text{let } x = a \text{ in } (\text{let } y = b \text{ in } c) \leftrightarrow \text{let } y = (\text{let } x = a \text{ in } b) \text{ in } c : C}$$

(Eq Let β) (where a is a name or an allocation)

$$\frac{E \vdash a :^{e_1} A \quad E, x:A \vdash b :^{e_2} B \quad E \vdash b\{x \leftarrow a\} :^{e_3} B}{E \vdash \text{let } x = a \text{ in } b \leftrightarrow b\{x \leftarrow a\} : B}$$

Finally, here are the new rules for *letregion*. For the sake of brevity, we write $(\nu\rho)a$ as a shorthand for *letregion* ρ in a .

Equational Theory: *letregion*

(Eq Letregion)

$$\frac{E, \rho \vdash a \leftrightarrow a' : A \quad \rho \notin \text{fr}(A)}{E \vdash (\nu\rho)a \leftrightarrow (\nu\rho)a' : A}$$

(Eq Drop)

$$\frac{E \vdash a :^e A \quad \rho \notin \text{dom}(E)}{E \vdash (\nu\rho)a \leftrightarrow a : A}$$

(Eq Swap)

$$\frac{E, \rho, \rho' \vdash a :^e A \quad \{\rho, \rho'\} \cap \text{fr}(A) = \emptyset}{E \vdash (\nu\rho)(\nu\rho')a \leftrightarrow (\nu\rho')(\nu\rho)a : A}$$

(Eq Letregion Let)

$$\frac{E, \rho \vdash a :^{e_1} A \quad E, x:A, \rho \vdash b :^{e_2} B \quad \rho \notin \text{fr}(A) \cup \text{fr}(B)}{E \vdash (\nu\rho)\text{let } x = a \text{ in } b \leftrightarrow \text{let } x = (\nu\rho)a \text{ in } (\nu\rho)b : B}$$

The rule (Eq Letregion) is a congruence rule. The rule (Eq Swap) allows region scopes to be re-ordered. The rule (Eq Drop) allows unused region scopes

to be discarded; we need the condition $\rho \notin \text{dom}(E)$, rather than the weaker condition $\rho \notin e \cup \text{fr}(A)$, to ensure that both $(\nu\rho)a$ and a are well-typed. The rule (Eq Letregion Let) allows a single region to be broken into two.

The following are derivable rules. The first is an instance of (Eq Drop). The second follows from the first, (Eq Letregion Let), and (Eq Trans).

$$\frac{\text{(Eq Appl } x(y)) \quad E \vdash x :^{\varnothing} (B \xrightarrow{e} A) \text{ at } \rho' \quad E \vdash y :^{\varnothing} B \quad \rho \notin \text{dom}(E)}{E \vdash (\nu\rho)x(y) \leftrightarrow x(y) : B}$$

$$\frac{\text{(Eq Appl) (where } \rho \notin \text{fr}((A \xrightarrow{e_2} B) \text{ at } \rho')) \quad E, \rho \vdash b :^{e_1} (A \xrightarrow{e_2} B) \text{ at } \rho' \quad E, \rho \vdash a :^{e_3} A}{E \vdash (\nu\rho)(b(a)) \leftrightarrow ((\nu\rho)b)((\nu\rho)a) : B}$$

Other examples of derivable rules are:

$$\frac{E \vdash x :^{\varnothing} A \xrightarrow{e} B \text{ at } \rho \quad E \vdash a :^{e'} A \quad y \notin \text{dom}(E)}{E \vdash x(a) \leftrightarrow \text{let } y = a \text{ in } x(y) : B}$$

$$\frac{E \vdash a :^{e'} A \xrightarrow{e} B \text{ at } \rho \quad E \vdash x :^{\varnothing} A \quad y \notin \text{dom}(E)}{E \vdash a(x) \leftrightarrow \text{let } y = a \text{ in } y(x) : B}$$

$$\frac{E, x:A \vdash b :^{e_1} B \quad E \vdash a :^{e_2} A \quad f \notin \text{dom}(E) \quad \rho \in \text{dom}(E)}{E \vdash \text{let } f = \lambda(x:A)b \text{ at } \rho \text{ in } f(a) \leftrightarrow b\{x \leftarrow a\} : B}$$

$$\frac{E \vdash a :^{e_1} A \quad E, x:A \vdash b :^{e_2} B \quad \rho \in \text{dom}(E)}{E \vdash \text{let } x = a \text{ in } b \leftrightarrow (\lambda(x:A)b \text{ at } \rho)(a) : B}$$

$$\frac{E \vdash a :^{e_1} A \quad E \vdash b :^{e_2} B \quad x \notin \text{dom}(E)}{E \vdash \text{let } x = a \text{ in } b \leftrightarrow b : B}$$

$$\frac{E \vdash a :^{e_1} A \quad E, x:A \vdash b :^{e_2} B \quad E \vdash b\{x \leftarrow a\} :^{e_3} B}{E \vdash \text{let } x = a \text{ in } b \leftrightarrow \text{let } x = a \text{ in } b\{x \leftarrow a\} : B}$$

$$\frac{E \vdash a :^{e_1} A \quad E, x:A \vdash b :^{e_2} B \quad E, x:A, y : B \vdash c :^{e_3} C}{E \vdash \text{let } x = a \text{ in let } y = b \text{ in } c \leftrightarrow \text{let } y = (\text{let } x = a \text{ in } b) \text{ in } (\text{let } x = a \text{ in } c) : C}$$

The following are special cases of (Eq Drop):

$$\frac{E, \rho \vdash x :^{\varnothing} A}{E \vdash (\nu\rho)x \leftrightarrow x :^{\varnothing} A}$$

$$\frac{E, \rho \vdash v \text{ at } \rho' :^{\{\rho'\}} (A \xrightarrow{e} B) \text{ at } \rho' \quad \rho \notin \text{fr}(A \xrightarrow{e} B \text{ at } \rho')}{E \vdash (\nu\rho)(v \text{ at } \rho') \leftrightarrow v \text{ at } \rho' :^{\{\rho'\}} (A \xrightarrow{e} B) \text{ at } \rho'}$$

In the following example, we apply (Eq Let Assoc) followed by (Eq Letregion Let) to optimise a computation by replacing a single global region ρ by two

smaller local regions ρ_1 and ρ_2 whose lives do not overlap, and hence could share storage.

$$\begin{aligned}
& \emptyset \vdash (\nu\rho) \text{let } f = \lambda(x)x \text{ at } \rho \text{ in let } y = f(5) \\
& \quad \text{in let } g = \lambda(z)y \text{ at } \rho \text{ in } g(42) \\
& \Leftrightarrow (\nu\rho) \text{let } y = (\text{let } f = \lambda(x)x \text{ at } \rho \text{ in } f(5)) \\
& \quad \text{in let } g = \lambda(z)y \text{ at } \rho \text{ in } g(42) \\
& \Leftrightarrow \text{let } y = (\nu\rho_1)(\text{let } f = \lambda(x)x \text{ at } \rho_1 \text{ in } f(5)) \\
& \quad \text{in } (\nu\rho_2) \text{let } g = \lambda(z)y \text{ at } \rho_2 \text{ in } g(42) \quad : \text{Lit}
\end{aligned}$$

Although our equations are not decorated with effects, for any derivable equation $a_1 \leftrightarrow a_2$ there is an effect e that is an upper bound of the effects of both a_1 and a_2 .

Lemma 5.1 *If $E \vdash a_1 \leftrightarrow a_2 : A$ then there is $e \subseteq \text{dom}(E)$ such that for each $i \in 1..2$, there is $e_i \subseteq e$ with $E \vdash a_i :^{e_i} A$.*

Using standard π -calculus techniques, we can show that our equational theory is sound with respect to our π -calculus semantics.

Theorem 5.2 *Suppose $E \vdash a \leftrightarrow b : A$ and $k \notin \text{dom}(E) \cup L$. Then:*

$$\llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k \approx \llbracket b \rrbracket k$$

Tofte and Talpin proved a result that the operational behaviour of a region-annotated term (like the terms of our calculus) is the same as its erasure to a pure term of the λ -calculus. We conjecture that our equational theory is sound for a standard contextual equivalence for the region calculus, and that this could easily be shown by appealing to Tofte and Talpin's result.

6 Extensions

In this section, we show that the main results of the paper apply not only to the simple region calculus of Section 2 but also to that calculus extended with recursive functions, lists, and region polymorphism. We describe this extended calculus in Section 6.1. Then in Section 6.2 we describe an extended π -calculus. Its extensions are recursive types, to model lists, and group polymorphism, to model region polymorphism. In Section 6.3 we define an encoding of the extended region calculus in this extended π -calculus. With the exception of the results in Section 5 concerning equational reasoning, all the other theorems in the paper concerning the unextended calculi can be generalized to the extended calculi. We omit the statement of these generalized theorems from this section, but in Appendix B we state and prove all these theorems. We conjecture that the material in Section 5 could be generalized also, but we have not investigated this generalization.

6.1 An Extended λ -Calculus

Here is the extended syntax of expressions and values.

Expressions and Values:

x, y, p, q, f, g	name: variable, pointer, literal
ρ	region variable
$a, b ::=$	expression
x	variable or pointer or literal
$v \text{ at } \rho$	allocation of v at ρ
$x[\rho_1, \dots, \rho_n](y)$	application
$\text{let } x = a \text{ in } b$	sequencing
$\text{letregion } \rho \text{ in } b$	region allocation and de-allocation
$\text{case } x \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2$	list case
$u, v ::=$	boxed value
$\mu(f:A)\lambda[\rho_1, \dots, \rho_n](x)b$	recursive function
nil	empty list
$x_1 :: x_2$	cons cell

Previously, the only kind of value was function abstraction. In this calculus, a boxed value can be a recursive, region-polymorphic function, an empty list, or a cons cell.

In a function value $\mu(f:A)\lambda[\rho_1, \dots, \rho_n](x)b$, the names f and x and the region variables ρ_1, \dots, ρ_n are bound, with scope b . During evaluation, the name x gets bound to the function's argument and the name f gets bound to the function itself, to enable recursive calls. The region parameters ρ_1, \dots, ρ_n allow the function to allocate and read from regions passed in as arguments. This region polymorphism is essential for efficient code generation in the ML Kit compiler (Tofte and Talpin 1997). Other kind of boxed values are lists, that is either the empty list, nil , or a cons cell, $x_1 :: x_2$, where the names x_1 and x_2 are heap pointers referring to the head and tail of the list, respectively.

A new expression for function application, $x[\rho_1, \dots, \rho_n](y)$, applies the function pointed to by x to the region parameters ρ_1, \dots, ρ_n , and the value parameter y . The other new expression, $\text{case } x \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2$, is for list discrimination. In this expression, the names y_1 and y_2 are bound, with scope b_2 . When the expression evaluates, if x is the empty list, b_1 runs. Otherwise, if x is a cons cell $x_1 :: x_2$, then $b_2\{y_1 \leftarrow x_1\}\{y_2 \leftarrow x_2\}$ runs. The other expressions of the extended calculus have the same interpretation as in the unextended calculus.

The definitions of regions, heaps, and stacks needed for the dynamic semantics are the same as before, though the set of values, v , stored in regions is extended.

Regions, Heaps and Stacks:

$r ::= (p_i \mapsto v_i)_{i \in 1..n}$	region, p_i distinct
$h ::= (\rho_i \mapsto r_i)_{i \in 1..n}$	heap, ρ_i distinct

$S ::= \{\rho_1, \dots, \rho_n\}$

stack of live regions

The evaluation relation, $S \cdot (a, h) \Downarrow (p, h')$, is defined by the rules in the following table.

Evaluation Rules:

(Eval Var)

$$\frac{}{S \cdot (p, h) \Downarrow (p, h)}$$

(Eval Alloc)

$$\frac{\rho \in S \quad p \notin \text{dom}_2(h)}{S \cdot (v \text{ at } \rho, h) \Downarrow (p, h + (\rho \mapsto (h(\rho) + (p \mapsto v))))}$$

 (Eval Appl) (where $\rho \in S$ and $h(\rho)(p) = \mu(f:A)\lambda[\rho_1, \dots, \rho_n](x)b$)

$$\frac{S \cdot (b\{f \leftarrow p\}\{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}\{x \leftarrow q\}, h) \Downarrow (p', h')}{S \cdot (p[\rho'_1, \dots, \rho'_n](q), h) \Downarrow (p', h')}$$

(Eval Let)

$$\frac{S \cdot (a, h) \Downarrow (p', h') \quad S \cdot (b\{x \leftarrow p'\}, h') \Downarrow (p'', h'')}{S \cdot (\text{let } x = a \text{ in } b, h) \Downarrow (p'', h')}$$

(Eval Letregion)

$$\frac{\rho \notin \text{dom}(h) \quad S \cup \{\rho\} \cdot (a, h + \rho \mapsto \emptyset) \Downarrow (p', h')}{S \cdot (\text{letregion } \rho \text{ in } a, h) \Downarrow (p', h')}$$

(Eval Case 1)

$$\frac{\rho \in S \quad h(\rho)(p) = \text{nil} \quad S \cdot (b_1, h) \Downarrow (p', h')}{S \cdot (\text{case } p \text{ of nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) \Downarrow (p', h')}$$

(Eval Case 2)

$$\frac{\rho \in S \quad h(\rho)(p) = q_1 :: q_2 \quad S \cdot (b_2\{y_1 \leftarrow q_1\}\{y_2 \leftarrow q_2\}, h) \Downarrow (p', h')}{S \cdot (\text{case } p \text{ of nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) \Downarrow (p', h')}$$

Next, we introduce the effects, types, and environments needed for the static semantics. The definitions of effects and environments are unchanged, but we need to introduce new types for region polymorphic functions and for lists.

In the extended type system, a function value $(\mu(f:F)\lambda[\rho_1, \dots, \rho_n](x)b)$ at ρ , will have a type $F = (\forall[\rho_1, \dots, \rho_n]A \xrightarrow{c} B)$ at ρ , where A is the type of the function argument x , and the regions ρ_1, \dots, ρ_n are bound. A list stored at ρ will have type $[A]$ at ρ , where A is the type of the elements of the list. Note that nil is an overloaded constant, which inhabits every well-formed type, and that each element of a list are stored in the same region than the list itself.

Effects, Types, and Environments:

$e ::= \{\rho_1, \dots, \rho_n\}$	effect
$A, B, F ::=$	type of expressions
Lit	type of literals
$V \text{ at } \rho$	type of V values at ρ
$U, V ::=$	type of boxed values
$\forall[\rho_1, \dots, \rho_n]A \xrightarrow{e} B$	polymorphic function
$[A]$	list
$E ::=$	environment
\emptyset	empty environment
E, ρ	entry for a region ρ
$E, x:A$	entry for a name x

In the type $(\forall[\rho_1, \dots, \rho_n]A \xrightarrow{e} B) \text{ at } \rho$, the regions ρ_1, \dots, ρ_n are bound with scope $A \xrightarrow{e} B$. Let $fr(A)$ be the set of region variables free in the type A . We have $fr(Lit) = \emptyset$, and $fr((\forall[\rho_1, \dots, \rho_n]A \xrightarrow{e} B) \text{ at } \rho) = (fr(A) \cup fr(B) \cup e) - \{\rho_1, \dots, \rho_n\} \cup \{\rho\}$, and $fr([A] \text{ at } \rho) = fr(A) \cup \{\rho\}$. We identify types up to consistent renaming of bound regions.

The static semantics consists of judgments with the same format as before: good environments, $E \vdash \diamond$, good types, $E \vdash A$, and good expressions, $E \vdash a :^e A$. The rules in the following tables define the static semantics. For any substitution σ of regions for regions and effect $e = \{\rho_1, \dots, \rho_n\}$, the effect $e\sigma$ is the set of regions $\{\sigma(\rho_1), \dots, \sigma(\rho_n)\}$.

Typing Rules:

(Env \emptyset)	(Env x)	(Env ρ)	(Type Lit)
$\frac{}{\emptyset \vdash \diamond}$	$\frac{E \vdash A \quad x \notin dom(E) \cup L}{E, x:A \vdash \diamond}$	$\frac{E \vdash \diamond \quad \rho \notin dom(E)}{E, \rho \vdash \diamond}$	$\frac{E \vdash \diamond}{E \vdash Lit}$
(Type \rightarrow) (where $E' = E, \rho_1, \dots, \rho_n$)	(Type List)		
$\frac{E' \vdash A \quad e \subseteq dom(E') \quad E' \vdash B \quad \rho \in dom(E)}{E \vdash (\forall[\rho_1, \dots, \rho_n]A \xrightarrow{e} B) \text{ at } \rho}$	$\frac{E \vdash A \quad \rho \in dom(E)}{E \vdash [A] \text{ at } \rho}$		
(Exp x)	(Exp ℓ)		
$\frac{E, x:A, E' \vdash \diamond}{E, x:A, E' \vdash x :^\emptyset A}$	$\frac{E \vdash \diamond \quad \ell \in L}{E \vdash \ell :^\emptyset Lit}$		
(Exp Appl) (where $\sigma = \{\rho_1 \leftarrow \rho'_1\} \dots \{\rho_n \leftarrow \rho'_n\}$ and $\{\rho'_1, \dots, \rho'_n\} \subseteq dom(E)$)			
$\frac{E \vdash x :^\emptyset (\forall[\rho_1, \dots, \rho_n]A \xrightarrow{e} B) \text{ at } \rho \quad E \vdash y :^\emptyset A\sigma}{E \vdash x[\rho'_1, \dots, \rho'_n](y) :^{\{\rho\} \cup (e\sigma)} B\sigma}$			
(Exp Let)	(Exp Letregion)		
$\frac{E \vdash a :^e A \quad E, x:A \vdash b :^{e'} B}{E \vdash let \ x = a \ in \ b :^{e \cup e'} B}$	$\frac{E, \rho \vdash a :^e A \quad E \vdash A}{E \vdash letregion \ \rho \ in \ a :^{e - \{\rho\}} A}$		

$$\begin{array}{c}
\text{(Exp Case)} \\
\frac{E \vdash x :^{\varnothing} [A] \text{ at } \rho \quad E \vdash b_1 :^{e_1} B \quad E, y_1:A, y_2:[A] \text{ at } \rho \vdash b_2 :^{e_2} B}{E \vdash \text{case } x \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2 :^{\{\rho\} \cup e_1 \cup e_2} B} \\
\\
\text{(Exp Fun) (where } F = (\forall[\rho_1, \dots, \rho_n] A \xrightarrow{e} B) \text{ at } \rho) \\
\frac{E, f:F, \rho_1, \dots, \rho_n, x:A \vdash b :^{e'} B \quad e' \subseteq e \subseteq \text{dom}(E, \rho_1, \dots, \rho_n)}{E \vdash (\mu(f:F)\lambda[\rho_1, \dots, \rho_n](x)b) \text{ at } \rho :^{\{\rho\}} F} \\
\\
\begin{array}{cc}
\text{(Exp Nil)} & \text{(Exp Cons)} \\
\frac{E \vdash [A] \text{ at } \rho}{E \vdash \text{nil} \text{ at } \rho :^{\{\rho\}} [A] \text{ at } \rho} & \frac{E \vdash x_1 :^{\varnothing} A \quad E \vdash x_2 :^{\varnothing} [A] \text{ at } \rho}{E \vdash (x_1 :: x_2) \text{ at } \rho :^{\{\rho\}} [A] \text{ at } \rho}
\end{array}
\end{array}$$

The definitions of region and heap typings, R and H , respectively, and of the judgments $E \vdash r \text{ at } \rho : R$, $H \models \diamond$, $H \models h$, and $H \models S \cdot (a, h) : A$ are exactly as in Section 2.

6.2 An Extended π -Calculus

We enrich our typed π -calculus with group polymorphism and recursive types.

The idea of group polymorphism is that instead of simply exchanging tuples of names with fixed types on a channel, we exchange tuples of names together with tuples of groups, where the types of the names depend on the groups. Accordingly, the type of a channel acquires the form $G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{H}$, where G is the group of the channel, G_1, \dots, G_m are group parameters, T_1, \dots, T_n are the types of the name parameters, and \mathbf{H} is the hidden effect. The types T_1, \dots, T_n and the effect \mathbf{H} may depend on the group parameters G_1, \dots, G_m . An output process takes the form $\overline{x}(G_1, \dots, G_m, y_1, \dots, y_n)$, where G_1, \dots, G_m are the group parameters, and y_1, \dots, y_n are the name parameters. An input process takes the form $x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P$ where G_1, \dots, G_m receive the group parameters, and y_1, \dots, y_n receive the name parameters. This treatment of group polymorphism, where group parameters are transmitted on channels, is inspired by previous treatments of type polymorphism in the π -calculus (Turner 1995; Pierce and Sangiorgi 1997), where type parameters are transmitted on channels. Group polymorphism allows to type-check richer behaviour, such as an encoding of region polymorphism, than previously. Still, group polymorphism does not introduce any new dynamic behaviour: the reductions of any well-typed process are equivalent to the reductions of its untyped erasure.

The idea of recursive types is standard. A recursive type takes the form $\mu(X)T$. A name of type $\mu(X)T$ is deemed also to have the unfolded type $T\{X \leftarrow \mu(X)T\}$, and vice versa. However, for the sake of simplicity, we do not identify a recursive type with its unfolding. A name may be assigned the type $\mu(X)X$, but such a name cannot be used for communication since we cannot unfold $\mu(X)X$ to a channel type.

The extended syntax of our π -calculus is as follows:

Types, Expressions, and Processes:

G, H	group
X	type variable
$T ::=$	channel type
X	type variable
$G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{H}$	channel type
$\mu(X)T$	recursive type
x, y, p, q	name: variable, channel
$P, Q, R ::=$	process
$x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P$	input (no $y_i \in \text{inp}(P)$)
$\bar{x}(G_1, \dots, G_m, y_1, \dots, y_n)$	output
$(\nu G)P$	new-group: group restriction
$(\nu x:T)P$	new-name: name restriction
$P \mid Q$	composition
$!P$	replication
$\mathbf{0}$	inactivity

In the type $G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{H}$, the groups G_1, \dots, G_m are bound with scope T_1, \dots, T_n and \mathbf{H} . In the type $\mu(X)T$, the type variable X is bound with scope T . In a process $x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P$, the groups G_1, \dots, G_m and the names y_1, \dots, y_n are bound; their scope is P . The other binders, new-name and new-group, have the same semantics as before. The definitions of free names of a process, $fn(P)$, free groups of a type, $fg(T)$, and free groups of a process, $fg(P)$, are as before, except for the following changes:

Free groups, $fg(T)$, of type T :

$$\begin{aligned}
 fg(X) &\triangleq \emptyset \\
 fg(G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{H}) &\triangleq \\
 &\quad \{G\} \cup ((fg(T_1) \cup \dots \cup fg(T_n) \cup \mathbf{H}) - \{G_1, \dots, G_m\}) \\
 fg(\mu(X)T) &\triangleq fg(T)
 \end{aligned}$$

Free groups, $fg(P)$, of process P :

$$\begin{aligned}
 fg(x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P) &\triangleq (fg(T_1) \cup \dots \cup fg(T_n) \cup fg(P)) \\
 &\quad - \{G_1, \dots, G_m\} \\
 fg(\bar{x}(G_1, \dots, G_m, y_1, \dots, y_n)) &\triangleq \{G_1, \dots, G_m\} \\
 fg((\nu G)P) &\triangleq fg(P) - \{G\} \\
 fg((\nu x:T)P) &\triangleq fg(T) \cup fg(P) \\
 fg(P \mid Q) &\triangleq fg(P) \cup fg(Q) \\
 fg(!P) &\triangleq fg(P) \\
 fg(\mathbf{0}) &\triangleq \emptyset
 \end{aligned}$$

We identify types and processes up to consistent renaming of bound groups, names, and type variables.

We define structural congruence $P \equiv Q$ by the same rules as before, except that we replace (Struct Input) with the following:

$$\begin{array}{l}
P \equiv Q \Rightarrow \hspace{15em} \text{(Struct Input)} \\
x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P \equiv \\
x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).Q
\end{array}$$

We define reduction $P \rightarrow Q$ by the following rules:

Reduction:

$\bar{x}\langle G'_1, \dots, G'_m, y'_1, \dots, y'_n \rangle \mid x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P$	(Red Interact)
$\rightarrow P\{G_1 \leftarrow G'_1\} \cdots \{G_m \leftarrow G'_m\} \{y_1 \leftarrow y'_1\} \cdots \{y_n \leftarrow y'_n\}$	
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	(Red Par)
$P \rightarrow Q \Rightarrow (\nu G)P \rightarrow (\nu G)Q$	(Red GRes)
$P \rightarrow Q \Rightarrow (\nu x:T)P \rightarrow (\nu x:T)Q$	(Red Res)
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red \equiv)

To take recursive types into account, we extend the definition of type environment to include type variables, X . The definition of the domain, $dom(E)$, of an environment, E , is also extended and is defined by the equations $dom(\emptyset) = \emptyset$, $dom(E, \rho) = dom(E) \cup \{\rho\}$, $dom(E, x:A) = dom(E) \cup \{x\}$ and $dom(E, X) = dom(E) \cup \{X\}$.

Environments:

$E ::=$	environment
\emptyset	empty environment
E, X	entry for a type variable X
E, G	entry for a group G
$E, x:T$	entry for a variable x

The judgments of the type system have the same format as previously: good environment $E \vdash \diamond$, good type $E \vdash T$, good name $E \vdash x : T$, and good process $E \vdash P : \mathbf{H}$. Their meaning is given inductively by the rules in the following tables.

Good environments:

$(\text{Env } \emptyset)$	$(\text{Env } x)$
$\frac{}{\emptyset \vdash \diamond}$	$\frac{E \vdash T \quad x \notin dom(E)}{E, x:T \vdash \diamond}$
$(\text{Env } G)$	$(\text{Env } X)$
$\frac{E \vdash \diamond \quad G \notin dom(E)}{E, G \vdash \diamond}$	$\frac{E \vdash \diamond \quad X \notin dom(E)}{E, X \vdash \diamond}$

Good types:

$$\frac{\text{(Type Chan) (where } E' = E, G_1, \dots, G_m)}{E' \vdash \diamond \quad G \in \text{dom}(E) \quad \mathbf{H} \subseteq \text{dom}(E') \quad E' \vdash T_i \quad \forall i \in 1..n} \\ E \vdash G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{H}$$

$$\frac{\text{(Type X)} \quad E', X, E'' \vdash \diamond}{E', X, E'' \vdash X} \quad \frac{\text{(Type Rec)} \quad E, X \vdash T}{E \vdash \mu(X)T}$$

Good names:

$$\frac{\text{(Exp } x)}{E', x:T, E'' \vdash \diamond} \quad \frac{\text{(Exp Unfold)}}{E \vdash x : \mu(X)T} \quad \frac{\text{(Exp Fold)}}{E \vdash x : T\{X \leftarrow \mu(X)T\}} \\ E', x:T, E'' \vdash x : T \quad E \vdash x : T\{X \leftarrow \mu(X)T\} \quad E \vdash x : \mu(X)T$$

Good processes:

$$\frac{\text{(Proc Input) (where } (\mathbf{G} - \mathbf{H}) \cap \{G_1, \dots, G_m\} = \emptyset)}{E \vdash x : G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{H} \\ E, G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n \vdash P : \mathbf{G}} \\ E \vdash x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P : \{G\} \cup (\mathbf{G} - \mathbf{H})$$

$$\frac{\text{(Proc Output) (where } \sigma = \{G_1 \leftarrow G'_1\} \dots \{G_m \leftarrow G'_m\})}{E \vdash x : G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{H} \\ \{G'_1, \dots, G'_m\} \subseteq \text{dom}(E) \quad E \vdash y'_i : T_i \sigma \quad \forall i \in 1..n} \\ E \vdash \bar{x}(G'_1, \dots, G'_m, y'_1, \dots, y'_n) : \{G\} \cup \mathbf{H}\sigma$$

$$\frac{\text{(Proc GRes)} \quad E, G \vdash P : \mathbf{H}}{E \vdash (\nu G)P : \mathbf{H} - \{G\}} \quad \frac{\text{(Proc Res)} \quad E, x:T \vdash P : \mathbf{H}}{E \vdash (\nu x:T)P : \mathbf{H}}$$

$$\frac{\text{(Proc Par)} \quad E \vdash P : \mathbf{G} \quad E \vdash Q : \mathbf{H}}{E \vdash P \mid Q : \mathbf{G} \cup \mathbf{H}} \quad \frac{\text{(Proc Repl)} \quad E \vdash P : \mathbf{H}}{E \vdash !P : \mathbf{H}} \quad \frac{\text{(Proc Zero)}}{E \vdash \mathbf{0} : \emptyset}$$

$$\frac{\text{(Proc Subsum)} \quad E \vdash P : \mathbf{G} \quad \mathbf{G} \subseteq \mathbf{H} \subseteq \text{dom}(E)}{E \vdash P : \mathbf{H}}$$

The standard rule (Type Rec) for checking goodness of a recursive types $\mu(X)T$ records the name of the recursively bound variable X by inserting it into the environment used to check goodness of the body T . This is the only circumstance in which we are interested in having type variables in an environment. We are only interested in the behaviour of processes type-checked in proper environments, those in which no type variables occur.

Proper environments:

Let E be *proper* if and only if $E \vdash \diamond$ but there is no X such that $E \vdash X$.

The relation $P \downarrow \beta$ where the barb $\beta \in \{x, \bar{x}\}$, is defined much as before.

Exhibition of a barb:

(Barb Input)	(Barb Output)
$x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P \downarrow x$	$\bar{x}(G_1, \dots, G_m, y_1, \dots, y_n) \downarrow \bar{x}$
(Barb GRes) $P \downarrow \beta$	(Barb Res) $P \downarrow \beta \quad \beta \notin \{x, \bar{x}\}$
$(\nu x:T)P \downarrow \beta$	$(\nu x:T)P \downarrow \beta$
	(Barb Par) $P \downarrow \beta$
	$P \mid Q \downarrow \beta$
	(Barb \equiv) $P \equiv Q \quad Q \downarrow \beta$
	$P \downarrow \beta$

The definition of a relation on typed processes is the same as before, except we restrict attention to proper environments:

- A relation on typed processes (of the extended π -calculus), \mathcal{S} , is a set of triples (E, P, Q) where E is a proper environment and P and Q are typed terms such that $E \vdash P$ and $E \vdash Q$.

The definition of barbed congruence, and the auxiliary notions including barbed bisimulation and barbed bisimilarity, are exactly as before.

6.3 An Extended Encoding

We translate the extended region calculus into the extended π -calculus as follows. In this encoding, the type of a boxed value located at region ρ is of the kind $\rho(\rho_1, \dots, \rho_m)[T_1, \dots, T_n] \setminus \mathbf{H}$. In the common case when $m = 0$, that is, the value is monomorphic, and has no hidden effect, we abbreviate the type to $\rho[T_1, \dots, T_n]$.

Translating of the region calculus to the π -calculus:

$\llbracket A \rrbracket$	type modelling the type A
$\llbracket E \rrbracket$	environment modelling proper environment E
$\llbracket a \rrbracket k$	process modelling term a , answer on k
$\llbracket p \mapsto v \rrbracket$	process modelling value v at pointer p
$\llbracket r \rrbracket$	process modelling region r
$\llbracket S \cdot (a, h) \rrbracket k$	process modelling configuration $S \cdot (a, h)$

Translation rules:

$\llbracket Lit \rrbracket \triangleq Lit[]$
$\llbracket (\forall[\rho_1, \dots, \rho_n]A \xrightarrow{e} B) \text{ at } \rho \rrbracket \triangleq \rho(\rho_1, \dots, \rho_n)(\llbracket A \rrbracket, K[\llbracket B \rrbracket]) \setminus (e \cup \{K\})$
$\llbracket A \text{ at } \rho \rrbracket \triangleq \mu(X)\rho[\rho[], \rho[\llbracket A \rrbracket, X]]$

$$\begin{aligned}
\llbracket \emptyset \rrbracket &\triangleq K, \text{Lit}, \ell_1: \text{Lit}[], \dots, \ell_n: \text{Lit}[] \\
\llbracket E, \rho \rrbracket &\triangleq \llbracket E \rrbracket, \rho \\
\llbracket E, x:A \rrbracket &\triangleq \llbracket E \rrbracket, x: \llbracket A \rrbracket \\
\llbracket x \rrbracket k &\triangleq \bar{k}(x) \\
\llbracket \text{let } x = a_A \text{ in } b \rrbracket k &\triangleq (\nu k': K[\llbracket A \rrbracket]) (\llbracket a \rrbracket k' \mid k'(x: \llbracket A \rrbracket). \llbracket b \rrbracket k) \\
\llbracket p[\rho_1, \dots, \rho_n](q) \rrbracket k &\triangleq \bar{p}(\rho_1, \dots, \rho_n, q, k) \\
\llbracket (v \text{ at } \rho)_A \rrbracket k &\triangleq (\nu p: \llbracket A \rrbracket) (\llbracket p \mapsto v \rrbracket \mid \bar{k}(p)) \\
\llbracket \text{letregion } \rho \text{ in } a \rrbracket k &\triangleq (\nu \rho) \llbracket a \rrbracket k \\
\llbracket \text{case } p_{\llbracket A \rrbracket \text{ at } \rho} \text{ of nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2 \rrbracket k &\triangleq (\nu z_1: \rho[]) (\nu z_2: \rho[\llbracket A \rrbracket, \llbracket [A] \text{ at } \rho \rrbracket]) \\
&\quad (\bar{p}(z_1, z_2) \mid z_1(). \llbracket b_1 \rrbracket k \mid z_2(y_1: \llbracket A \rrbracket, y_2: \llbracket [A] \text{ at } \rho \rrbracket). \llbracket b_2 \rrbracket k) \\
\llbracket p \mapsto \mu(f:F) \lambda[\rho_1, \dots, \rho_n](x)b \rrbracket &\triangleq !p(\rho_1, \dots, \rho_n, x: \llbracket A \rrbracket, k: K[\llbracket B \rrbracket]). \llbracket b\{f \leftarrow p\} \rrbracket k \\
&\quad \text{where } F = (\forall[\rho_1, \dots, \rho_n] A \xrightarrow{c} B) \text{ at } \rho \\
\llbracket p \mapsto \text{nil}_{\llbracket A \rrbracket \text{ at } \rho} \rrbracket &\triangleq !p(z_1: \rho[], z_2: \rho[\llbracket A \rrbracket, \llbracket [A] \text{ at } \rho \rrbracket]). \bar{z}_1() \\
\llbracket p \mapsto (x_1 :: x_2)_{\llbracket A \rrbracket \text{ at } \rho} \rrbracket &\triangleq !p(z_1: \rho[], z_2: \rho[\llbracket A \rrbracket, \llbracket [A] \text{ at } \rho \rrbracket]). \bar{z}_2(x_1, x_2) \\
\llbracket (p_i \mapsto v_i)_{i \in 1..n} \rrbracket &\triangleq \prod_{i \in 1..n} \llbracket p_i \mapsto v_i \rrbracket \\
\llbracket (\rho_i \mapsto r_i)_{i \in 1..n} \rrbracket &\triangleq \prod_{i \in 1..n} \llbracket r_i \rrbracket \\
\llbracket S \cdot (a, h_H) \rrbracket k &\triangleq (\nu \vec{\rho}_{\text{defunct}}) (\nu \llbracket ptr(H) \rrbracket) (\llbracket a \rrbracket k \mid \llbracket h \rrbracket) \\
&\quad \text{where } \{\vec{\rho}_{\text{defunct}}\} = \text{dom}(H) - S
\end{aligned}$$

The translation of the extended region calculus is an extension of the encoding given in Section 4. In particular, the encodings of type environments, regions, heaps and configurations are unchanged.

The encoding of lists and the case expression are standard (Milner 1999). A polymorphic recursive function is modelled as a replicated input process, awaiting the argument of the function, a continuation on which to return a result and for group parameters representing the region parameters to the function. A function is invoked by sending it the argument and a continuation channel.

Appendix B states and proves reformulations of all the results stated in Sections 2, 3, and 4 in terms of the extended calculi of this section.

7 Conclusions

We showed that the static and dynamic semantics of Tofte and Talpin's region calculus are preserved by a translation into a typed π -calculus. The *letregion* construct is modelled by a new-group construct originally introduced into process calculi by Cardelli, Ghelli, and Gordon (2000a). We showed that the rather subtle correctness of memory de-allocation in the region calculus is an instance of Theorem 4.3, a new garbage collection principle for the π -calculus. The translation is an example of how the new-group construct accounts for the type generativity introduced by *letregion*, just as the standard new-name construct of the π -calculus accounts for dynamic generation of values.

Banerjee, Heintze, and Riecke (1999) give an alternative proof of the soundness of region-based memory management. Theirs is obtained by interpreting the region calculus in a polymorphic λ -calculus equipped with a new binary type constructor $\#$ that behaves like a union or intersection type. Their techniques are those of denotational semantics, completely different from the operational techniques of this paper. The formal connections between the two approaches are not obvious but would be intriguing to investigate. A possible advantage of our semantics in the π -calculus is that it could easily be extended to interpret a region calculus with concurrency, but that remains future work. Another line of future work is to consider the semantics of other region calculi (Aiken, Faehndrich, and Levien 1995; Crary, Walker, and Morrisett 1999; Hughes and Pareto 1999) in terms of the π -calculus. Finally, various researchers (Moggi and Palumbo 1999; Semmelroth and Sabry 1999) have noted a connection between the monadic encapsulation of state in Haskell (Launchbury and Peyton Jones 1995) and regions; hence it would be illuminating to interpret monadic encapsulation in the π -calculus.

Acknowledgements

Luca Cardelli participated in the initial discussions that led to this paper. We had useful conversations with Giorgio Ghelli, Cédric Fournet and Mads Tofte on the connections between groups and regions. Thanks to Simon Helsen for pointing out a problem with the rules (Eq Fun β) and (Eq Let β) in the original version of this technical report. Luca Cardelli, Tony Hoare, and Andy Moran commented on a draft.

References

- Abadi, M. and A. D. Gordon (1999). A calculus for cryptographic protocols: The spi calculus. *Information and Computation* 148, 1–70. An extended version appears as Research Report 149, Digital Equipment Corporation Systems Research Center, January 1998.
- Aiken, A., M. Faehndrich, and R. Levien (1995). Better static memory management: Improvements to region-based analysis of higher-order languages. In *Proceedings PLDI'95*, pp. 174–185.
- Banerjee, A., N. Heintze, and J. Riecke (1999). Region analysis and the polymorphic lambda calculus. In *Proceedings LICS'99*.
- Benton, N. and A. Kennedy (1999). Monads, effects and transformations. In *Proceedings HOOTS99*, Volume 26 of *Electronic Notes in Theoretical Computer Science*, pp. 1–18. Elsevier.
- Birkedal, L., M. Tofte, and M. Vejlstrup (1996). From region inference to von Neumann machines via region representation inference. In *Proceedings POPL'96*, pp. 171–183. ACM.
- Boudol, G. (1992). Asynchrony and the π -calculus. Technical Report 1702, INRIA.
- Cardelli, L., G. Ghelli, and A. D. Gordon (2000a). Ambient groups and mobility types. In *Proceedings of IFIP TCS2000*. to appear.
- Cardelli, L., G. Ghelli, and A. D. Gordon (2000b). Group creation and secrecy. Submitted for publication.
- Crary, K., D. Walker, and G. Morrisett (1999). Typed memory management in a calculus of capabilities. In *Proceedings POPL'99*, pp. 262–275.
- Dal Zilio, S. (1999). A bisimulation for the blue calculus. Technical Report 3664, INRIA.
- Fournet, C. and G. Gonthier (1996). The reflexive CHAM and the Join-calculus. In *Proceedings POPL'96*, pp. 372–385.
- Gifford, D. K. and J. M. Lucassen (1986). Integrating functional and imperative programming. In *Proceedings L'FP'86*, pp. 28–38.
- Honda, K. (1992). Two bisimilarities for the ν -calculus. Technical Report 92-002, Department of Computer Science, Keio University.
- Hughes, J. and L. Pareto (1999). Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings ICFP'99*, pp. 70–81.
- Launchbury, J. and S. Peyton Jones (1995). State in Haskell. *Lisp and Symbolic Computation* 8(4), 293–341.
- Leroy, X. (1996). A syntactic theory of type generativity and sharing. *Journal of Functional Programming* 6(5), 667–698.

- Merro, M. and D. Sangiorgi (1998). On asynchrony in name-passing calculi. In *Proceedings ICALP'98*, Volume 1443 of *Lecture Notes in Computer Science*, pp. 856–867. Springer.
- Milner, R. (1999). *Communicating and Mobile Systems: the π -Calculus*. CUP.
- Milner, R., J. Parrow, and D. Walker (1992). A calculus of mobile processes, parts i and ii. *Journal of Information and Computation* 100, 1–77.
- Milner, R. and D. Sangiorgi (1992). Barbed bisimulation. In *Proceedings ICALP'92*, Volume 623 of *Lecture Notes in Computer Science*, pp. 685–695. Springer.
- Milner, R., M. Tofte, R. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press.
- Moggi, E. (1989). Notions of computations and monads. *Theoretical Computer Science* 93, 55–92.
- Moggi, E. and F. Palumbo (1999). Monadic encapsulation of effects: a revised approach. In *Proceedings HOOTS99*, Volume 26 of *Electronic Notes in Theoretical Computer Science*, pp. 119–136. Elsevier.
- Pierce, B. and D. Sangiorgi (1996). Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science* 6(5), 409–454.
- Pierce, B. and D. Sangiorgi (1997). Behavioral equivalence in the polymorphic pi-calculus. In *Principles of Programming Languages (POPL)*. Full version to appear in *Journal of the Association for Computing Machinery (JACM)*. Also available as INRIA-Sophia Antipolis Rapport de Recherche No. 3042 and as Indiana University Computer Science Technical Report 468.
- Pierce, B. C. and D. N. Turner (1997). Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, editors, MIT Press, 2000.
- Plotkin, G. D. (1975). Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science* 1, 125–159.
- Russo, C. V. (1996). Standard ML type generativity as existential quantification. Technical Report ECS-LFCS-96-344, LFCS, University of Edinburgh.
- Sangiorgi, D. and D. Walker (2000). *The pi-calculus: a theory of mobile processes*. Cambridge University Press. (forthcoming).
- Semmelroth, M. and A. Sabry (1999). Monadic encapsulation in ML. In *Proceedings ICFP'99*, pp. 8–17.
- Talpin, J.-P. and P. Jouvelot (1992). Polymorphic type, region and effect inference. *Journal of Functional Programming* 2(3), 245–271.

- Tofte, M. and J.-P. Talpin (1997). Region-based memory management. *Information and Computation* 132(2), 109–176.
- Turner, D. N. (1995). *The polymorphic pi-calculus: theory and implementation*. Ph. D. thesis, University of Edinburgh.
- Wadler, P. (1998). The marriage of effects and monads. In *Proceedings ICFP'98*, pp. 63–74.
- Walker, D. (1995). Objects in the pi-calculus. *Information and Computation* 116(2), 253–271.

A Review of the Untyped π -Calculus

In this section, we review the syntax and semantics of the untyped, polyadic, choice-free, asynchronous π -calculus (Milner 1999; Boudol 1992; Honda 1992). We impose two additional (standard) simplifications, that are: (1) the recipient of a name may only use it in output actions; (2) there are no operators for testing the equality (or inequality) of names. Intuitively, only the capability to output on a named channel may be transmitted.

The π -calculus fragment defined by these restrictions, also known as the local π -calculus (Merro and Sangiorgi 1998), has a richer equational theory than the full π -calculus, and can be regarded as a basis for some proposals of concurrent programming languages (Fournet and Gonthier 1996; Pierce and Turner 1997). The additional algebraic laws obtained in the local variant of π , such as, for example, the replication laws listed subsequently in Proposition A.8, are required in the proof of Theorem 5.2, the correctness of our proposed equational theory for the region calculus.

The syntax and dynamic semantics of the untyped π -calculus are defined in Appendixes A.1 and A.2, respectively. In Appendix A.3 we define an alternative semantics for the calculus based on a labelled transition system, that makes it easier to reason about possible reductions of a process. We also formulate Proposition A.1, which relates the reduction and transition semantics. In Appendix A.4 we define barbed congruence for the untyped calculus and we prove several algebraic laws that are useful in Appendix B.

A.1 Syntax

Processes of this calculus are those obtained from the typed π -calculus processes defined in Section 3 by erasing all type and group annotations.

Processes:

x, y, p, q	names
$P, Q, R ::=$	process
$x(y_1, \dots, y_n).P$	input (no $y_i \in \text{inp}(P)$)
$\bar{x}(y_1, \dots, y_n)$	output
$(\nu x)P$	restriction
$P \mid Q$	composition
$!P$	replication
$\mathbf{0}$	inactivity

The locality property is ensured using a syntactic restriction on the definition of inputs, $x(y_1, \dots, y_n).P$, namely that no parameter y_i is in $\text{inp}(P)$, where $\text{inp}(P)$ is the set of names x such that an input $x(z_1, \dots, z_m).P'$ occurs as a subprocess of P , with x not bound.

We write $P\{x \leftarrow x'\}$ for the outcome of a capture-avoiding substitution of x' for each free occurrence of the variable x in the process P . We identify processes

up to renaming of bound variables. We write $P = Q$ to mean that P and Q are the same up to renaming of bound variables.

A.2 Dynamic Semantics

We formalize the semantics of the untyped π -calculus using techniques identical to those applied in Section 2.2. In particular, a *reduction relation* between processes, $P \rightarrow Q$, is defined on top of an auxiliary *structural congruence relation*, $P \equiv Q$, that identifies processes up to simple rearrangements.

Structural Congruence:

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu x)P \equiv (\nu x)Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow x(y_1, \dots, y_n).P \equiv x(y_1, \dots, y_n).Q$	(Struct Input)
$P \mid \mathbf{0} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$	(Struct Res Res)
$x \notin fn(P) \Rightarrow (\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$	(Struct Res Par)

Reduction:

$\bar{x}(y_1, \dots, y_n) \mid x(z_1, \dots, z_n).P \rightarrow P\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\}$	(Red Interact)
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	(Red Par)
$P \rightarrow Q \Rightarrow (\nu x)P \rightarrow (\nu x)Q$	(Red Res)
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red \equiv)

This presentation of the π -calculus semantics allows for a simple and compact definition of the reduction rules in which the sub-processes having to interact—the redexes in λ -calculus terminology—appear in contiguous position. Nonetheless, the operational semantics of concurrent systems are commonly defined using labelled transition systems and, whereas a reduction semantics may be much more enlightening and simple than a transition semantics, the latter makes it easier to reason about the possible reductions of a process. For instance, it will be difficult to prove Lemma A.2, given in Appendix A.3, without the help of a labelled transition system.

A.3 Labelled Transition Semantics

The definitions in this section are adapted from the presentation of the labelled transition system of the spi calculus (Abadi and Gordon 1999). In order to define the labelled transition semantics, we need some new syntactic forms: abstractions, concretions, and agents.

- An *abstraction* is an expression of the form $(\vec{x})P$, where P is a process and \vec{x} is a sequence of variables x_1, \dots, x_n such that $n \geq 0$ and x_1, \dots, x_n are pairwise distinct and bound in P .
- A *concretion* is an expression of the form $(\nu \vec{z})\langle \vec{y} \rangle Q$, where Q is a process and \vec{z} and \vec{y} are sequences of variables z_1, \dots, z_m , and y_1, \dots, y_n , respectively, such that $m, n \geq 0$, and $\{\vec{z}\} \subseteq \{\vec{y}\}$, and z_1, \dots, z_m are pairwise distinct and bound in $\langle \vec{y} \rangle Q$.
- An *agent* is either a process, an abstraction, or a concretion. We use the metavariables A and B to stand for arbitrary agents.

For any abstraction, $(\vec{x})P$, let its *arity*, $|(\vec{x})P|$, be the length of the sequence \vec{x} . Similarly, for any concretion, $(\nu \vec{z})\langle \vec{y} \rangle Q$, let its *arity*, $|(\nu \vec{z})\langle \vec{y} \rangle Q|$, be the length of the sequence \vec{y} .

Let $fv(A)$ be the sets of free variables of an agent A . Like processes, both abstractions and concretions are identified up to consistent renaming of bound variables.

We extend the restriction and composition operators to arbitrary agents, as follows. For an abstraction, $(\vec{x})P$, we set:

$$\begin{aligned} (\nu y)(\vec{x})P &\triangleq (\vec{x})(\nu y)P \\ R \mid (\vec{x})P &\triangleq (\vec{x})(R \mid P) \end{aligned}$$

where we assume that the bound variables \vec{x} are disjoint from $\{y\} \cup fv(R)$.

For a concretion, $(\nu \vec{z})\langle \vec{y} \rangle Q$, we set:

$$\begin{aligned} (\nu x)(\nu \vec{z})\langle \vec{y} \rangle Q &\triangleq \begin{cases} (\nu x, \vec{z})\langle \vec{y} \rangle Q & \text{if } x \in \{\vec{y}\} \\ (\nu \vec{z})\langle \vec{y} \rangle (\nu x)Q & \text{otherwise} \end{cases} \\ R \mid (\nu \vec{z})\langle \vec{y} \rangle Q &\triangleq (\nu \vec{z})\langle \vec{y} \rangle (R \mid Q) \end{aligned}$$

assuming that $x \notin \{\vec{z}\}$ and that $\{\vec{z}\} \cap fv(R) = \emptyset$.

We define the dual composition $A \mid R$ symmetrically.

Next, we define processes obtained by combining abstractions and concretions of equal arity. If F is the abstraction $(\vec{x})P$ where $\vec{x} = x_1, \dots, x_n$ and C is the concretion $(\nu \vec{z})\langle \vec{y} \rangle Q$ where $\vec{y} = y_1, \dots, y_n$ and $\{\vec{z}\} \cap fv(P) = \emptyset$, we define the *interactions* $F@C$ and $C@F$ to be the processes given by:

$$\begin{aligned} F@C &\triangleq (\nu \vec{z})(P\{x_1 \leftarrow y_1\} \cdots \{x_n \leftarrow y_n\} \mid Q) \\ C@F &\triangleq (\nu \vec{z})(Q \mid P\{x_1 \leftarrow y_1\} \cdots \{x_n \leftarrow y_n\}) \end{aligned}$$

An *action* is either a barb or the distinguished *silent action* τ . The *labelled transition system* is written $P \xrightarrow{\alpha} A$, where P is a process, α is an action, and A is an agent. We define this relation inductively, by the following rules:

The Labelled Transition System:

(Trans In)	(Trans Out)
$x(y_1, \dots, y_n).P \xrightarrow{x} (y_1, \dots, y_n)P$	$\bar{x}\langle y_1, \dots, y_n \rangle \xrightarrow{\bar{x}} (\nu)\langle y_1, \dots, y_n \rangle \mathbf{0}$
(Trans Inter 1) (with $ F = C $)	(Trans Inter 2) (with $ F = C $)
$\frac{P \xrightarrow{x} F \quad Q \xrightarrow{\bar{x}} C}{P \mid Q \xrightarrow{\tau} F @ C}$	$\frac{P \xrightarrow{\bar{x}} C \quad Q \xrightarrow{x} F}{P \mid Q \xrightarrow{\tau} C @ F}$
(Trans Par 1)	(Trans Par 2)
$\frac{P \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} A \mid Q}$	$\frac{Q \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} P \mid A}$
(Trans Res)	(Trans Repl)
$\frac{P \xrightarrow{\alpha} A \quad \alpha \notin \{x, \bar{x}\}}{(\nu x)P \xrightarrow{\alpha} (\nu x)A}$	$\frac{P \mid !P \xrightarrow{\alpha} A}{!P \xrightarrow{\alpha} A}$

The following is a basic result that states that modulo structural congruence, the reduction relation exactly represents the silent action of the transition semantics. A proof of this property can be obtained by adapting the detailed proof of an equivalent result found in the extended version of Abadi and Gordon (1999) paper on the spi calculus.

Proposition A.1 $P \rightarrow Q$ if and only if there is a process R such that $P \xrightarrow{\tau} R$ and $R \equiv Q$.

Lemma A.2 $(\nu x)P \rightarrow Q$ if and only if there is a process P' such that $P \rightarrow P'$ and $Q \equiv (\nu x)P'$.

Proof Assume $(\nu x)P \rightarrow Q$. By Proposition A.1, there is a process R such that $(\nu x)P \xrightarrow{\tau} R$ and $R \equiv Q$. The judgment $(\nu x)P \xrightarrow{\tau} R$ must have been derived by (Trans Res) with $R = (\nu x)P'$ and $P \xrightarrow{\tau} P'$. By Proposition A.1, $P \rightarrow P'$. \square

A.4 Barbed Congruence

The notion of equivalence between untyped terms that we consider in this paper is barbed congruence (Milner and Sangiorgi 1992), a bisimulation-based behavioural equivalence that preserves a notion of observation, called barbs.

A *barb*, β , is either a name x or a co-name \bar{x} . We write $P \Downarrow \beta$ if there exists P' such that $P \rightarrow^* P'$ and $P' \downarrow \beta$, where the relation \downarrow is defined in the following table.

Exhibition of a barb:

(Barb Input)	(Barb Output)
$\frac{}{x(y_1, \dots, y_n).P \downarrow x}$	$\frac{}{\bar{x}\langle y_1, \dots, y_n \rangle \downarrow \bar{x}}$
(Barb Res)	(Barb Par) (Barb \equiv)
$\frac{P \downarrow \beta \quad \beta \notin \{x, \bar{x}\}}{(\nu x)P \downarrow \beta}$	$\frac{P \downarrow \beta}{P \mid Q \downarrow \beta} \quad \frac{P \equiv Q \quad Q \downarrow \beta}{P \downarrow \beta}$

The barbs exhibited by a process, P , are related to the labelled transitions that P can perform, that is, to the external communications through which a process may interact with an outer context. We can formalize this idea using the following proposition.

Proposition A.3 $P \downarrow \beta$ if and only if there is an agent A such that $P \xrightarrow{\beta} A$.

What follows is a series of definitions leading up to our definition of barbed congruence for the untyped π -calculus.

- For any relation on processes \mathcal{S} , let $P \equiv \mathcal{S} \equiv Q$ mean there are processes P' and Q' such that $P \equiv P'$, $P' \mathcal{S} Q'$, and $Q' \equiv Q$.
- A symmetric relation \mathcal{S} is a *barbed bisimulation* if and only if $P \mathcal{S} Q$ implies:
 - (1) If $P \downarrow \bar{x}$ then $Q \Downarrow \bar{x}$.
 - (2) If $P \rightarrow P'$ then there is Q' such that $Q \rightarrow^* Q'$ and $P' \equiv \mathcal{S} \equiv Q'$.
- A *renaming*, σ , is a substitution $\{x_1 \leftarrow x'_1\} \cdots \{x_n \leftarrow x'_n\}$ of names for names where $n \geq 0$ and the names x_1, \dots, x_n are pairwise distinct. Let $\text{dom}(\sigma) = \{x_1, \dots, x_n\}$ and $\text{ran}(\sigma) = \{x'_1, \dots, x'_n\}$. If $x = x_j$ for some $j \in 1..n$, let $\sigma(x) = x'_j$. Otherwise, if $x \notin \text{dom}(\sigma)$, let $\sigma(x) = x$.
- *Barbed bisimilarity*, $\dot{\approx}$, is the relation on processes such that $P \dot{\approx} Q$ if and only if there is a barbed bisimulation \mathcal{S} such that $P \mathcal{S} Q$.
- *Barbed congruence*, \approx , is the relation on processes such that $P \approx Q$ if and only if for all processes R and renamings σ we have that $P\sigma \mid R \dot{\approx} Q\sigma \mid R$.

The following are basic properties of barbed congruence for the untyped π -calculus. As in the typed case, barbed congruence is a congruence relation preserved by renamings that includes structural congruence.

Proposition A.4

- (1) *Barbed congruence is reflexive, transitive, and symmetric.*
- (2) *Barbed congruence satisfies the congruence properties.*

- If $P \approx Q$ then $x(y_1, \dots, y_n).P \approx x(y_1, \dots, y_n).Q$.
 - If $P \approx Q$ then $P \mid R \approx Q \mid R$.
 - If $P \approx Q$ then $(\nu x)P \approx (\nu x)Q$.
 - If $P \approx Q$ then $!P \approx !Q$.
- (3) If $P \approx Q$ then $P\sigma \approx Q\sigma$ for any arbitrary substitution σ from names to names.
- (4) If $P \equiv Q$ then $P \approx Q$.
- (5) If $x \notin fv(P)$ then $(\nu x)P \approx P$.

Next, we prove a non-interference property for communications over a restricted channel. This property plays an important role in the soundness proof of the equational theory developed in Section 5.

Lemma A.5 (Non-Interference) *If $k \notin \{z\} \cup fv(P)$, then:*

$$(\nu k)(\bar{k}\langle z \mid k(x).P) \approx P\{x \leftarrow z\}$$

Proof Let \mathcal{S} be the smallest reflexive and symmetric relation such that:

$$(\nu k)(\bar{k}\langle z \mid k(x).P) \mid R \mathcal{S} (\nu k)(P\{x \leftarrow z\}) \mid R$$

for all processes P and R such that $k \notin \{z\} \cup fv(P)$. We show \mathcal{S} is a barbed bisimulation. Assume $P_1 \mathcal{S} P_2$. Then either P_1 equals P_2 , which is a trivial case, or $P_1 = (\nu k)(P\{x \leftarrow z\}) \mid R$ and $P_2 = (\nu k)(\bar{k}\langle z \mid k(x).P) \mid R$, or we have the symmetric case, $P_1 = (\nu k)(\bar{k}\langle z \mid k(x).P) \mid R$ and $P_2 = (\nu k)(P\{x \leftarrow z\}) \mid R$. Assume we are in the latter case.

- (1) Assume $P_1 \downarrow \bar{x}$. By Proposition A.3, it must be the case that $R \downarrow \bar{x}$. Hence, $P_2 \downarrow \bar{x}$.
- (2) Assume $P_2 \downarrow \bar{x}$. Since $P_1 \rightarrow P_2$, we have $P_2 \downarrow \bar{x}$.
- (3) Assume $P_1 \rightarrow Q_1$. By Proposition A.1 and inspection of the possible transitions, either $Q_1 \equiv P_2$, or there is a process R' such that $R \rightarrow R'$ and $Q_1 \equiv P_1 \mid R'$. In the former case, since \mathcal{S} is reflexive, we get that $Q_1 \equiv \mathcal{S} \equiv P_2$. In the latter case, let Q_2 be the process $(\nu k)(P\{x \leftarrow z\}) \mid R'$. Since $P_1 \rightarrow Q_2$, we get that $P_2 \rightarrow Q_2$ and $Q_1 \equiv \mathcal{S} \equiv Q_2$, as required.
- (4) Assume $P_2 \rightarrow Q_2$. Since $P_1 \rightarrow P_2$, we get that $P_1 \rightarrow^* Q_1$ and, by \mathcal{S} reflexive, we get that $Q_1 \equiv \mathcal{S} \equiv Q_1$, as required.

Since \mathcal{S} is a barbed bisimulation, we get that

$$(\nu k)(\bar{k}\langle z \mid k(x).P) \mid R \overset{\bullet}{\approx} (\nu k)(P\{x \leftarrow z\}) \mid R$$

for all processes P and R such that $k \notin \{z\} \cup fv(P)$.

Let R be an arbitrary process and σ be an arbitrary renaming. We can assume that the bound names of $(\nu k)(P\{x \leftarrow z\})$ are not in $\text{dom}(\sigma) \cup \text{ran}(\sigma)$. Therefore:

$$\begin{aligned} (\nu k)(\bar{k}\langle z \rangle \mid k(x).P)\sigma \mid R &\equiv (\nu k)(\bar{k}\langle \sigma(z) \rangle \mid k(x).(P\sigma)) \mid R \\ &\stackrel{\bullet}{\approx} (\nu k)(P\sigma\{x \leftarrow \sigma(z)\}) \mid R \\ &= (\nu k)(P\{x \leftarrow z\}\sigma) \mid R \end{aligned}$$

Hence, $(\nu k)(\bar{k}\langle z \rangle \mid k(x).P) \approx (\nu k)(P\{x \leftarrow z\})$. By Proposition A.4 (5), we get that $(\nu k)(\bar{k}\langle z \rangle \mid k(x).P) \approx P\{x \leftarrow z\}$. \square

We can prove the following algebraic laws by a similar method.

Lemma A.6 *If $k \notin \{x, y_1, \dots, y_n\}$ then:*

$$x(y_1, \dots, y_n).(\nu k)P \approx (\nu k)x(y_1, \dots, y_n).P$$

Lemma A.7 (Garbage Collection) *If $k \notin \text{fv}(Q)$ then:*

- (1) $(\nu k)(k(x_1, \dots, x_n).P \mid Q) \approx Q$
- (2) $(\nu k)(!k(x_1, \dots, x_n).P \mid Q) \approx Q$.

Assume p does not appear free in input position in P, Q , that is, $p \notin \text{inp}(P) \cup \text{inp}(Q)$, let the operator $\text{def } p(x, k) = P \text{ in } Q$ denote the process $(\nu p)(!p(x, k).P \mid Q)$. Such processes are found in encodings of the λ -calculus in the π -calculus and also in our encoding of the region calculus. For example, $\text{erase}([\lambda(x:A)b \text{ at } \rho]k)$ can be rewritten $\text{def } p(x, k) = \llbracket b \rrbracket k \text{ in } \bar{k}\langle p \rangle$.

Replicated Resources:

For all processes P and Q , such that $x \notin \text{inp}(P) \cup \text{inp}(Q)$, we define the process $\text{def } x(y_1, \dots, y_n) = P \text{ in } Q$ to be $(\nu x)(!x(y_1, \dots, y_n).P \mid Q)$.

One of the algebraic laws valid in the local π -calculus and not in the full π -calculus is the *replication theorem* of Milner that, intuitively, states that private resources can be safely duplicated, that is, for example:

$$\begin{aligned} \text{def } x(y_1, \dots, y_n) = P \text{ in } (Q \mid R) &\approx \\ (\text{def } x(y_1, \dots, y_n) = P \text{ in } Q) \mid (\text{def } x(y_1, \dots, y_n) = P \text{ in } R) & \end{aligned}$$

We state below Proposition A.8, which lists a more complete set of replication laws. An equivalent of this property has been proved for the full π -calculus (Milner 1999), where the equivalence used is strong ground congruence (Milner, Parrow, and Walker 1992). But this equality holds only with the side condition that the link to the resource (the channel x in this example) may not be emitted, that is, does not appear in object position of an output. Only recently, Merro and Sangiorgi (Merro and Sangiorgi 1998) proved the same equation, without the first side condition, for barbed congruence in the local π -calculus. The same laws have also been proved for a local variant of the blue calculus (Dal Zilio 1999).

Proposition A.8 (Replication Laws)

- (1) If $p \notin \text{fv}(P)$ then $\text{def } p(\vec{y}) = R \text{ in } P \approx P$
- (2) $\text{def } p(\vec{y}) = R \text{ in } (P \mid Q) \approx (\text{def } p(\vec{y}) = R \text{ in } P) \mid (\text{def } p(\vec{y}) = R \text{ in } Q)$
- (3) If $p \neq q$ and $q \notin \text{fv}(R)$ then:

$$\begin{aligned} & \text{def } p(\vec{y}) = R \text{ in } (\text{def } q(\vec{z}) = S \text{ in } P) \\ & \approx \text{def } q(\vec{z}) = (\text{def } p(\vec{y}) = R \text{ in } S) \text{ in } (\text{def } p(\vec{y}) = R \text{ in } P) \end{aligned}$$

- (4) If $\{\vec{z}\} \cap \text{fv}(p(\vec{y}).R) = \emptyset$ then:

$$x(\vec{z}).(\text{def } p(\vec{y}) = R \text{ in } P) \approx \text{def } p(\vec{y}) = R \text{ in } x(\vec{z}).P$$

- (5) $\text{def } p(y_1, \dots, y_n) = P \text{ in } (\overline{p}(z_1, \dots, z_n) \mid Q)$
 $\approx \text{def } p(y_1, \dots, y_n) = P \text{ in } (P\{y_1 \leftarrow z_1\} \cdots \{y_n \leftarrow z_n\} \mid Q)$

We also prove another useful property of barbed congruence.

Lemma A.9 *If $x \notin \text{fv}(k'(y).Q)$ then:*

$$(\nu k')(k(x).(P \mid k'(y).Q)) \approx (\nu k')(k(x).P \mid k'(y).Q)$$

Proof To show that $(\nu k')(k(x).(P \mid k'(y).Q))$ and $(\nu k')(k(x).P \mid k'(y).Q)$ are barbed congruent, we define a barbed bisimulation, \mathcal{S} , such that:

$$(\nu \vec{z})(\nu k')(k(x).(P \mid k'(y).Q)) \mid R \ \mathcal{S} \ (\nu \vec{z})(\nu k')(k(x).P \mid k'(y).Q) \mid R \quad (1)$$

Let \mathcal{S} be the smallest reflexive and symmetric relation such that (1) holds for all untyped processes P, Q, R and sequences of names \vec{z} where $x \notin \text{fv}(k'(y).Q)$.

We show that \mathcal{S} is a barbed bisimulation. Assume $P_1 \mathcal{S} P_2$. Then either P_1 equals P_2 , which is a trivial case, or $P_1 = (\nu \vec{z})((\nu k')(k(x).P \mid k'(y).Q) \mid R)$ and $P_2 = (\nu \vec{z})((\nu k')k(x).(P \mid k'(y).Q) \mid R)$, or we have the symmetric case, $P_1 = (\nu \vec{z})((\nu k')k(x).(P \mid k'(y).Q) \mid R)$ and $P_2 = (\nu \vec{z})((\nu k')(k(x).P \mid k'(y).Q) \mid R)$. Assume we are in the latter case.

- (1) Suppose $P_1 \downarrow \overline{p}$. By Proposition A.3, $R \downarrow \overline{p}$ with $p \notin \{\vec{z}\}$. By (Barb Par) and (Barb Res) several times, $P_2 \downarrow \overline{p}$, as required.
- (2) Suppose $P_2 \downarrow \overline{p}$. By Proposition A.3, $R \downarrow \overline{p}$ with $p \notin \{\vec{z}\}$. By (Barb Par) and (Barb Res) several times, $P_1 \downarrow \overline{p}$, as required.
- (3) Suppose $P_1 \rightarrow S$. By Proposition A.1 and inspection of the possible transitions, there is a sequence of names, \vec{w} , such that $R \equiv (\nu \vec{w})(\overline{k}(p) \mid R')$ and $S \equiv (\nu \vec{z}, \vec{w})((\nu k')(P\{x \leftarrow p\} \mid k'(y).Q) \mid R')$. By (Red \equiv) and (Red Interact), $P_2 \rightarrow S$. Since \mathcal{S} is reflexive, we get that $S \mathcal{S} S$, as required.

- (4) Suppose $P_2 \rightarrow S$. By Proposition A.1 and inspection of the possible transitions, there is a sequence of names, \vec{w} , such that $R \equiv (\nu \vec{w})(\bar{k}\langle p \rangle \mid R')$ and $S \equiv (\nu \vec{z}, \vec{w})(\nu k')(P\{x \leftarrow p\} \mid k'(y).Q) \mid R'$. By (Red \equiv) and (Red Interact), and since $x \notin fv(k'(y).Q)$, we get that $P_1 \rightarrow S$. Since \mathcal{S} is reflexive, we get that $S \mathcal{S} S$, as required.

Since \mathcal{S} is a barbed bisimulation, we get that $(\nu k')k(x).(P \mid k'(y).Q) \mid R \overset{\bullet}{\approx} (\nu k')(k(x).P \mid k'(y).Q) \mid R$, for all untyped processes P, Q, R such that $x \notin fv(k'(y).Q)$. Let σ be an arbitrary substitution of names for names. We can assume that $(dom(\sigma) \cup ran(\sigma)) \cap \{k', x, y\} = \emptyset$, and therefore:

$$\begin{aligned} (\nu k')k(x).(P \mid k'(y).Q)\sigma \mid R &\equiv (\nu k')k(x).(P\sigma \mid k'(y).Q\sigma) \mid R \\ (\nu k')(k(x).P \mid k'(y).Q)\sigma \mid R &\equiv (\nu k')(k(x).P\sigma \mid k'(y).Q\sigma) \mid R \end{aligned}$$

where $x \notin fv(k'(y).Q\sigma)$. Hence, $(\nu k')k(x).(P \mid k'(y).Q) \approx (\nu k')(k(x).P \mid k'(y).Q)$, as required. \square

B Proofs

In this appendix, we prove all the propositions stated without proof in the main body of the paper. We split the appendix into several sections. Throughout, with the exception of Appendix B.7, we work with the extended calculi of Section 6. Proofs of all the corresponding theorems for the unextended calculi may be obtained by simplifying the proofs for the extended calculi.

- (1) In Appendix B.1 we prove Theorem B.13, the subject reduction property for the extended region calculus, and Proposition B.14, the property that well-typed configurations do not lead to runtime errors. These facts correspond to Theorem 2.1 and Proposition 2.2, respectively, for the unextended region calculus.
- (2) In Appendix B.2, we prove Proposition B.31, the subject reduction property for our extended π -calculus, and Proposition B.32, effect soundness, the property that the group of any barb of a process is included in its effect. These facts correspond to Proposition 3.2 and Proposition 3.3, respectively, for the unextended π -calculus.
- (3) In Appendix B.3, we prove Proposition B.38, which asserts that the reductions of a typed process according to the typed operational semantics are equivalent to the reductions of the untyped erasure of the process according to the untyped operational semantics. This fact corresponds to Proposition 3.1 for the unextended π -calculus.
- (4) In Appendix B.4, we prove Proposition B.46, that barbed congruence for the extended π -calculus satisfies the congruence properties. This fact corresponds to Proposition 3.4 (2), for the unextended π -calculus.

- (5) In Appendix B.5, we prove Theorem B.53, the garbage collection principle for our extended π -calculus. This fact corresponds to Theorem 4.3 for the unextended π -calculus.
- (6) In Appendix B.6 we prove various properties of the encoding of the region calculus in the π -calculus.
- Appendix B.6.1 proves Theorem B.59, which asserts that the encoding preserves the static semantics.
- Appendix B.6.2 introduces an auxiliary small-step semantics for the region calculus.
- Appendix B.6.3 exploits the auxiliary small-step semantics in order to prove Theorem B.67, which asserts that the encoding preserves the dynamic semantics.
- Appendix B.6.4 proves Theorem B.68, which asserts that defunct regions make no difference to the behaviour of a program.
- The three theorems proved in this appendix correspond to Theorem 4.1, Theorem 4.2, and Theorem 4.4, respectively, for the unextended calculi.
- (7) In Appendix B.7, we prove the auxiliary lemma, Lemma 5.1, and the soundness of the equational theory for the unextended region calculus, Theorem 5.2, as stated in Section 5.

B.1 Subject Reduction for the λ -Calculus

In this section, we prove Theorem B.13, that in the extended region calculus reduction preserves types. We also prove Proposition B.14, that well-typed values are allocated in lived regions and that well-typed function applications invoke closures stored in lived regions. These two properties combined imply that a well-typed expressions cannot yield a runtime errors.

The proof of these properties use a series of simple intermediate results, Lemmas B.1 to B.12, that are classic in proof of subject reduction. For example, Lemma B.3, that is an example of so-called exchange lemma, Lemma B.6, that is an example of weakening lemma, or Lemmas B.11 and B.12, that are examples of substitutions lemmas. For the sake of brevity, we omit the proofs of these intermediate properties.

In the type and effect system introduced in Sections 2.3 and 6.1, each judgment has the form $E \vdash \mathcal{J}$, where E is a typing judgment and \mathcal{J} is an *assertion* that is either \diamond , for well-formed environments, or a type A , for well-formed types, or $a :^e A$, for good expressions a with type A and effect e . In the rest of this paper, we use the symbol \mathcal{J} to denote an assertion.

Lemma B.1 *If $E \vdash \mathcal{J}$ then $E \vdash \diamond$.*

Lemma B.2 *If $E, E' \vdash \mathcal{J}$ then $\text{dom}(E) \cap \text{dom}(E') = \emptyset$ and $E \vdash \diamond$.*

Lemma B.3 *If $E, x:A, \rho, E' \vdash \mathcal{J}$ then $E, \rho, x:A, E' \vdash \mathcal{J}$.*

Lemma B.4 *If $E, E' \vdash \mathcal{J}$, $x \notin \text{dom}(E, E') \cup L$ and $E \vdash A$ then $E, x:A, E' \vdash \mathcal{J}$.*

Lemma B.5 *If $E, E' \vdash \mathcal{J}$ and $\rho \notin \text{dom}(E, E')$ then $E, \rho, E' \vdash \mathcal{J}$.*

Lemma B.6 *If $E \vdash \mathcal{J}$ and $E, E' \vdash \diamond$ then $E, E' \vdash \mathcal{J}$.*

Lemma B.7 *Let $(p_i:A_i)_{i \in 1..n}$ be the environment $\text{ptr}(H)$. If $H \models \diamond$ then $\text{dom}(H) \vdash A_i$ for each $i \in 1..n$.*

Lemma B.8 *If $H \models \diamond$ and $H \asymp H'$ and $H' \models \diamond$ then $H + H' \models \diamond$.*

Lemma B.9 *If $\text{env}(H) \vdash \mathcal{J}$ and $H \asymp H'$ and $H' \models \diamond$ then $\text{env}(H + H') \vdash \mathcal{J}$.*

Lemma B.10 *If $E \vdash x :^e A$ and $E \vdash x :^{e'} B$ then $e = e' = \emptyset$ and $A = B$.*

Lemma B.11 *If $E, x:A, E' \vdash \mathcal{J}$ and $E \vdash p :^\emptyset A$ then $E, E' \vdash \mathcal{J}\{x \leftarrow p\}$.*

Lemma B.12 *If $E, \rho, E' \vdash \mathcal{J}$ and ρ' is a region defined in $\text{dom}(E)$ then $E, E'\{\rho \leftarrow \rho'\} \vdash \mathcal{J}\{\rho \leftarrow \rho'\}$.*

The following is the subject reduction theorem for our extended region calculus. A proof of Theorem 2.1, subject reduction for the unextended region calculus, can be obtained by simplifying the following proof.

Theorem B.13 *If $H \models S \cdot (a, h) : A$ and $S \cdot (a, h) \Downarrow (p', h')$ there is H' such that $H \asymp H'$ and $H + H' \models S \cdot (p', h') : A$.*

Proof By induction on the derivation of $S \cdot (a, h) \Downarrow (p', h')$.

(Eval Var) Then $S \cdot (p, h) \Downarrow (p, h)$, and we have $H \models S \cdot (p, h) : A$ by assumption. Take $H' = \emptyset$ and we trivially have $H \asymp H'$ and $H + H' \models S \cdot (p, h) : A$.

(Eval Alloc) Then $S \cdot (v \text{ at } \rho, h) \Downarrow (p, h + (\rho \mapsto (h(\rho) + (p \mapsto v))))$ with $\rho \in S$ and $p \notin \text{dom}_2(h)$.

By (Config Good), $H \models S \cdot (v \text{ at } \rho, h) : A$ means that $\text{env}(H) \vdash v \text{ at } \rho :^e A$ for some $e \subseteq S$, and that $H \models h$ and $S \subseteq \text{dom}(H)$. Since only (Exp Nil), (Exp Cons) or (Exp Fun) can derive $\text{env}(H) \vdash v \text{ at } \rho :^e A$, we have $A = V \text{ at } \rho$, for some V , and $e = \{\rho\}$.

Let H' be the heap typing $\rho \mapsto (p:A)$. Since $p \notin \text{dom}_2(H) = \text{dom}_2(h)$, we have $\text{env}(H + H') \vdash \diamond$. Hence, $\text{env}(H + H') \vdash p :^\emptyset A$. Moreover $S \subseteq \text{dom}(H + H')$.

By (Heap Good), $H \models h$ and $\rho \in S$ imply that $\text{env}(H) \vdash h(\rho) \text{ at } \rho : H(\rho)$. Therefore, $\text{env}(H + H') \vdash h(\rho) \text{ at } \rho + (p \mapsto v \text{ at } \rho) : H(\rho) + (p:A)$. Hence, $H + H' \models h + \rho \mapsto (h(\rho) + p \mapsto v)$.

We have $\text{env}(H + H') \vdash p \text{ at } \rho : A$ and $S \subseteq \text{dom}(H + H')$ and $H + H' \models h + \rho \mapsto (h(\rho) + p \mapsto v)$. Hence, by (Config Good), $H + H' \models S \cdot (p, h) : A$, as required.

(Eval Appl) Then $S \cdot (p[\rho'_1, \dots, \rho'_n](q), h) \Downarrow (p', h')$ derives from $S \cdot (b\{f \leftarrow p\} \{x \leftarrow q\} \sigma, h) \Downarrow (p', h')$ where $\rho \in S$ and $h(\rho)(p)$ is the function $\mu(f:F)\lambda[\rho_1, \dots, \rho_n](x)b$ and $\sigma = \{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$ and $F = \forall[\rho_1, \dots, \rho_n](B_1 \xrightarrow{e'} B_2)$ at ρ .

By (Config Good), $H \models S \cdot (p[\rho'_1, \dots, \rho'_n](q), h) : A$ means that $\text{env}(H) \vdash p[\rho'_1, \dots, \rho'_n](q) :^e A$ for some $e \subseteq S$, and that $H \models h$ and $S \subseteq \text{dom}(H)$.

Only (Exp Appl) can derive $\text{env}(H) \vdash p[\rho'_1, \dots, \rho'_n](q) :^e A$ and so we have $\text{env}(H) \vdash p :^\varnothing F$ and $\text{env}(H) \vdash q :^\varnothing B_1 \sigma$ and $A = B_2 \sigma$, where $e = \{\rho\} \cup e' \sigma$ and $\{\rho'_1, \dots, \rho'_n\} \subseteq \text{dom}(H)$. Since $H \models h$ and $\rho \in \text{dom}(H)$, we have that $\text{env}(H) \vdash h(\rho)$ at $\rho : H(\rho)$, and in particular, $\text{env}(H) \vdash \mu(f:F)\lambda[\rho_1, \dots, \rho_n](x)b$ at $\rho : \{\rho\} F$.

Only (Exp Fun) can derive $\text{env}(H) \vdash \mu(f:F)\lambda[\rho_1, \dots, \rho_n](x)b$ at $\rho : \{\rho\} F$, and so $\text{env}(H), f:F, \rho_1, \dots, \rho_n, x:B_1 \vdash b :^{e''} B_2$ where $e'' \subseteq e' \subseteq \text{dom}(E, \rho_1, \dots, \rho_n)$. By Lemma B.11 and B.12, since $\text{env}(H) \vdash q :^\varnothing B_1 \sigma$ and $\text{env}(H) \vdash p :^\varnothing F$, we get that $\text{env}(H) \vdash b\{f \leftarrow p\} \{x \leftarrow q\} \sigma :^{e'' \sigma} B_2 \sigma$. By (Config Good), $H \models S \cdot (b\{f \leftarrow p\} \{x \leftarrow q\} \sigma, h) : A$.

By induction hypothesis, since $S \cdot (b\{f \leftarrow p\} \{x \leftarrow q\} \sigma, h) \Downarrow (p', h')$, we get that there is H' with $H \asymp H'$ and $H + H' \models S \cdot (p', h') : A$, as required.

(Eval Let) Then $S \cdot (\text{let } x = b \text{ in } a, h) \Downarrow (p'', h'')$ derives from $S \cdot (b, h) \Downarrow (p', h')$ and $S \cdot (a\{x \leftarrow p'\}, h') \Downarrow (p'', h'')$.

By (Config Good), $H \models S \cdot (\text{let } x = b \text{ in } a, h) : A$ means that $\text{env}(H) \vdash \text{let } x = b \text{ in } a :^e A$ for some $e \subseteq S$, and that $H \models h$ and $S \subseteq \text{dom}(H)$.

Only (Exp Let) can derive $\text{env}(H) \vdash \text{let } x = b \text{ in } a :^e A$ and so we have $\text{env}(H) \vdash b :^{e_b} B$ and $\text{env}(H), x:B \vdash a :^{e_a} A$ for some e_b, e_a and B , such that $e = e_a \cup e_b$. By (Config Good), $\text{env}(H) \vdash b :^{e_b} B$ and $e_b \subseteq S$ and $H \models h$ and $S \subseteq \text{dom}(H)$ imply that $H \models S \cdot (b, h) : B$.

By induction hypothesis, since $S \cdot (b, h) \Downarrow (p', h')$, we get that there is H' with $H \asymp H'$ and $H + H' \models S \cdot (p', h') : B$. By (Config Good), this means that $\text{env}(H + H') \vdash p' :^{e'} B$ for some $e' \subseteq S$, and that $H + H' \models h'$ and $S \subseteq \text{dom}(H + H')$. Only (Exp x) or (Exp l) can derive this and so it must be that $e' = \emptyset$.

By Lemma B.9, since $\text{env}(H), x:B \vdash a :^{e_a} A$, we get that $\text{env}(H + H'), x:B \vdash a :^{e_a} A$. By Lemma B.11, since $\text{env}(H + H') \vdash p' :^\varnothing B$, we get that $\text{env}(H + H') \vdash a\{x \leftarrow p'\} :^{e_a} A$. Therefore, by (Config Good), $\text{env}(H + H') \models S \cdot (a\{x \leftarrow p'\}, h') : A$.

By induction hypothesis, since $S \cdot (a\{x \leftarrow p'\}, h') \Downarrow (p'', h'')$, we get that there is H'' such that $H + H' \asymp H''$ and $(H + H') + H'' \models S \cdot (p'', h'') : A$. To complete the case, note that $H \asymp H' + H''$ and $H + (H' + H'') \models S \cdot (p'', h'') : A$.

(Eval Letregion) Then $S \cdot (\text{letregion } \rho \text{ in } a, h) \Downarrow (p', h')$ derives from $S \cup \{\rho\} \cdot (a, h + \rho \mapsto \emptyset) \Downarrow (p', h')$ with $\rho \notin \text{dom}(h)$.

By (Config Good), $H \models S \cdot (\text{letregion } \rho \text{ in } a, h) : A$ means that $\text{env}(H) \vdash \text{letregion } \rho \text{ in } a :^e A$ and $H \models h$ for some $e \subseteq S$.

Only (Exp Letregion) can derive this and so we have $\text{env}(H), \rho \vdash a :^{e'} A$ with $\text{env}(H) \vdash A$ and $e = e' - \{\rho\}$. In particular $\rho \notin \text{dom}(H)$.

Let H' be the heap typing ($\rho \mapsto \emptyset$). We have $H \simeq H'$ and $\text{env}(H + H') = \text{env}(H), \rho$. By (Config Good), $\text{env}(H + H') \vdash a :^{e'} A$, and $e' \subseteq S \cup \{\rho\}$ and $S \cup \{\rho\} \subseteq \text{dom}(H + H')$ and $H + H' \models h + \rho \mapsto \emptyset$ imply that $H + H' \models (S \cup \{\rho\}) \cdot (a, h + \rho \mapsto \emptyset) : A$.

By induction hypothesis, since $(S \cup \{\rho\}) \cdot (a, h + \rho \mapsto \emptyset) \Downarrow (p', h')$, we get that there is H'' with $H + H' \simeq H''$ and $(H + H') + H'' \models (S \cup \{\rho\}) \cdot (p', h') : A$. To complete the case, note that $H \simeq H' + H''$ and $H + (H' + H'') \models S \cdot (p', h') : A$.

(Eval Case 1) Then $S \cdot (\text{case } p \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) \Downarrow (p', h')$. This derives from $S \cdot (b_1, h) \Downarrow (p', h')$ with $\rho \in S$ and $h(\rho)(p) = \text{nil}$.

By (Config Good), $H \models S \cdot (\text{case } p \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) : A$ means that $\text{env}(H) \vdash \text{case } p \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2 :^e A$ and $H \models h$ for some $e \subseteq S$.

Only (Exp Case) can derive this and so we have $\text{env}(H) \vdash p :^\emptyset [B]$ at ρ and $\text{env}(H) \vdash b_1 :^{e_1} A$ and $\text{env}(H), y_1:B, y_2:[B]$ at $\rho \vdash b_2 :^{e_2} A$.

By induction hypothesis, since $S \cdot (b_1, h) \Downarrow (p', h')$, we get that there is H' with $H \simeq H'$ and $H + H' \models S \cdot (p', h') : A$, as required.

(Eval Case 2) Then $S \cdot (\text{case } p \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) \Downarrow (p', h')$. This derives from $S \cdot (b_2\{y_1 \leftarrow q_1\}\{y_2 \leftarrow q_2\}, h) \Downarrow (p', h')$ with $\rho \in S$ and $h(\rho)(p) = q_1 :: q_2$.

By (Config Good), $H \models S \cdot (\text{case } p \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) : A$ means that $\text{env}(H) \vdash \text{case } p \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2 :^e A$ and $H \models h$ for some $e \subseteq S$.

Only (Exp Case) can derive this and so we have $\text{env}(H) \vdash p :^\emptyset [B]$ at ρ and $\text{env}(H) \vdash b_1 :^{e_1} A$ and $\text{env}(H), y_1:B, y_2:[B]$ at $\rho \vdash b_2 :^{e_2} A$.

Therefore, by hypothesis, $h(\rho)(p) = q_1 :: q_2$ and $\text{env}(H) \vdash p :^\emptyset [B]$ at ρ . Hence, by (Config Good), $\text{env}(H) \vdash (q_1 :: q_2)$ at $\rho :^{\{\rho\}} [B]$ at ρ . Only (Exp Cons) can derive this and so we have $E \vdash q_1 :^\emptyset B$ and $E \vdash q_2 :^\emptyset [B]$ at ρ .

By Lemma B.11, since $\text{env}(H), y_1:B, y_2:[B]$ at $\rho \vdash b_2 :^{e_2} A$, we get that $\text{env}(H) \vdash b_2\{y_1 \leftarrow q_1\}\{y_2 \leftarrow q_2\} :^{e_2} A$.

By induction hypothesis, since $S \cdot (b_2\{y_1 \leftarrow q_1\}\{y_2 \leftarrow q_2\}, h) \Downarrow (p', h')$, we get that there is H' with $H \simeq H'$ and $H + H' \models S \cdot (p', h') : A$, as required. \square

Next, we show that well-typed configurations avoid the runtime errors of allocation or invocation of a closure in a defunct region. A proof of Proposition 2.2, an equivalent property for the unextended region calculus, can be obtained by simplifying the following proof.

Proposition B.14

- (1) If $H \models S \cdot (v \text{ at } \rho, h) : A$ then $\rho \in S$.
- (2) If $H \models S \cdot (p[\rho'_1, \dots, \rho'_n](q), h) : A$ then there are ρ and v such that $\rho \in S$, $h(\rho)(p) = v$, and v is a function of the form $\mu(f:F)\lambda[\rho_1, \dots, \rho_n](x)b$, where F is the type $(\forall[\rho_1, \dots, \rho_n]B \xrightarrow{e} A)$ at ρ and there is e' such that $e' \subseteq e \subseteq \text{dom}(E, \rho_1, \dots, \rho_n)$ and $\text{env}(H), f:F, \rho_1, \dots, \rho_n, x:B \vdash b :^{e'} A$.

Proof For part (1), assume $H \models S \cdot (v \text{ at } \rho, h) : A$. By (Config Good) we get that $\text{env}(H) \vdash (v \text{ at } \rho) :^e A$ for some effect e , with $e \cup \text{fg}(A) \subseteq S$. Only (Exp Nil), (Exp Cons) or (Exp Fun) can derive this and so we have $e = \{\rho\}$. Hence, $\rho \in S$.

For part (2), assume $H \models S \cdot (p[\rho'_1, \dots, \rho'_n](q), h) : A$. By (Config Good) we get that $H \models h$ and that $\text{env}(H) \vdash p[\rho'_1, \dots, \rho'_n](q) :^e A$ for some effects e , with $e \cup \text{fg}(A) \subseteq S$. Only (Exp Appl) can derive this and so we have $\text{env}(H) \vdash p :^\varnothing F$ for some region ρ , with $F = (\forall[\rho_1, \dots, \rho_n](B \xrightarrow{e'} A))$ at ρ and $e = \{\rho\} \cup e' \{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$. Only (Exp x) can derive $\text{env}(H) \vdash p :^\varnothing F$ and so there is a region ρ' such that $H(\rho')(p) = F$. By (Heap Good), $\rho' = \rho$ and $\rho \in \text{dom}(H)$ and there is a value v such that $H(\rho)(p) = v$. By (Region Good), $\text{env}(H) \vdash v \text{ at } \rho : \{\rho\} F$. Only (Exp Fun) can derive this. Hence, v is a function of the form $\mu(f:F)\lambda[\rho_1, \dots, \rho_n](x)b$ and there is e'' such that $e'' \subseteq e' \subseteq \text{dom}(E, \rho_1, \dots, \rho_n)$ and $\text{env}(H), f:F, \rho_1, \dots, \rho_n, x:B \vdash b :^{e''} A$. \square

B.2 Subject Reduction for the π -Calculus

We show that reduction in the π -calculus preserves types and effects. Like in the previous section on subject reduction for the λ -calculus, we use intermediate results whose proofs are omitted for the sake of brevity. We will also use the symbol \mathcal{J} to denote an assertion, that is either \diamond , a type T , a channel typing $x : T$, or a process typing $P : \mathbf{H}$.

Lemma B.15 If $E \vdash P : \mathbf{H}$ then $\mathbf{H} \subseteq \text{dom}(E)$.

Lemma B.16 If $E \vdash \mathcal{J}$ then $E \vdash \diamond$.

Lemma B.17 If $E, E' \vdash \mathcal{J}$ then $\text{dom}(E) \cap \text{dom}(E') = \varnothing$ and $E \vdash \diamond$.

Lemma B.18 If $E, x:T, E' \vdash \mathcal{J}$ then $E \vdash T$.

Lemma B.19 If $E \vdash \mathcal{J}$ then $\text{fg}(\mathcal{J}) \subseteq \text{dom}(E)$.

Lemma B.20 If $E \vdash x : T_1$ and $E \vdash x : T_2$, where T_1 and T_2 are channel types of the form $G(H_1, \dots, H_m)[T'_1, \dots, T'_n] \setminus \mathbf{H}$, then $T_1 = T_2$.

Lemma B.21 If $E, x_1:T_1, x_2:T_2, E' \vdash \mathcal{J}$ then $E, x_2:T_2, x_1:T_1, E' \vdash \mathcal{J}$.

Lemma B.22 If $E, G, H, E' \vdash \mathcal{J}$ then $E, H, G, E' \vdash \mathcal{J}$.

Lemma B.23 *If $E, G, x:T, E' \vdash \mathcal{J}$ and $G \notin \text{fg}(T)$ then $E, x:T, G, E' \vdash \mathcal{J}$.*

Lemma B.24 *If $E, x:T, G, E' \vdash \mathcal{J}$ then $E, G, x:T, E' \vdash \mathcal{J}$.*

Lemma B.25 *If $E, x:T, E' \vdash \mathcal{J}$ and $x \notin \text{fn}(\mathcal{J})$ then $E, E' \vdash \mathcal{J}$.*

Lemma B.26 *If $E, G, E' \vdash \mathcal{J}$ and $G \notin \text{fg}(\mathcal{J}) \cup \text{dom}(E')$ then $E, E' \vdash \mathcal{J}$.*

Lemma B.27 *If $E \vdash \mathcal{J}$ and $E, E' \vdash \diamond$ then $E, E' \vdash \mathcal{J}$.*

Lemma B.28 *If $E, x:T, E' \vdash \mathcal{J}$ and $E \vdash y : T$ then $E, E' \vdash \mathcal{J}\{x \leftarrow y\}$.*

Lemma B.29 *If $E, G, E' \vdash \mathcal{J}$ and H is a group defined in $\text{dom}(E)$ then $E, E'\{G \leftarrow H\} \vdash \mathcal{J}\{G \leftarrow H\}$.*

Lemma B.30 *If $E \vdash P : \mathbf{H}$ and $P \equiv Q$ then $E \vdash Q : \mathbf{H}$.*

Proof The lemma follows by showing that $P \equiv Q$ implies:

(1) If $E \vdash P : \mathbf{H}$ then $E \vdash Q : \mathbf{H}$.

(2) If $E \vdash Q : \mathbf{H}$ then $E \vdash P : \mathbf{H}$.

We proceed by induction on the derivation of $P \equiv Q$.

(Struct Refl) Trivial.

(Struct Symm) Then $Q \equiv P$. For (1), assume $E \vdash P : \mathbf{H}$. By induction hypothesis (2), $Q \equiv P$ implies that $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Trans) Then there is R such that $P \equiv R$ and $R \equiv Q$. For (1), assume $E \vdash P : \mathbf{H}$. By induction hypothesis (1), $E \vdash R : \mathbf{H}$. Again, by induction hypothesis (1), $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Res) Then $P = (\nu x:T)P'$ and $Q = (\nu x:T)Q'$ for some P', Q' , with $P' \equiv Q'$. For (1), assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Res), with $E, x:T \vdash P' : \mathbf{H}$. By induction hypothesis (1), $E, x:T \vdash Q' : \mathbf{H}$. By (Proc Res), $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct GRes) Then $P = (\nu G)P'$ and $Q = (\nu G)Q'$ for some P', Q' , with $P' \equiv Q'$. For (1), assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc GRes), with $E, G \vdash P' : \mathbf{G}$ where $\mathbf{H} = \mathbf{G} - \{G\}$. By induction hypothesis (1), $E, G \vdash Q' : \mathbf{G}$. By (Proc GRes), $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Par) Then $P = P' \mid R$ and $Q = Q' \mid R$ for some P', Q', R , with $P' \equiv Q'$. For (1), assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Par), with $E \vdash P' : \mathbf{H}'$ and $E \vdash R : \mathbf{H}''$, where $\mathbf{H} = \mathbf{H}' \cup \mathbf{H}''$. By induction hypothesis (1), $E \vdash Q' : \mathbf{H}'$. By (Proc Par), $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Repl) Then $P = !P'$ and $Q = !Q'$ for some P', Q' , with $P' \equiv Q'$. For (1), assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Repl), with $E \vdash P' : \mathbf{H}$. By induction hypothesis (1), $E \vdash Q' : \mathbf{H}$. By (Proc Repl), $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Input) Then $P = x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P'$ and $Q = x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).Q'$ for some P', Q' , with $P' \equiv Q'$. For (1), assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Input), with:

$$\begin{aligned} E \vdash x : G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{H}' \\ E, G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n \vdash P' : \mathbf{G} \end{aligned}$$

where $\mathbf{H} = \{G\} \cup (\mathbf{G} - \mathbf{H}')$ and $(\mathbf{G} - \mathbf{H}') \cap \{G_1, \dots, G_m\} = \emptyset$. By induction hypothesis (1), $E, G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n \vdash Q' : \mathbf{G}$. By (Proc Input), $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Par Zero) Then $P = Q \mid \mathbf{0}$.

For (1), assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Par), with $E \vdash Q : \mathbf{H}$ and $E \vdash \mathbf{0} : \emptyset$. Hence, $E \vdash Q : \mathbf{H}$.

For (2), assume $E \vdash Q : \mathbf{H}$. By Lemma B.16, $E \vdash \diamond$. By (Proc Zero), $E \vdash \mathbf{0} : \emptyset$. By (Proc Par), $E \vdash Q \mid \mathbf{0} : \mathbf{H}$, that is, $E \vdash P : \mathbf{H}$.

(Struct Par Comm) Then $P = P' \mid P''$ and $Q = P'' \mid P'$ for some P', P'' . For (1), assume $E \vdash P' \mid P'' : \mathbf{H}$. This must have been derived from (Proc Par), with $E \vdash P' : \mathbf{H}'$ and $E \vdash P'' : \mathbf{H}''$ where $\mathbf{H} = \mathbf{H}' \cup \mathbf{H}''$. By (Proc Par), $E \vdash P'' \mid P' : \mathbf{H}$. Hence, $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Par Assoc) Then $P = (P' \mid P'') \mid P'''$ and $Q = P' \mid (P'' \mid P''')$ for some P', P'', P''' . For (1), assume $E \vdash (P' \mid P'') \mid P''' : \mathbf{H}$. This must have been derived from (Proc Par), with $E \vdash (P' \mid P'') : \mathbf{H}_1$ and $E \vdash P''' : \mathbf{H}_2$, where $(\mathbf{H}_1 \cup \mathbf{H}_2) = \mathbf{H}$, followed by a number of subsumption steps implying $E \vdash (P' \mid P'') : \mathbf{H}_3$, where $\mathbf{H}_3 \subseteq \mathbf{H}_1$ by transitivity, and followed by (Proc Par), with $E \vdash P' : \mathbf{H}_4$, and $E \vdash P'' : \mathbf{H}_5$ where $(\mathbf{H}_4 \cup \mathbf{H}_5) = \mathbf{H}_3$. By (Proc Par) twice, $E \vdash P' \mid (P'' \mid P''') : \mathbf{H}_4 \cup (\mathbf{H}_5 \cup \mathbf{H}_2)$. By (Proc Subsum) and Lemma B.15, and since $\mathbf{H}_4 \cup (\mathbf{H}_5 \cup \mathbf{H}_2) = (\mathbf{H}_3 \cup \mathbf{H}_2) \subseteq (\mathbf{H}_1 \cup \mathbf{H}_2) = \mathbf{H}$, we get that $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Repl Par) Then $P = !R$ and $Q = R \mid !R$ for some R .

For (1), assume $E \vdash !R : \mathbf{H}$. This must have been derived from (Proc Repl), with $E \vdash R : \mathbf{H}$. By (Proc Par), $E \vdash R \mid !R : \mathbf{H} \cup \mathbf{H}$. Hence, $E \vdash Q : \mathbf{H}$.

For (2), assume $E \vdash R \mid !R : \mathbf{H}$. This must have been derived from (Proc Par), with $E \vdash R : \mathbf{H}'$ and $E \vdash !R : \mathbf{H}''$ where $\mathbf{H} = \mathbf{H}' \cup \mathbf{H}''$. Hence, $\mathbf{H}'' \subseteq \mathbf{H}$. By (Proc Subsum) and Lemma B.15, $E \vdash !R : \mathbf{H}$. Hence, $E \vdash P : \mathbf{H}$.

(Struct Res Res) Then $P = (\nu x_1:T_1)(\nu x_2:T_2)R$ and $Q = (\nu x_2:T_2)(\nu x_1:T_1)R$ for some R , with $x_1 \neq x_2$. For (1), assume $E \vdash (\nu x_1:T_1)(\nu x_2:T_2)R : \mathbf{H}$. This must have been derived from (Proc Res), with $E, x_1:T_1 \vdash (\nu x_2:T_2)R : \mathbf{H}$, followed by a number of subsumption steps implying $E \vdash (\nu x_2:T_2)R : \mathbf{H}'$, where $\mathbf{H}' \subseteq \mathbf{H}$ by transitivity, and followed by (Proc Res), with $E, x_1:T_1, x_2:T_2 \vdash R : \mathbf{H}'$. By Lemma B.21, $E, x_2:T_2, x_1:T_1 \vdash R : \mathbf{H}'$. By (Proc Res) twice, (Proc Subsum) and Lemma B.15 we have $E \vdash (\nu x_2:T_2)(\nu x_1:T_1)R : \mathbf{H}$. Hence, $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct Res Par) Then $P = (\nu x:T)(P' \mid P'')$ and $Q = P' \mid (\nu x:T)P''$ for some P', P'' , with $x \notin \text{fn}(P')$.

For (1), assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Res) with $E, x:T \vdash P' \mid P'' : \mathbf{H}$, followed by a number of subsumption steps implying $E, x:T \vdash P' \mid P'' : \mathbf{G}$, where $\mathbf{G} \subseteq \mathbf{H}$ by transitivity, and followed by (Proc Par), with $E, x:T \vdash P' : \mathbf{G}'$ and $E, x:T \vdash P'' : \mathbf{G}''$, where $\mathbf{G} = \mathbf{G}' \cup \mathbf{G}''$. By Lemma B.25, since $x \notin \text{fn}(P')$, we have $E \vdash P' : \mathbf{G}'$. By (Proc Res), $E \vdash (\nu x:T)P'' : \mathbf{G}''$. By (Proc Par) we have $E \vdash P' \mid (\nu x:T)P'' : \mathbf{G}$. By (Proc Subsum) and Lemma B.15, we get that $E \vdash Q : \mathbf{H}$.

For (2), assume $E \vdash Q : \mathbf{H}$. This must have been derived from (Proc Par), with $E \vdash P' : \mathbf{H}'$, $E \vdash (\nu x:T)P'' : \mathbf{H}''$ and $\mathbf{H} = \mathbf{H}' \cup \mathbf{H}''$, followed by a number of subsumption steps implying $E \vdash (\nu x:T)P'' : \mathbf{G}$, where $\mathbf{G} \subseteq \mathbf{H}''$ by transitivity, and followed by (Proc Res), with $E, x:T \vdash P'' : \mathbf{G}$. By Lemma B.17 and B.18 we have $x \notin \text{dom}(E)$ and $E \vdash T$. By Lemma B.27, $E, x:T \vdash P' : \mathbf{H}'$. By (Proc Par), $E, x:T \vdash P' \mid P'' : (\mathbf{H}' \cup \mathbf{G})$ where $(\mathbf{H}' \cup \mathbf{G}) \subseteq \mathbf{H}$. By (Proc Res), (Proc Subsum) and Lemma B.15, we get that $E \vdash (\nu x:T)(P' \mid P'') : \mathbf{H}$. Hence, $E \vdash P : \mathbf{H}$.

(Struct GRes GRes) Then $P = (\nu G_1)(\nu G_2)R$ and $Q = (\nu G_2)(\nu G_1)R$ for some R . We can consider that $G_1 \neq G_2$.

For (1), assume $E \vdash (\nu G_1)(\nu G_2)R : \mathbf{H}$. This must have been derived from (Proc GRes), with $E, G_1 \vdash (\nu G_2)R : \mathbf{H}'$ and $\mathbf{H} = \mathbf{H}' - \{G_1\}$, followed by a number of subsumption steps implying $E, G_1 \vdash (\nu G_2)R : \mathbf{H}''$, where $\mathbf{H}'' \subseteq \mathbf{H}'$ by transitivity, and followed by (Proc GRes), with $E, G_1, G_2 \vdash R : \mathbf{H}'''$ and $\mathbf{H}'' = \mathbf{H}''' - \{G_2\}$. By Lemma B.22, $E, G_2, G_1 \vdash R : \mathbf{H}'''$, where $\mathbf{H}''' - \{G_1, G_2\} \subseteq \mathbf{H}$. By (Proc Subsum) and (Proc GRes) twice we have $E \vdash (\nu G_2)(\nu G_1)R : \mathbf{H}$. Hence, $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct GRes Res) Then $P = (\nu G)(\nu x:T)R$ and $Q = (\nu x:T)(\nu G)R$ for some R , with $G \notin \text{fg}(T)$. For (1), assume $E \vdash (\nu G)(\nu x:T)R : \mathbf{H}$. This must have been derived from (Proc GRes), with $E, G \vdash (\nu x:T)R : \mathbf{H}'$ and $\mathbf{H} = \mathbf{H}' - \{G\}$, followed by a number of subsumption steps implying $E, G \vdash (\nu x:T)R : \mathbf{H}''$, where $\mathbf{H}'' \subseteq \mathbf{H}'$ by transitivity, and followed by (Proc Res), with $E, G, x:T \vdash R : \mathbf{H}''$. By Lemma B.23, $E, x:T, G \vdash R : \mathbf{H}''$. By (Proc GRes), (Proc Res), (Proc Subsum) and Lemma B.15, we get that $E \vdash Q : \mathbf{H}$. Part (2) is symmetric.

(Struct GRes Par) Then $P = (\nu G)(P' \mid P'')$ and $Q = P' \mid (\nu G)P''$ for some P', P'' , with $G \notin \text{fg}(P')$.

For (1), assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc GRes) with $E, G \vdash P' \mid P'' : \mathbf{H}'$ and $\mathbf{H} = \mathbf{H}' - \{G\}$, followed by a number of subsumption steps implying $E, G \vdash P' \mid P'' : \mathbf{G}$, where $\mathbf{G} \subseteq \mathbf{H}'$ by transitivity, and followed by (Proc Par) with $E, G \vdash P' : \mathbf{G}'$ and $E, G \vdash P'' : \mathbf{G}''$, where $\mathbf{G} = \mathbf{G}' \cup \mathbf{G}''$. By Lemma B.26, since $x \notin \text{fg}(P')$, we have $E \vdash P' : \mathbf{G}'$. By (Proc GRes), $E \vdash (\nu G)P'' : \mathbf{G}'' - \{G\}$. By (Proc Par) we have $E \vdash P' \mid (\nu G)P'' : \mathbf{G} - \{G\}$. By (Proc Subsum) and Lemma B.15, we get that $E \vdash Q : \mathbf{H}$.

For (2), assume $E \vdash Q : \mathbf{H}$. This must have been derived from (Proc Par), with $E \vdash P' : \mathbf{H}'$, $E \vdash (\nu G)P'' : \mathbf{H}''$ and $\mathbf{H} = \mathbf{H}' \cup \mathbf{H}''$, followed by a number of subsumption steps implying $E \vdash (\nu G)P'' : \mathbf{G}$, where $\mathbf{G} \subseteq \mathbf{H}''$ by transitivity, and followed by (Proc GRes), with $E, G \vdash P'' : \mathbf{G}'$, where $\mathbf{G} = \mathbf{G}' - \{G\}$. By Lemma B.17 we have $G \notin \text{dom}(E)$ and by Lemma B.19 $G \notin \mathbf{H}'$. By Lemma B.27, $E, G \vdash P' : \mathbf{H}'$. By (Proc Par), $E, G \vdash P' \mid P'' : (\mathbf{H}' \cup \mathbf{G}')$. By (Proc GRes), (Proc Subsum) and Lemma B.15, we get that $E \vdash (\nu G)(P' \mid P'') : \mathbf{H}$. Hence, $E \vdash P : \mathbf{H}$. \square

The following is the subject reduction property for our extended π -calculus. A proof of Proposition 3.2, subject reduction for the unextended π -calculus, can be obtained by simplifying the following proof.

Proposition B.31 *If $E \vdash P : \mathbf{H}$ and $P \rightarrow Q$ then $E \vdash Q : \mathbf{H}$.*

Proof By induction on the derivation of $P \rightarrow Q$.

(Red Interact) Then $P = \bar{x}(G'_1, \dots, G'_m, y'_1, \dots, y'_n) \mid x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P'$ and $Q = P' \sigma \{y_1 \leftarrow y'_1\} \cdots \{y_n \leftarrow y'_n\}$ where σ is the substitution $\{G_1 \leftarrow G'_1\} \cdots \{G_m \leftarrow G'_m\}$.

Assume $E \vdash P : \mathbf{H}$. By Lemma B.15, $\mathbf{H} \subseteq \text{dom}(E)$. The judgment $E \vdash P : \mathbf{H}$ must have been derived from (Proc Par), with $E \vdash \bar{x}(G'_1, \dots, G'_m, y'_1, \dots, y'_n) : \mathbf{H}_1$, and $E \vdash x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P' : \mathbf{H}_2$ where $\mathbf{H} = \mathbf{H}_1 \cup \mathbf{H}_2$. The former must have been derived from a number of subsumption steps implying $E \vdash \bar{x}(G'_1, \dots, G'_m, y'_1, \dots, y'_n) : \mathbf{H}_3$, where $\mathbf{H}_3 \subseteq \mathbf{H}_1$, followed by (Proc Output), with $E \vdash y'_1 : T_1 \sigma, \dots, E \vdash y'_n : T_n \sigma$, and $E \vdash x : G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$, and:

$$\{G\} \cup \mathbf{G} \sigma = \mathbf{H}_3 \quad (2)$$

By Lemma B.20, the latter must have been derived from (Proc Input), with $E, G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n \vdash P' : \mathbf{H}_4$ and $(\mathbf{H}_4 - \mathbf{G}) \cap \{G_1, \dots, G_m\} = \emptyset$, followed by a number of subsumption steps implying $\{G\} \cup (\mathbf{H}_4 - \mathbf{G}) \subseteq \mathbf{H}_2 \subseteq \text{dom}(E)$ by transitivity. In particular, we have that:

$$(\mathbf{H}_4 - \mathbf{G}) \sigma = (\mathbf{H}_4 - \mathbf{G}) \subseteq \mathbf{H}_2 \quad (3)$$

By Lemma B.19, since $E \vdash \bar{x}\langle G'_1, \dots, G'_m, y'_1, \dots, y'_n \rangle$, we have that $\{G'_1, \dots, G'_m\} \subseteq \text{dom}(E)$. Then, by Lemma B.29 several times, it follows that $E, y_1:T_1\sigma, \dots, y_n:T_n\sigma \vdash P'\sigma : \mathbf{H}_4\sigma$. By Lemma B.28, $E \vdash P'\sigma \{y_1 \leftarrow y'_1\} \cdots \{y_n \leftarrow y'_n\} : \mathbf{H}_4\sigma$.

By definition of set difference, $\mathbf{H}_4\sigma = (\mathbf{H}_4 - \mathbf{G})\sigma \cup \mathbf{G}\sigma$, and therefore $\mathbf{H}_4\sigma = (\{G\} \cup (\mathbf{H}_4 - \mathbf{G})\sigma) \cup (\{G\} \cup \mathbf{G})\sigma$. Using the different inclusions obtained in this item, and especially equations 2 and 3, we get that $\mathbf{H}_4\sigma \subseteq (\{G\} \cup (\mathbf{H}_4 - \mathbf{G})\sigma) \cup (\{G\} \cup \mathbf{G})\sigma \subseteq (\mathbf{H}_2 \cup \mathbf{H}_3) \subseteq (\mathbf{H}_2 \cup \mathbf{H}_1) = \mathbf{H} \subseteq \text{dom}(E)$. Then $E \vdash Q : \mathbf{H}$.

(Red Par) Then $P = P' \mid R$ and $Q = Q' \mid R$ for some P', Q', R such that $P' \rightarrow Q'$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Par), with $E \vdash P' : \mathbf{H}'$ and $E \vdash R : \mathbf{H}''$ where $\mathbf{H} = \mathbf{H}' \cup \mathbf{H}''$. By induction hypothesis $E \vdash Q' : \mathbf{H}'$. By (Proc Par), $E \vdash Q' \mid R : \mathbf{H}' \cup \mathbf{H}''$. Hence, $E \vdash Q : \mathbf{H}$.

(Red GRes) Then $P = (\nu G)P'$ and $Q = (\nu G)Q'$ for some P', Q' such that $P' \rightarrow Q'$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc GRes), with $E, G \vdash P' : \mathbf{H}'$ and $\mathbf{H} = \mathbf{H}' - \{G\}$. By induction hypothesis, $E, G \vdash Q' : \mathbf{H}'$. By (Proc GRes), $E \vdash Q : \mathbf{H}$.

(Red Res) Then $P = (\nu x:T)P'$ and $Q = (\nu x:T)Q'$ for some P', Q' such that $P' \rightarrow Q'$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Res), with $E, x:T \vdash P' : \mathbf{H}$. By induction hypothesis, $E, x:T \vdash Q' : \mathbf{H}$. By (Proc Res), $E \vdash Q : \mathbf{H}$.

(Red \equiv) Then $P \equiv P'$ and $Q \equiv Q'$ for some P', Q' such that $P' \rightarrow Q'$. Assume $E \vdash P : \mathbf{H}$. By Lemma B.30, $E \vdash P' : \mathbf{H}$. By induction hypothesis, $E \vdash Q' : \mathbf{H}$. By Lemma B.30, $E \vdash Q : \mathbf{H}$. \square

Next, we prove effect soundness for our extended π -calculus, the property that the group of any barb of a process is included in its effect. This fact correspond to Proposition 3.3 for the unextended π -calculus.

Proposition B.32 *If $E \vdash P : \mathbf{H}$ and $P \downarrow \beta$ with $\beta \in \{x, \bar{x}\}$ then there is a type $G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ such that $E \vdash x : G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ and $G \in \mathbf{H}$.*

Proof By induction on the derivation of $P \downarrow \beta$.

(Barb Input) Then $P = x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P'$ and $\beta = x$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Input) with $E \vdash P : \mathbf{H}_1$ and $E \vdash x : G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ and $G \in \mathbf{H}_1$ followed by a number of subsumption steps implying $\mathbf{H}_1 \subseteq \mathbf{H}$. Hence $G \in \mathbf{H}$.

(Barb Output) Then $P = \bar{x}\langle G'_1, \dots, G'_m, y'_1, \dots, y'_n \rangle$ and $\beta = \bar{x}$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Output) with $E \vdash P : \mathbf{H}_1$

and $E \vdash x : G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ and $\mathbf{H}_1 = \{G\} \cup \mathbf{G}\sigma$, where σ is the substitution $\{G_1 \leftarrow G'_1\} \cdots \{G_m \leftarrow G'_m\}$, followed by a number of subsumption steps implying $\mathbf{H}_1 \subseteq \mathbf{H}$. Hence, $G \in \mathbf{H}$.

(Barb GRes) Then $P = (\nu G')P'$ for some P' such that $P' \downarrow \beta$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc GRes) with $E \vdash P : \mathbf{H}_1$ and $E, G' \vdash P' : \mathbf{H}_2$ and $\mathbf{H}_1 = \mathbf{H}_2 - \{G'\}$, followed by a number of subsumption steps implying $\mathbf{H}_1 \subseteq \mathbf{H}$. By induction hypothesis, there is a type $W \triangleq G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ such that $E \vdash x : W$ and $G \in \mathbf{H}_2$. Hence, $G \in \mathbf{H}$.

(Barb Res) Then $P = (\nu y:T)P'$ for some y, P' such that $x \neq y$ and $P' \downarrow \beta$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Res) with $E \vdash P : \mathbf{H}_1$ and $E, y:T \vdash P' : \mathbf{H}_1$, followed by a number of subsumption steps implying $\mathbf{H}_1 \subseteq \mathbf{H}$. By induction hypothesis, there is a type $W \triangleq G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ such that $E, y:T \vdash x : W$ and $G \in \mathbf{H}_1$. Hence $G \in \mathbf{H}$. By Lemma B.25, $E \vdash x : W$.

(Barb Par) Then $P = (P' \mid P'')$ with $P' \downarrow \beta$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Par) with $E \vdash P : \mathbf{H}_1$, and $E \vdash P' : \mathbf{H}'$, and $E \vdash P'' : \mathbf{H}''$, and $\mathbf{H}_1 = \mathbf{H}' \cup \mathbf{H}''$, followed by a number of subsumption steps implying $\mathbf{H}_1 \subseteq \mathbf{H}$. By induction hypothesis, there is a type $W \triangleq G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ such that $E \vdash x : W$ and $G \in \mathbf{H}'$. Hence, $G \in \mathbf{H}$.

(Barb \equiv) Then $P \equiv P'$ for some P' such that $P' \downarrow \beta$. Assume $E \vdash P : \mathbf{H}$. By Lemma B.30, $E \vdash P' : \mathbf{H}$. By induction hypothesis, there is a type $W \triangleq G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ such that $E \vdash x : W$ and $G \in \mathbf{H}$. \square

B.3 Correctness of Type Erasure

In this section, we study the relations between the typed and untyped version of the π -calculus defined in this paper. We prove Proposition B.38, which gives a simple correspondence between the reductions of a typed term, say P , and the reductions of the untyped process obtained by erasing all type informations from P . The benefit of this result is that it allows us to use the labelled transition semantics given in Appendix A.3 to reason about typed process. This is particularly useful because, in contrast with a labelled transition, a reduction tells us nothing about the possible interactions of a process with an arbitrary environment. Moreover, it is simpler to enumerate the possible transitions of a process than its possible reductions.

Lemma B.33 *For all typed processes P and Q , if $P \equiv Q$ then $\text{erase}(P) \equiv \text{erase}(Q)$.*

Proof An easy induction on the derivation of $P \equiv Q$. \square

Lemma B.34 *For all typed processes P , $P \downarrow \beta$ if and only if $\text{erase}(P) \downarrow \beta$.*

Proof An easy induction on the derivations of $P \downarrow \beta$ and $\text{erase}(P) \downarrow \beta$. \square

If \vec{G} is a sequence G_1, \dots, G_n of groups, let $(\nu\vec{G})P = (\nu G_1) \cdots (\nu G_n)P$. In particular, if $n = 0$, then $(\nu\vec{G})P = P$.

Lemma B.35 Assume $E \vdash P$.

- (1) If $\text{erase}(P) = Q \mid R$ then there exist two typed processes, Q' , R' , and a sequence of groups, \vec{G} , such that $P = (\nu\vec{G})(Q' \mid R')$, and $\text{erase}(Q') = Q$, and $\text{erase}(R') = R$.
- (2) If $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$ and $\text{erase}(P) = \bar{x}(y_1, \dots, y_n)$ then there exist two sequences of groups, \vec{G} and G'_1, \dots, G'_m , such that $P = (\nu\vec{G})\bar{x}(G'_1, \dots, G'_m, y_1, \dots, y_n)$ and $\{G'_1, \dots, G'_m\} \subseteq \text{dom}(E) \cup \{\vec{G}\}$.
- (3) If $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$ and $\text{erase}(P) = x(y_1, \dots, y_n).R$ then there exist a typed process, Q , and a sequence of groups, \vec{G} , such that $\text{erase}(Q) = R$ and $P = (\nu\vec{G})x(H_1, \dots, H_m, y_1:T_1, \dots, y_n:T_n).Q$.
- (4) If $\text{erase}(P) = (\nu x)R$ then there exist a typed process, Q , a sequence, \vec{G} , and a type, T , such that $P = (\nu\vec{G})(\nu x:T)Q$, and $\text{erase}(Q) = R$.
- (5) If $\text{erase}(P) = !R$ then there exist a typed process, Q , and a sequence, \vec{G} , such that $P = (\nu\vec{G})!Q$, and $\text{erase}(Q) = R$.
- (6) If $\text{erase}(P) = \mathbf{0}$ then there exists a sequence, \vec{G} , such that $P = (\nu\vec{G})\mathbf{0}$.

Proof An easy induction on the structure of P . \square

Lemma B.36 For all typed processes, P , we have $\text{fn}(\text{erase}(P)) = \text{fn}(P)$.

Proof An easy induction on the structure of P . \square

Lemma B.37 If $E \vdash P$ and $\text{erase}(P) \xrightarrow{\tau} R$ then there exists a typed process, Q , such that $P \rightarrow Q$ and $\text{erase}(Q) = R$.

Proof Assume $E \vdash P$. The lemma follows by showing that:

- (1) If $\text{erase}(P) \xrightarrow{x} A$ and $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$, where $A = (y_1, \dots, y_n)R$, then there exist a typed process, Q , and a sequence, \vec{G} , such that $\text{erase}(Q) = R$ and $P \mid \bar{x}(G'_1, \dots, G'_m, y'_1, \dots, y'_n) \rightarrow (\nu\vec{G})Q\sigma\{y_1 \leftarrow y'_1\} \cdots \{y_n \leftarrow y'_n\}$ for any sequences $y'_1, \dots, y'_n, G'_1, \dots, G'_m$, where $E \vdash y'_i : T_i\sigma$ for each $i \in 1..n$ and $\sigma = \{H_1 \leftarrow G'_1\} \cdots \{H_m \leftarrow G'_m\}$.
- (2) If $\text{erase}(P) \xrightarrow{\bar{x}} A$ and $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$, where $A = (\nu\vec{z})(y_1, \dots, y_n)R$, then there exist a typed process, Q , and two sequences of groups, \vec{G} and G'_1, \dots, G'_m , and types, \vec{T} , such that $\text{erase}(Q) = R$ and $P \equiv (\nu\vec{G})(\nu\vec{z}:\vec{T})(\bar{x}(G'_1, \dots, G'_m, y_1, \dots, y_n) \mid Q)$ and $E, \vec{G}, \vec{z}:\vec{T} \vdash y_i : T_i\sigma$ for each $i \in 1..n$, where $\sigma = \{H_1 \leftarrow G'_1\} \cdots \{H_m \leftarrow G'_m\}$.

- (3) If $\text{erase}(P) \xrightarrow{\tau} A$ then there exists a typed process, Q , such that $P \rightarrow Q$ and $\text{erase}(Q) = A$.

We proceed by induction on the derivation of $\text{erase}(P) \xrightarrow{\alpha} A$.

(Trans In) Then $\alpha = x$, $\text{erase}(P) = x(z_1, \dots, z_n).R$ and $A = (z_1, \dots, z_n)R$. Assume $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$. By Lemma B.35, there exist a typed process, Q , and a sequence, \vec{G} , such that $P = (\nu \vec{G})x(H_1, \dots, H_m, z_1:T_1, \dots, z_n:T_n).Q$, and $\text{erase}(Q) = R$. By (Red Interact) and (Red \equiv), $P \mid \bar{x}\langle G'_1, \dots, G'_m, y'_1, \dots, y'_n \rangle \rightarrow (\nu \vec{G})Q\{z_1 \leftarrow y'_1\} \cdots \{z_n \leftarrow y'_n\}$, as required.

(Trans Out) Then $\alpha = \bar{x}$, $\text{erase}(P) = \bar{x}\langle y_1, \dots, y_n \rangle$ and $A = (\nu)\langle y_1, \dots, y_n \rangle \mathbf{0}$, that is $R = \mathbf{0}$. Assume $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$. By Lemma B.35, there exists sequences, \vec{G} and G'_1, \dots, G'_m , such that $P = (\nu \vec{G})\bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle$. Let Q be the typed process $\mathbf{0}$. Hence, $\text{erase}(Q) = \mathbf{0}$ and, by (Struct Par Zero) and (Struct GRes), P is structurally equivalent to $(\nu \vec{G})(\bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \mid \mathbf{0})$, as required.

(Trans Inter 1) Then $\alpha = \tau$ and $\text{erase}(P) = (R_1 \mid R_2)$, where $R_1 \xrightarrow{x} (z_1, \dots, z_n)R'_1$ and $R_2 \xrightarrow{\bar{x}} (\nu \vec{z})\langle y_1, \dots, y_n \rangle R'_2$, and $A = (\nu \vec{z})(R'_1\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\} \mid R'_2)$. By Lemma B.35, there exist two typed processes, P_1 , P_2 , and a sequence, \vec{G} , such that $P = (\nu \vec{G})(P_1 \mid P_2)$ and $\text{erase}(P_i) = R_i$ for each $i \in \{1, 2\}$. Assume $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$.

By induction hypothesis (1), there is a typed process, Q_1 , and a sequence, \vec{G}_1 , such that $P_1 \mid \bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \rightarrow (\nu \vec{G}_1)Q_1\{H_1 \leftarrow G'_1\} \cdots \{H_m \leftarrow G'_m\}\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\}$ and $\text{erase}(Q_1) = R'_1$ for any well-typed sequences y_1, \dots, y_n .

By induction hypothesis (2), there is a typed process, Q_2 , sequences, \vec{G}_2 and G'_1, \dots, G'_m , and types \vec{T} , such that $P_2 \equiv (\nu \vec{G}_2)(\nu \vec{z}:\vec{T})(\bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \mid Q_2)$ and $\text{erase}(Q_2) = R'_2$ and $E, \vec{G}, \vec{z}:\vec{T} \vdash y_i : T_i\sigma$ for each $i \in 1..n$, where $\sigma = \{H_1 \leftarrow G'_1\} \cdots \{H_m \leftarrow G'_m\}$.

Since the names \vec{G}_2 and \vec{z} are bound, we may assume $\{\vec{G}_2\} \cap \text{fg}(P_1) = \emptyset$ and $\{\vec{z}\} \cap \text{fn}(P_1) = \emptyset$. By (Struct GRes Par), (Struct Res Par), (Struct GRes Res), and (Struct Par Assoc), $P_1 \mid P_2 \equiv (\nu \vec{G}_2)(\nu \vec{z}:\vec{T})((P_1 \mid \bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle) \mid Q_2)$. By (Red \equiv), (Red Par), (Red Res) and (Red GRes), $P_1 \mid P_2 \rightarrow (\nu \vec{G}_2)(\nu \vec{z}:\vec{T})((\nu \vec{G}_1)Q_1\sigma\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\} \mid Q_2)$. Let Q be the typed process $(\nu \vec{G})(\nu \vec{G}_2)(\nu \vec{z}:\vec{T})((\nu \vec{G}_1)Q_1\sigma\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\} \mid Q_2)$. By (Red GRes), $P \rightarrow Q$. By definition of the erasure function, $\text{erase}(Q) = (\nu \vec{z})(\text{erase}(Q_1)\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\} \mid \text{erase}(Q_2))$. Hence $\text{erase}(Q) = A$, as required.

Case (Trans Inter 2) is symmetric.

(Trans Par 1) Then $\text{erase}(P) = R_1 \mid R_2$, where $R_1 \xrightarrow{\alpha} A_1$ and $A = A_1 \mid R_2$. By Lemma B.35, there exist two typed processes, P_1 , P_2 , and a sequence,

\vec{G} , such that $P = (\nu\vec{G})(P_1 \mid P_2)$ and $erase(P_i) = R_i$ for each $i \in \{1, 2\}$. We proceed by case analysis on the action α .

(In) We have $\alpha = x$ and $A_1 = (z_1, \dots, z_n)R'_1$. Assume $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$. Since the names z_1, \dots, z_n are bound, we may assume $\{z_1, \dots, z_n\} \cap fv(R_2) = \emptyset$. Hence, $A = (z_1, \dots, z_n)(R'_1 \mid R_2)$, that is, $R = R'_1 \mid R_2$. By induction hypothesis (1), there is a typed process, Q_1 , and a sequence, \vec{G}'_1 , such that $erase(Q_1) = R'_1$ and $P_1 \mid \bar{x}\langle G'_1, \dots, G'_m, y'_1, \dots, y'_n \rangle \rightarrow (\nu\vec{G}'_1)Q_1\sigma\{z_1 \leftarrow y'_1\} \cdots \{z_n \leftarrow y'_n\}$ for any sequences $G'_1, \dots, G'_m, y'_1, \dots, y'_n$, where $E \vdash y'_i : T_i\sigma$ for each $i \in 1..n$ and $\sigma = \{H_1 \leftarrow G'_1\} \cdots \{H_m \leftarrow G'_m\}$. Since the groups \vec{G}'_1 are bound, we may assume $\{\vec{G}'_1\} \cap fv(R_2) = \emptyset$. Let $Q = Q_1 \mid P_2$. By (Struct GRes Par), (Struct Par Assoc) and (Struct Par Comm), $P \mid \bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \equiv (\nu\vec{G})((P_1 \mid \bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle) \mid P_2)$. By (Red \equiv), (Red Par) and (Red GRes), $P \mid \bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \rightarrow (\nu\vec{G}, \vec{G}'_1)Q\sigma\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\}$. Moreover, $erase(Q) = erase(Q_1) \mid erase(P_2) = erase(R)$, as required.

(Out) We have $\alpha = \bar{x}$ and $A_1 = (\nu\vec{z})\langle y_1, \dots, y_n \rangle R'_1$. Assume $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$. Since the names \vec{z} are bound, we may assume $\{\vec{z}\} \cap fv(R_2) = \emptyset$. Hence, $A = (\nu\vec{z})\langle y_1, \dots, y_n \rangle (R'_1 \mid R_2)$, that is, $R = R'_1 \mid R_2$. By induction hypothesis (2), there is a typed process, Q_1 , sequences, \vec{G}'_1 and G'_1, \dots, G'_m , and types \vec{T} such that $erase(Q_1) = R'_1$ and $P_1 \equiv (\nu\vec{G}'_1)(\nu\vec{z}:\vec{T})(\bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \mid Q_1)$ and $E, \vec{G}, \vec{z}:\vec{T} \vdash y_i : T_i\sigma$ for each $i \in 1..n$, where $\sigma = \{H_1 \leftarrow G'_1\} \cdots \{H_m \leftarrow G'_m\}$. Since the groups \vec{G}'_1 are bound, we may assume $\{\vec{G}'_1\} \cap fv(R_2) = \emptyset$. Let $Q = Q_1 \mid P_2$. By (Struct GRes), (Struct GRes Par), (Struct Res Par), (Struct Par Assoc) and (Struct Par Comm), $P \equiv (\nu\vec{G}, \vec{G}'_1)(\nu\vec{z}:\vec{T})(\bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \mid Q)$. Moreover, $erase(Q) = erase(Q_1) \mid erase(P_2) = erase(R)$, as required.

(Tau) We have $\alpha = \tau$. By induction hypothesis (3), there is a typed process, Q_1 , such that $erase(Q_1) = A_1$ and $P_1 \rightarrow Q_1$. Let $Q = Q_1 \mid P_2$. By (Red Par), $P \rightarrow Q$. Moreover, $erase(Q) = erase(Q_1) \mid erase(P_2) = erase(A)$, as required.

Cases (Trans Par 2) and (Trans Res) are similar.

(Trans Repl) Then $erase(P) = !S$ where $S \mid !S \xrightarrow{\alpha} A$. By Lemma B.35, there exist a typed process, P' and a sequence, \vec{G}' , such that $P = (\nu\vec{G}')!P'$ and $erase(P') = S$. Therefore, we have a derivation of $erase((\nu\vec{G}')!P' \mid !P') \xrightarrow{\alpha} A$ and, by (Struct Repl Par) and (Struct GRes), $(\nu\vec{G}')!P' \mid !P' \equiv P$. We proceed by case analysis on the action α .

(In) We have $\alpha = x$ and $A = (z_1, \dots, z_n)R$. Assume $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$. By induction hypothesis (1), there is a typed process, Q , and a sequence, \vec{G}' , such that $erase(Q) = R$ and $(\nu\vec{G}')!P' \mid$

$!P' \mid \bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \rightarrow (\nu \vec{G})Q\sigma\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\}$ for any well-typed sequences $G'_1, \dots, G'_m, y_1, \dots, y_n$ with $\sigma = \{H_1 \leftarrow G'_1\} \cdots \{H_m \leftarrow G'_m\}$. By (Struct Par) and $(\nu \vec{G}') (P' \mid !P') \equiv P$, we get that $P \mid \bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \rightarrow (\nu \vec{G})Q\sigma\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\}$, as required.

(Out) We have $\alpha = \bar{x}$ and $A = (\nu \vec{z})\langle y_1, \dots, y_n \rangle R$. Assume $E \vdash x : G(H_1, \dots, H_m)[T_1, \dots, T_n] \setminus \mathbf{H}$. By induction hypothesis (2), there is a typed process, Q , sequences, \vec{G} and G'_1, \dots, G'_m , and types \vec{T} , such that $\text{erase}(Q) = R$ and $(\nu \vec{G}') (P' \mid !P') \equiv (\nu \vec{G})(\nu \vec{z}:\vec{T})(\bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \mid Q)$ and y_1, \dots, y_n are well-typed. Hence, By (Struct Trans) and $(\nu \vec{G}') (P' \mid !P') \equiv P$, we get that $P \equiv (\nu \vec{G})(\nu \vec{z}:\vec{T})(\bar{x}\langle G'_1, \dots, G'_m, y_1, \dots, y_n \rangle \mid Q)$, as required.

(Tau) We have $\alpha = \tau$. By induction hypothesis (3), there is a typed process, Q , such that $\text{erase}(Q) = R$ and $(\nu \vec{G}') (P' \mid !P') \rightarrow Q$. By (Red \equiv) and $(\nu \vec{G}') (P' \mid !P') \equiv P$, we get that $P \rightarrow Q$. \square

The following asserts that the reductions of a typed process, of our extending π -calculus, according to the typed operational semantics are equivalent to the reductions of the untyped erasure of the process according to the untyped operational semantics. A proof of Proposition 3.1, a similar property for the unextended π -calculus, can be obtained by simplifying the following proof.

Proposition B.38 *For all typed processes P and Q , if $P \rightarrow Q$ then $\text{erase}(P) \rightarrow \text{erase}(Q)$. If $E \vdash P$ and $\text{erase}(P) \rightarrow R$ then there is a typed process Q such that $P \rightarrow Q$ and $R \equiv \text{erase}(Q)$.*

Proof The first implication is proved by a simple induction on the derivation of $P \rightarrow Q$, with appeal to Lemma B.33.

Assume $E \vdash P$ and $\text{erase}(P) \rightarrow R$. By Proposition A.1, there is an untyped process S such that $\text{erase}(P) \xrightarrow{\tau} S$ and $S \equiv R$. By Lemma B.37, there exists a typed process Q such that $P \rightarrow Q$ and $\text{erase}(Q) = S$. By (Struct Trans), $\text{erase}(Q) \equiv R$, as required. \square

Next, we show that if the erasure of two typed processes are equivalent, according to the untyped barbed congruence defined in Appendix A.4, then these processes are barbed congruent.

Proposition B.39 *If $E \vdash P$ and $E \vdash Q$ and $\text{erase}(P) \approx \text{erase}(Q)$ then $E \vdash P \approx Q$.*

Proof Let \mathcal{S} be the relation on typed processes such that $E \vdash P \mathcal{S} Q$ if and only if $E \vdash P$ and $E \vdash Q$ and $\text{erase}(P) \approx \text{erase}(Q)$. We prove that \mathcal{S} is a bisimulation.

- (1) Assume $E \vdash P \mathcal{S} Q$. By definition, we get that $E \vdash P$ and $E \vdash Q$ and $\text{erase}(P) \approx \text{erase}(Q)$. By Proposition A.4 (1), $\text{erase}(Q) \approx \text{erase}(P)$. Hence, $E \vdash Q \mathcal{S} P$.

- (2) Assume $E \vdash P \mathcal{S} Q$ and $P \downarrow \bar{x}$. By Lemma B.34, $erase(P) \downarrow \bar{x}$. Since $erase(P) \approx erase(Q)$, we get that $erase(Q) \downarrow \bar{x}$. By Lemma B.37 several times, there is a typed process R such that $Q \rightarrow^* R$ and $erase(R) \downarrow \bar{x}$. By Lemma B.34, $Q \downarrow \bar{x}$, as required.
- (3) Assume $E \vdash P \mathcal{S} Q$ and $P \rightarrow P'$. By Proposition 3.1, $erase(P) \rightarrow erase(P')$. Since $erase(P) \approx erase(Q)$, we get that $erase(Q) \rightarrow^* R$ for some untyped process R such that $R \approx erase(P')$. By Proposition 3.1 several times, there is a typed process Q' such that $Q \rightarrow^* Q'$ and $erase(Q') \equiv R$. By Propositions A.4 (1) and A.4 (4), we get that $erase(Q') \equiv erase(P')$. By Proposition 3.2, $E \vdash P'$ and $E \vdash Q'$. Hence, $E \vdash P' \mathcal{S} Q'$, as required.

Since \mathcal{S} is a bisimulation, we get that $E \vdash P$ and $E \vdash Q$ and $erase(P) \approx erase(Q)$ implies $E \vdash P \dot{\approx} Q$.

Assume $E \vdash P$ and $E \vdash Q$ and $erase(P) \approx erase(Q)$. Let R be an arbitrary process and σ be an arbitrary E -renaming and E' be an environment such that $E\sigma, E' \vdash R$. In particular, by Lemma B.16, it must be the case that $E\sigma, E' \vdash \diamond$. By Lemmas B.27 and B.28, $E\sigma, E' \vdash P\sigma$. By (Proc Par), $E\sigma, E' \vdash (P\sigma \mid R)$ and $E\sigma, E' \vdash (Q\sigma \mid R)$. Moreover, $erase(P\sigma \mid R) = erase(P)\sigma \mid erase(R)$ and $erase(Q\sigma \mid R) = erase(Q)\sigma \mid erase(R)$. Since the untyped barbed congruence, \approx , is closed by substitution and parallel composition (see Propositions A.4 (2) and A.4 (3)), we get that $erase(P\sigma \mid R) \approx erase(Q\sigma \mid R)$. Hence, $E\sigma, E' \vdash (P\sigma \mid R) \dot{\approx} (Q\sigma \mid R)$, as required. \square

B.4 Properties of Barbed Congruence

In this section we study some properties of typed barbed congruence. We prove Lemma B.40, that structurally equivalent processes of the extended π -calculus are barbed congruent. This fact corresponds to Proposition 3.4 (4), for the unextended π -calculus. We also prove Proposition B.46, that barbed congruence is indeed a compositional equivalence relation. The proof of this property relies on Lemma B.45, that barbed congruence is preserved by arbitrary substitutions of groups for groups. It also relies on the fact that, by definition, barbed congruence is preserved by E -renamings, that is substitution of names for names that cannot identify names with different types.

Lemma B.40 *If $P \equiv Q$ and $E \vdash P$ then $E \vdash P \approx Q$*

Proof Let \mathcal{S} be the smallest relation on typed processes that contains \approx and such that $E \vdash P \mathcal{S} Q$ if $P \equiv Q$ and $E \vdash P$. The relation \mathcal{S} is a well-defined relation on typed processes since \approx is a relation on typed processes and if $P \equiv Q$ and $E \vdash P$ then, by Proposition B.31, $E \vdash Q$. Note that \mathcal{S} is symmetric. We prove that \mathcal{S} is a barbed bisimulation. The only interesting case is when $E \vdash P$ and $P \equiv Q$.

- (1) Assume $P \downarrow \beta$. By rule (Barb \equiv), $Q \downarrow \beta$, as required.

- (2) Assume $P \rightarrow P'$. By (Red \equiv), $Q \rightarrow P'$ and, since \approx is reflexive, $P' \mathcal{S} P'$, as required.

Therefore \mathcal{S} is a barbed bisimulation and for all processes P, Q such that $E \vdash P$ and $P \equiv Q$, we get that $E \vdash P \dot{\approx} Q$.

Assume $E \vdash P$ and $P \equiv Q$. Let R be an arbitrary typed process, σ be an arbitrary E -renaming and E' be an environment such that $E, E' \vdash R$. By Lemma B.28 several times and rule (Struct Par), $E\sigma, E' \vdash P\sigma \mid R$ and $P\sigma \mid R \equiv Q\sigma \mid R$. Therefore, $E\sigma, E' \vdash P\sigma \mid R \dot{\approx} Q\sigma \mid R$. Hence, $E \vdash P \approx Q$, as required. \square

We show that barbed congruence for the extended π -calculus is closed by E -renamings, that is substitutions of names for names that respect types. This fact corresponds to Proposition 3.4 (3), for the unextended π -calculus.

Lemma B.41 *If $E \vdash P \approx Q$ and σ is an E -renaming then $E\sigma \vdash P\sigma \approx Q\sigma$.*

Proof Assume $E \vdash P \approx Q$ and σ is an E -renaming. By definition, $E\sigma, E' \vdash P\sigma \mid R \approx Q\sigma \mid R$ for any process R such that $E\sigma, E' \vdash R$. Since $E\sigma \vdash \mathbf{0}$, we get that $E\sigma \vdash P\sigma \mid \mathbf{0} \approx Q\sigma \mid \mathbf{0}$. By Lemma B.40 and transitivity of \approx , since $P\sigma \mid \mathbf{0} \equiv P\sigma$ and $Q\sigma \mid \mathbf{0} \equiv Q$, we get that $E\sigma \vdash P\sigma \approx Q\sigma$. \square

Next, we prove Lemmas B.42 and B.43, that structural equivalence and reduction are preserved by arbitrary substitution of groups for groups. These properties are used in the proof of Lemma B.44, that barbed congruence is preserved by substitutions of groups for groups.

Lemma B.42 *If $P \equiv P'$ then $P\{G \leftarrow G'\} \equiv P'\{G \leftarrow G'\}$.*

Proof An easy induction on the derivation of $P \equiv P'$. \square

Lemma B.43 *If $P \rightarrow P'$ then $P\{G \leftarrow G'\} \rightarrow P'\{G \leftarrow G'\}$.*

Proof An easy induction on the derivation of $P \rightarrow P'$, with appeal to Lemma B.42. \square

The following shows that barbed congruence is preserved by substitution of groups. This fact agrees with our previous observation that types do not interfere with the operational behaviour of processes, see Proposition B.38.

Lemma B.44 *If $E, G, E' \vdash P \approx Q$ and $G' \in \text{dom}(E)$ then $E, E'\{G \leftarrow G'\} \vdash P\{G \leftarrow G'\} \approx Q\{G \leftarrow G'\}$.*

Proof Let \mathcal{S} be the smallest relation on typed processes that contains \approx and such that $E, E'\{G \leftarrow G'\} \vdash P\{G \leftarrow G'\} \mathcal{S} Q\{G \leftarrow G'\}$ if $E, G, E' \vdash P \approx Q$ and $G' \in \text{dom}(E)$. Assume $E, G, E' \vdash P$ and $G' \in \text{dom}(E)$. By Lemma B.29, $E, E'\{G \leftarrow G'\} \vdash P\{G \leftarrow G'\}$. Therefore, if $E, G, E' \vdash P \approx Q$ then the processes $P\{G \leftarrow G'\}$ and $Q\{G \leftarrow G'\}$ are well-typed in the environment $E, E'\{G \leftarrow G'\}$. Hence, \mathcal{S} is a relation on typed processes. Next, we prove that \mathcal{S} is a barbed bisimulation. The only interesting case is when $E, E'\{G \leftarrow G'\} \vdash P\{G \leftarrow G'\} \mathcal{S} Q\{G \leftarrow G'\}$, where $E, G, E' \vdash P \approx Q$. Note that \mathcal{S} is symmetric.

- (1) Assume $P\{G \leftarrow G'\} \downarrow \bar{x}$. By Lemma B.34, $\text{erase}(P\{G \leftarrow G'\}) \downarrow \bar{x}$, that is, $\text{erase}(P) \downarrow \bar{x}$. By Lemma B.34 again, $P \downarrow \bar{x}$. Since P and Q are barbed congruent, $Q \downarrow \bar{x}$. By Lemma B.34 several times, since $\text{erase}(Q) = \text{erase}(Q)\{G \leftarrow G'\}$, we get that $Q\{G \leftarrow G'\} \downarrow \bar{x}$.
- (2) Assume $P\{G \leftarrow G'\} \rightarrow P'$. By Lemma B.43, $P \rightarrow P'\{G' \leftarrow G\}$. Since P and Q are barbed congruent, $Q \rightarrow^* Q'$ with $E, G, E' \vdash Q' \approx P'\{G' \leftarrow G\}$. By Lemma B.43 several times, $Q\{G \leftarrow G'\} \rightarrow^* Q'\{G \leftarrow G'\}$. By definition, $E, E'\{G \leftarrow G'\} \vdash Q'\{G \leftarrow G'\} \mathcal{S} P'$, as required.

Therefore \mathcal{S} is a barbed bisimulation and if $E, G, E' \vdash P \approx Q$ and $G' \in \text{dom}(E)$ then $E, E'\{G \leftarrow G'\} \vdash P\{G \leftarrow G'\} \approx Q\{G \leftarrow G'\}$. Let R be an arbitrary typed process, σ be an arbitrary $(E, E'\{G \leftarrow G'\})$ -renaming and E'' be an environment such that $E, E'\{G \leftarrow G'\}, E'' \vdash R$. Assume $E, G, E' \vdash P \approx Q$. By Lemma B.29 and (Proc Par), $E, G, E', E''\{G' \leftarrow G\} \vdash P\sigma \mid R\{G' \leftarrow G\} \approx Q\sigma \mid R\{G' \leftarrow G\}$. Then $E, E'\{G \leftarrow G'\}, E'' \vdash P\{G \leftarrow G'\}\sigma \mid R \approx Q\{G \leftarrow G'\}\sigma \mid R$. Hence, $E, E'\{G \leftarrow G'\} \vdash P\{G \leftarrow G'\} \approx Q\{G \leftarrow G'\}$. \square

We introduce some new notations to simplify the presentation of the following properties. If E is a type environment $G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n$, let $x(E).P$ be the process $x(G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n).P$. In particular, $x(\emptyset).P = x().P$. If σ is a substitution of groups for groups, the environment $E\sigma$ is defined as follows: $\emptyset\sigma \triangleq \emptyset$; $(E', x:T)\sigma \triangleq E'\sigma, x:T\sigma$; $(E', G)\sigma \triangleq E'\sigma, \sigma(G)$ if $\sigma(G) \notin \text{dom}(E'\sigma)$, and $E'\sigma$ otherwise.

A corollary of Lemma B.44 is the following property.

Lemma B.45 *If $E \vdash P \approx Q$ and σ is a substitution of groups for groups then $E\sigma \vdash P\sigma \approx Q\sigma$.*

Next, we prove that barbed congruence for the extended π -calculus satisfies the congruence property. This fact corresponds to Proposition 3.4 (2), for the unextended π -calculus.

Proposition B.46

- (1) *Let E' be the environment $G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n$. If $E, E' \vdash P \approx Q$ then $E \vdash x(E').P \approx x(E').Q$.*
- (2) *If $E \vdash P \approx Q$ and $E \vdash R$ then $E \vdash P \mid R \approx Q \mid R$.*
- (3) *If $E, x:T \vdash P \approx Q$ then $E \vdash (\nu x:T)P \approx (\nu x:T)Q$.*
- (4) *If $E, G \vdash P \approx Q$ then $E \vdash (\nu G)P \approx (\nu G)Q$.*
- (5) *If $E \vdash P \approx Q$ then $E \vdash !P \approx !Q$.*

Proof For the sake of brevity, we only prove the case for input prefix, which is the most difficult case. The proofs for the other cases are similar. As in the proof of Lemma B.40, the property follows by defining a candidate barbed

bisimulation, \mathcal{S} , that is closed by renamings and parallel composition. Let \mathcal{S} be the smallest relation on typed processes that contains \approx and such that $E \vdash x(E').P \mid R \mathcal{S} x(E').Q \mid R$ for all processes P, Q, R such that $E, E' \vdash P \approx Q$ and $E \vdash R$. Note that \mathcal{S} is a symmetric relation on typed processes. We prove that \mathcal{S} is a barbed bisimulation. The only interesting case is when $E \vdash x(E').P \mid R \mathcal{S} x(E').Q \mid R$, where $E, E' \vdash P \approx Q$.

- (1) Assume $x(E').P \mid R \downarrow \bar{x}$. By Lemma B.34, Proposition A.3 and inspection of the possible transitions, it must be the case that $R \downarrow \bar{x}$. By (Barb Par) and (Barb \equiv), since $R \downarrow \bar{x}$, we get that $x(E').Q \mid R \downarrow \bar{x}$, as required.
- (2) Assume $x(E').P \mid R \rightarrow P'$. Suppose E' is the type environment $G_1, \dots, G_m, y_1:T_1, \dots, y_n:T_n$. By Propositions B.38 and A.1 and inspection of the possible transitions, either (1) $R \rightarrow R'$ and $P' \equiv x(E').P \mid R'$, or (2) $R \equiv (\nu E'')(\bar{x}\langle H_1, \dots, H_m, z_1, \dots, z_n \rangle \mid R')$ and $P' \equiv (\nu E'')(P\sigma_G\sigma_y \mid R')$ with $\sigma_G = \{G_1 \leftarrow H_1\} \cdots \{G_m \leftarrow H_m\}$ and $\sigma_y = \{y_1 \leftarrow z_1\} \cdots \{y_n \leftarrow z_n\}$.

For (1), by (Red Par), $x(E').Q \mid R \rightarrow x(E').Q \mid R'$. By Proposition B.31, $E \vdash R'$. Hence, $E \vdash x(E').P \mid R' \mathcal{S} x(E').Q \mid R'$, as required.

For (2), let Q' be the process $(\nu E'')(Q\sigma_G\sigma_y \mid R')$. By (Red Interact), (Red Par) and (Red \equiv), $x(E').Q \mid R \rightarrow Q'$. By Lemma B.45, since $E, E' \vdash P \approx Q$, we get that $(E, E')\sigma_G \vdash P\sigma_G \approx Q\sigma_G$. Since the names in E' and E'' are bound, we can assume that $\text{dom}(E') \cap \text{dom}(E'') = \emptyset$. By Lemma B.27 several times and since $E, E'' \vdash \bar{x}\langle H_1, \dots, H_m, z_1, \dots, z_n \rangle \mid R'$, we get that $E''' \vdash P\sigma_G \approx Q\sigma_G$, where E''' is the type environment $E, E'', y_1:T_1\sigma_G, \dots, y_n:T_n\sigma_G$. Since $E \vdash x(E').P \mid R$, $E \vdash x(E').Q \mid R$ and $E, E'' \vdash \bar{x}\langle H_1, \dots, H_m, z_1, \dots, z_n \rangle \mid R'$, the substitution σ_y is an E''' -renaming. By Lemma B.41, since $(E, E')\sigma_G \vdash P\sigma_G \approx Q\sigma_G$ and $z_i \in \text{dom}(E, E'')$ for each $i \in 1..n$, we get that $E, E'' \vdash P\sigma_G\sigma_y \approx Q\sigma_G\sigma_y$. Therefore, using laws (2), (3) and (4), we get that $E \vdash (\nu E'')(P\sigma_G\sigma_y \mid R') \approx (\nu E'')(Q\sigma_G\sigma_y \mid R')$. Hence, since the relation \approx (and then also \equiv) is in \mathcal{S} , we get that $E \vdash P' \mathcal{S} Q'$, as required.

Therefore \mathcal{S} is a barbed bisimulation and if $E, E' \vdash P \approx Q$ then $E \vdash x(E').P \mid R \overset{\circ}{\approx} x(E').Q \mid R$ for any process R such that $E \vdash R$. Assume $E, E' \vdash P \approx Q$. Let R be an arbitrary typed process, σ be an arbitrary E -renaming and E'' be an environment such that $E, E'' \vdash R$. Since the names in E' are bound we can assume that $\text{dom}(E') \cap \text{dom}(E'') = \emptyset$. Therefore, by Lemmas B.27 and B.28, $E\sigma, E'', E' \vdash P\sigma \approx Q\sigma$, and then $E\sigma, E'' \vdash x(E').P\sigma \mid R \overset{\circ}{\approx} x(E').Q\sigma \mid R$. Hence, $E \vdash x(E').P \approx x(E').Q$, as desired. \square

B.5 Garbage Collection for the π -Calculus

In this section we prove Theorem B.53, the garbage collection principle used to prove the soundness of the region analysis. This property follows from several intermediate lemmas that prove that processes with non-intersecting effects cannot interact. For example, Lemma B.50 shows that these processes cannot

synchronize. In the sense that their parallel composition do not introduce new silent transitions.

Next, we give three properties of barbs that will prove useful in the proof of Lemma B.52.

Lemma B.47 *For any process P , if $(\nu G)P \downarrow \beta$ then $P \downarrow \beta$. If $(\nu y:T)P \downarrow \beta$ then $P \downarrow \beta$ with $\beta \in \{x, \bar{x}\}$ and $x \neq y$.*

Lemma B.48 *For any processes P, Q , if $(P \mid Q) \downarrow \beta$ then $P \downarrow \beta$ or $Q \downarrow \beta$.*

Lemma B.49 *For any processes R , if $E, G, E' \vdash R : \{G\}$ and $R \downarrow \beta$ and $\beta \in \{x, \bar{x}\}$ then $x \in \text{dom}(E')$.*

Proof Assume $E, G, E' \vdash R : \{G\}$ and $R \downarrow \beta$ and $\beta \in \{x, \bar{x}\}$. By Proposition B.32, there is a type $T = G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ such that $E \vdash x : T$. Since $E, G, E' \vdash R : \{G\}$, we have $E, G, E' \vdash \diamond$. Therefore, we get that $G \notin \text{dom}(E)$. Hence, $x \in \text{dom}(E')$. \square

Lemma B.50 *For any processes P and R such that $E, G, E' \vdash P : \mathbf{H}$ and $E, G, E' \vdash R : \{G\}$ and $G \notin \mathbf{H}$, if $\text{erase}(P \mid R) \xrightarrow{\alpha} A$ then there is an agent A' such that $\text{erase}(P) \xrightarrow{\alpha} A'$ and $A = A' \mid \text{erase}(R)$, or such that $\text{erase}(R) \xrightarrow{\alpha} A'$ and $A = \text{erase}(P) \mid A'$.*

Proof Assume $E, G, E' \vdash P : \mathbf{H}$ and $E, G, E' \vdash R : \{G\}$ and $G \notin \mathbf{H}$ and $\text{erase}(P) \mid \text{erase}(R) \xrightarrow{\alpha} A$. We proceed by case analysis on the last rule used to derived this reduction. We prove that the only possible rules are (Trans Par 1) and (Trans Par 2).

(Trans Par 1) Then $\text{erase}(P) \xrightarrow{\alpha} A'$ with $A = A' \mid \text{erase}(R)$.

(Trans Par 2) Then $\text{erase}(R) \xrightarrow{\alpha} A'$ with $A = \text{erase}(P) \mid A'$.

(Trans Inter 1) Then $\alpha = \tau$ and $\text{erase}(P) \xrightarrow{x} F$ and $\text{erase}(R) \xrightarrow{\bar{x}} C$ and $A = F @ C$. By Proposition A.3 and Lemma B.34, and since $\text{erase}(R) \xrightarrow{\bar{x}} C$, it must be the case that $R \downarrow \bar{x}$. Therefore, by Proposition B.32, there is a type $T = G(G_1, \dots, G_m)[T_1, \dots, T_n] \setminus \mathbf{G}$ such that $E, G, E' \vdash x : T$. Symmetrically, using Proposition A.3, Lemma B.34, and the transition $\text{erase}(P) \xrightarrow{x} F$, we prove that $P \downarrow x$. Therefore, by Proposition B.32, there is a type $T' = G'(G'_1, \dots, G'_{m'})[T'_1, \dots, T'_{n'}] \setminus \mathbf{G}'$ such that $G' \in \mathbf{H}$. By Lemma B.20, it must be the case that G equals G' , which contradicts $G \notin \mathbf{H}$. Case (Trans Inter 2) is symmetric. \square

Lemma B.51 *For any processes P and R such that $E, G, E' \vdash P : \mathbf{H}$ and $E, G, E' \vdash R : \{G\}$ and $G \notin \mathbf{H}$, if $P \mid R \rightarrow Q$ then there is a process P' such that $P \rightarrow P'$ and $Q \equiv P' \mid R$, or there is a process R' such that $R \rightarrow R'$ and $Q \equiv P \mid R'$.*

Proof Assume $P \mid R \rightarrow Q$. By Proposition A.1, there is a process S such that $\text{erase}(P) \mid \text{erase}(R) \xrightarrow{\tau} S$ and $\text{erase}(Q) \equiv S$. By Lemma B.50, there is an agent A' such that $\text{erase}(P) \xrightarrow{\tau} A'$ and $S = A' \mid \text{erase}(R)$, or such that $\text{erase}(R) \xrightarrow{\tau} A'$ and $S = \text{erase}(P) \mid A'$. By Propositions A.1 and 3.1 we get that, in the first case, there exists P' such that $P \rightarrow P'$ and $Q \equiv P' \mid R$ and, in the latter case, there is R' such that $R \rightarrow R'$ and $Q \equiv P \mid R'$. \square

Lemma B.52 *For any processes P, R such that $E, G, E' \vdash R : \{G\}$ and $E, G, E' \vdash P : \mathbf{H}$ and $G \notin \mathbf{H}$, we have: $E \vdash (\nu G)(\nu E')(P \mid R) \dot{\approx} (\nu G)(\nu E')P$.*

Proof To show that $(\nu G)(\nu E')(P \mid R)$ and $(\nu G)(\nu E')P$ are barbed bisimilar, and hence prove the lemma, we define a barbed bisimulation, \mathcal{S} , such that $E \vdash (\nu G)(\nu E')(P \mid R) \mathcal{S} (\nu G)(\nu E')P$. Let \mathcal{S} be the smallest symmetric relation on typed processes such that $E \vdash (\nu G)(\nu E')(P \mid R) \mathcal{S} (\nu G)(\nu E')P$ for all processes P, R such that $E, G, E' \vdash P : \mathbf{H}$ and $E, G, E' \vdash R : \{G\}$ and $G \notin \mathbf{H}$. We prove that \mathcal{S} is a barbed bisimulation.

Consider any processes P and R such that $E, G, E' \vdash P : \mathbf{H}$ and $E, G, E' \vdash R : \{G\}$ and $G \notin \mathbf{H}$.

- (1) By (Proc Par), (Proc GRes) and (Proc Res), $E \vdash (\nu G)(\nu E')(P \mid R) : (\mathbf{H} \cup \{G\}) - \text{dom}(G, E')$ and $E \vdash (\nu G)(\nu E')P : \mathbf{H} - \text{dom}(G, E')$. Hence, $E \vdash (\nu G)(\nu E')(P \mid R)$ and $E \vdash (\nu G)(\nu E')P$.
- (2) Suppose $(\nu G)(\nu E')(P \mid R) \downarrow \bar{x}$. By Lemma B.47, $(P \mid R) \downarrow \bar{x}$ with $x \notin \text{dom}(E')$. By Lemma B.48, $P \downarrow \bar{x}$ or $R \downarrow \bar{x}$. In the second case, by Lemma B.49, we get that $x \in \text{dom}(E')$, which contradicts the fact that $x \notin \text{dom}(E')$. Hence, $P \downarrow \bar{x}$ and, by (Barb Res) several times, $(\nu G)(\nu E')P \downarrow \bar{x}$, as required.
- (3) Suppose $(\nu G)(\nu E')P \downarrow \bar{x}$. By Lemma B.47, $P \downarrow \bar{x}$ with $x \notin \text{dom}(E')$. By (Barb Par) and (Barb Res), $(\nu G)(\nu E')(P \mid R) \downarrow \bar{x}$, as required.
- (4) Suppose $(\nu G)(\nu E')(P \mid R) \rightarrow Q$. By Lemma A.2, there is a process Q' such that $(P \mid R) \rightarrow Q'$ and $Q \equiv (\nu G)(\nu E')Q'$. By Lemma B.51, there is a process P' such that $P \rightarrow P'$ and $Q' \equiv P' \mid R$, or there is a typed process R' such that $R \rightarrow R'$ and $Q' \equiv P \mid R'$. We consider first the case where the reduction comes from P .
 - Assume $P \rightarrow P'$ and $Q' \equiv P' \mid R$. By (Red Res) and (Red GRes), $(\nu G)(\nu E')P \rightarrow (\nu G)(\nu E')P'$ and $Q \equiv (\nu G)(\nu E')(P' \mid R)$. By Proposition 3.2, $E, G, E' \vdash P' : \mathbf{H}$. Hence, $E \vdash Q \equiv \mathcal{S} \equiv (\nu G)(\nu E')P'$, as required.

We consider now the case where the reduction comes from R .

- Assume $R \rightarrow R'$ and $Q' \equiv P \mid R'$. By (Red Res) and (Red GRes), $Q \equiv (\nu G)(\nu E')Q' \equiv (\nu G)(\nu E')(P \mid R')$. By Proposition 3.2, we get that $E, G, E' \vdash R' : \{G\}$. Therefore $E \vdash Q \equiv \mathcal{S} \equiv (\nu G)(\nu E')P$, as required.

- (5) Suppose $(\nu G)(\nu E')P \rightarrow Q$. By Lemma A.2, there is a process P' such that $P \rightarrow P'$ and $Q \equiv (\nu G)(\nu E')P'$. By (Red Par), (Red Res) and (Red GRes), $(\nu G)(\nu E')(P \mid R) \rightarrow (\nu G)(\nu E')(P' \mid R)$. By Proposition 3.2, $E, G, E' \vdash P' : \mathbf{H}$. Hence, $E \vdash Q \equiv S \equiv (\nu G)(\nu E')(P' \mid R)$, as required.

Hence, S is a barbed bisimulation and $E \vdash (\nu G)(\nu E')(P \mid R) \dot{\approx} (\nu G)(\nu E')P$. \square

The following is the garbage collection principle for our extended π -calculus. A proof of Theorem 4.3, garbage collection for the unextended π -calculus, can be obtained by simplifying the following proof.

Theorem B.53 *Suppose $E, G, E' \vdash P : \mathbf{H}$ and $E, G, E' \vdash R : \{G\}$ where $G \notin \mathbf{H}$. Then $E \vdash (\nu G)(\nu E')(P \mid R) \approx (\nu G)(\nu E')P$.*

Proof To show that $(\nu G)(\nu E')(P \mid R)$ and $(\nu G)(\nu E')P$ are barbed congruent, and hence prove the theorem, we consider an arbitrary process Q , type environment E'' and E -renaming σ such that $E\sigma, E'' \vdash Q$, and show that $E\sigma, E'' \vdash (\nu G)(\nu E')(P \mid R)\sigma \mid Q \dot{\approx} ((\nu G)(\nu E')P)\sigma \mid Q$.

Assume $E\sigma, E'' \vdash Q : \mathbf{G}$. Since the names in $\text{dom}(G, E')$ are bound, we may assume that $\text{dom}(G, E') \cap (\text{dom}(E'') \cup \text{dom}(\sigma) \cup \text{ran}(\sigma)) = \emptyset$. Hence, since $\text{fv}(Q) \subseteq \text{dom}(E\sigma, E'')$ and $\mathbf{G} \subseteq \text{dom}(E\sigma, E'')$, we get that $\text{fv}(Q) \cap \text{dom}(G, E') = \mathbf{G} \cap \text{dom}(G, E') = \emptyset$ and $G \notin \mathbf{G}$. By Lemma B.28 several times, and since $E, G, E' \vdash P : \mathbf{H}$, we get that $E\sigma, E'', G, E' \vdash P\sigma : \mathbf{H}$. By Lemma B.27, $E\sigma, E'', G, E' \vdash Q : \mathbf{G}$. By (Proc Par), $E\sigma, E'', G, E' \vdash (P\sigma \mid Q) : \mathbf{G} \cup \mathbf{H}$ with $G \notin \mathbf{G} \cup \mathbf{H}$. Since $\text{dom}(G, E') \cap (\text{dom}(E'') \cup \text{dom}(\sigma) \cup \text{ran}(\sigma)) = \emptyset$, we get that:

$$\begin{aligned} (\nu G)(\nu E')(P \mid R)\sigma \mid Q &\equiv (\nu G)(\nu E')(P\sigma \mid Q \mid R\sigma) \\ ((\nu G)(\nu E')P)\sigma \mid Q &\equiv (\nu G)(\nu E')(P\sigma \mid Q) \end{aligned}$$

By Lemma B.52, $E\sigma, E' \vdash (\nu G)(\nu E')(P \mid R)\sigma \mid Q \dot{\approx} ((\nu G)(\nu E')P)\sigma \mid Q$, as required. \square

B.6 Properties of the Encoding

In this section we prove the soundness of the region analysis for the extended region calculus. For the sake of clarity, this proof is divided into simpler goals as follows.

In Section B.6.1, we prove that our encoding of the extended region calculus in the extended π -calculus preserves the static semantics given in Sections 6.1.

In Section B.6.3, we prove Theorem B.67, a similar result for the dynamic semantics. Results of dynamic adequacy are often difficult to prove directly when the source calculus is defined with a big-step semantics. To circumvent this difficulty, we follow a standard method and define an equivalent small-step semantics for the region calculus. This semantics is given in Section B.6.2 where we also prove Theorem B.62, which relates small-steps and big-steps reductions.

In Section B.6.4, we prove that defunct regions make no difference to the behaviour of a well-typed program. This result is essentially based on the garbage

collection theorem proved in Section B.5, which is used to prove that the encoding of a well-typed configuration is behaviourally equivalent to the process obtained by erasing from the memory heap all the references stored in defunct regions.

B.6.1 Proof of Static Adequacy

We prove a series of properties, Lemma B.54 to B.58, that correspond to the properties listed in Theorem 4.1 for the unextended calculi.

Lemma B.54

- (1) If $E \vdash \diamond$ then $\llbracket E \rrbracket \vdash \diamond$.
- (2) If $E \vdash A$ then $\llbracket E \rrbracket \vdash \llbracket A \rrbracket$.

Proof Part (1) follows easily by induction on the derivation of $E \vdash \diamond$. We prove part (2) by induction on the derivation of $E \vdash A$.

(Type Lit) Then $A = \text{Lit}$, and $\llbracket A \rrbracket = \text{Lit} \llbracket \cdot \rrbracket$. By definition, $\text{Lit} \in \text{dom}(\llbracket \cdot \rrbracket)$. By part (1), $\llbracket E \rrbracket \vdash \diamond$. By (Type Chan), $\llbracket E \rrbracket \vdash \text{Lit} \llbracket \cdot \rrbracket$.

(Type \rightarrow) Then $A = (\forall[\rho_1, \dots, \rho_n] B_1 \xrightarrow{e} B_2)$ at ρ , where $E' = E, \rho_1, \dots, \rho_n$, and $E' \vdash B_1$, and $e \subseteq \text{dom}(E')$, and $E' \vdash B_2$, and $\rho \in \text{dom}(E)$.

By induction hypothesis, $\llbracket E \rrbracket, \rho_1, \dots, \rho_n \vdash \llbracket B_i \rrbracket$ for each $i \in \{1, 2\}$. By (Type Chan), since $\rho \in \text{dom}(\llbracket E \rrbracket)$, we get that $\llbracket E \rrbracket \vdash \rho(\rho_1, \dots, \rho_n) \llbracket \llbracket B_1 \rrbracket, K \llbracket \llbracket B_2 \rrbracket \rrbracket \setminus (e \cup \{K\})$, as required.

(Type List) Then $A = [B]$ at ρ , where $E \vdash B$ and $\rho \in \text{dom}(E)$. By induction hypothesis, $\llbracket E \rrbracket \vdash \llbracket B \rrbracket$. By definition, $\llbracket A \rrbracket$ is the recursive type $\mu(X)[\rho \llbracket \cdot \rrbracket, \rho \llbracket \llbracket B \rrbracket, X \rrbracket]$. Since the name X is bound, we may assume that $X \notin \text{dom}(E)$. By Lemma B.27, $\llbracket E \rrbracket, X \vdash \llbracket B \rrbracket$. By (Type Chan), since $\rho \in \text{dom}(\llbracket E \rrbracket)$, $\llbracket E \rrbracket, X \vdash \rho \llbracket \llbracket B \rrbracket, X \rrbracket$. By (Type Chan) and (Type Rec), $\llbracket E \rrbracket \vdash \llbracket A \rrbracket$, as required. \square

Lemma B.55

- (1) If $E \vdash x :^{\varnothing} A$ then $\llbracket E \rrbracket \vdash x : \llbracket A \rrbracket$.
- (2) If $E \vdash a :^e A$ and $k \notin L \cup \text{dom}(E)$ then $\llbracket E \rrbracket, k : K \llbracket \llbracket A \rrbracket \rrbracket \vdash \llbracket a \rrbracket k : e \cup \{K\}$.

Proof Part (1) follows easily by induction on the structure of E . We prove part (2) by induction on the derivation of $E \vdash a :^e A$. Recall that $G[T_1, \dots, T_n]$ is a shorthand for the type $G() [T_1, \dots, T_n] \setminus \emptyset$.

(Exp x) Then $a = x$ and $E = E_1, x : A, E_2$ and $e = \emptyset$. Assume $k \notin L \cup \text{dom}(E)$. By part (1), $\llbracket E \rrbracket \vdash x : \llbracket A \rrbracket$. By (Proc Output) and Lemma B.27, $\llbracket E \rrbracket, k : K \llbracket \llbracket A \rrbracket \rrbracket \vdash \bar{k} \langle x \rangle : \{K\}$, as required.

(Exp l) Then $a = l$, where $l \in L$ and $A = Lit$. By definition, $\llbracket E \rrbracket \vdash l : Lit$. Assume $k \notin L \cup dom(E)$. By (Proc Output) and Lemma B.27, $\llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash \bar{k}(l) : \{K\}$, as required.

(Exp Appl) Then $a = x[\rho'_1, \dots, \rho'_n](y)$ and $e = \{\rho\} \cup e'\sigma$, with $E \vdash x : {}^\circ F$ and $E \vdash y : {}^\circ B_1\sigma$ and $F = (\forall[\rho_1, \dots, \rho_n]B_1 \xrightarrow{e'} B_2)$ at ρ and $\sigma = \{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$ and $A = B_2\sigma$ and $\{\rho'_1, \dots, \rho'_n\} \subseteq dom(E)$. By part (1), $\llbracket E \rrbracket \vdash x : \llbracket F \rrbracket$ and $\llbracket E \rrbracket \vdash y : \llbracket B_1\sigma \rrbracket$. Assume $k \notin L \cup dom(E)$. By (Exp Unfold) and Lemma B.27:

$$\llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash x : \rho(\rho_1, \dots, \rho_n) \llbracket B_1 \rrbracket, K[\llbracket B_2 \rrbracket] \setminus (e' \cup \{K\})$$

By (Proc Output), $\llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash \bar{x}(\rho'_1, \dots, \rho'_n, y, k) : e'\sigma \cup \{K\}$, as required.

(Exp Let) Then $a = (let\ x = b_B\ in\ c)$ and $e = e' \cup e''$, with $E \vdash b : {}^{e'} B$ and $E, x : B \vdash c : {}^{e''} A$. Assume $k \notin L \cup dom(E)$. By induction hypothesis:

$$\left\{ \begin{array}{l} \llbracket E \rrbracket, k' : K[\llbracket B \rrbracket] \vdash \llbracket b \rrbracket k' : e' \cup \{K\} \\ \llbracket E \rrbracket, x : \llbracket B \rrbracket, k : K[\llbracket A \rrbracket] \vdash \llbracket c \rrbracket k : e'' \cup \{K\} \end{array} \right.$$

By (Proc Input) and Lemmas B.25 and B.21:

$$\llbracket E \rrbracket, k : K[\llbracket A \rrbracket], k' : K[\llbracket B \rrbracket] \vdash k'(x : \llbracket B \rrbracket). \llbracket c \rrbracket k : \{K\} \cup (e'' \cup \{K\})$$

By (Proc Par) and (Proc Res):

$$\llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash (\nu k' : K[\llbracket B \rrbracket]) (\llbracket b \rrbracket k' \mid k'(x : \llbracket B \rrbracket). \llbracket c \rrbracket k) : (e' \cup \{K\}) \cup (e'' \cup \{K\})$$

Hence, $\llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k : e \cup \{K\}$, as required.

(Exp Letregion) Then $a = letregion\ \rho\ in\ b$ and $e = e' - \{\rho\}$, with $E, \rho \vdash b : {}^{e'} A$ and $E \vdash A$. Assume $k \notin L \cup dom(E)$. By induction hypothesis, $\llbracket E \rrbracket, \rho, k : K[\llbracket A \rrbracket] \vdash \llbracket b \rrbracket k : e' \cup \{K\}$. By Lemma B.54 (2), $\llbracket E \rrbracket \vdash \llbracket A \rrbracket$. Therefore, since $\rho \notin dom(E)$, we have that $\rho \notin fg(K[\llbracket A \rrbracket])$ and, by Lemma B.24, $\llbracket E \rrbracket, k : K[\llbracket A \rrbracket], \rho \vdash \llbracket b \rrbracket k : e' \cup \{K\}$. By (Proc GRes), $\llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash (\nu \rho) \llbracket b \rrbracket k : (e' \cup \{K\}) - \{\rho\}$, as required.

(Exp Case) Then $a = case\ x_{[B]at\rho}\ of\ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2$ and $e = \{\rho\} \cup e_1 \cup e_2$, with $E \vdash x : {}^\circ [B]$ at ρ and $E \vdash b_1 : {}^{e_1} A$ and $E, y_1 : B, y_2 : [B]$ at $\rho \vdash b_2 : {}^{e_2} A$. Assume $k \notin L \cup dom(E)$. By part (1) and induction hypothesis:

$$\left\{ \begin{array}{l} \llbracket E \rrbracket \vdash x : \llbracket [B] at\ \rho \rrbracket \\ \llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash \llbracket b_1 \rrbracket k : e_1 \cup \{K\} \\ \llbracket E \rrbracket, y_1 : [B], y_2 : \llbracket [B] at\ \rho \rrbracket, k : K[\llbracket A \rrbracket] \vdash \llbracket b_2 \rrbracket k : e_2 \cup \{K\} \end{array} \right.$$

By (Exp Unfold), $\llbracket E \rrbracket \vdash x : \rho[\rho[], \rho[\llbracket B \rrbracket], \llbracket [B] \text{ at } \rho \rrbracket]$. By Lemma B.27 and (Proc Output):

$$\left\{ \begin{array}{l} \llbracket E \rrbracket, k:K[\llbracket [A] \rrbracket], z_1:\rho[], z_2:\rho[\llbracket [B] \rrbracket], \llbracket [B] \text{ at } \rho \rrbracket] \vdash \bar{x}\langle z_1, z_2 \rangle : \{\rho\} \\ \llbracket E \rrbracket, k:K[\llbracket [A] \rrbracket], z_1:\rho[], z_2:\rho[\llbracket [B] \rrbracket], \llbracket [B] \text{ at } \rho \rrbracket] \vdash z_1().\llbracket [b_1] \rrbracket k : e_1 \cup \{K\} \\ \llbracket E \rrbracket, k:K[\llbracket [A] \rrbracket], z_1:\rho[], z_2:\rho[\llbracket [B] \rrbracket], \llbracket [B] \text{ at } \rho \rrbracket] \vdash \\ z_2(y_1:\llbracket [B] \rrbracket, y_2:\llbracket [B] \text{ at } \rho \rrbracket).\llbracket [b_2] \rrbracket k : e_2 \cup \{K\} \end{array} \right.$$

By (Proc Par) and (Proc Res), $\llbracket E \rrbracket, k:K[\llbracket [A] \rrbracket] \vdash \llbracket [a] \rrbracket k : \{\rho\} \cup (e_1 \cup \{K\}) \cup (e_2 \cup \{K\})$, as required.

(Exp Fun) Then $a = v \text{ at } \rho$ and $e = \{\rho\}$ and $A = (\forall[\rho_1, \dots, \rho_n]B_1 \xrightarrow{e} B_2) \text{ at } \rho$, where v is the function $(\mu(f:A)\lambda[\rho_1, \dots, \rho_n](x)b)$ and $E, f:A, \rho_1, \dots, \rho_n, x:B_1 \vdash b : e'$ B_2 and $e' \subseteq e \subseteq \text{dom}(E, \rho_1, \dots, \rho_n)$. Assume $k \notin L \cup \text{dom}(E) \cup \{p\}$. Since f and p are bound names, we can also assume that $k \notin \{f, p\}$. By induction hypothesis and Lemma B.27:

$$\llbracket E \rrbracket, p:[A], f:[A], \rho_1, \dots, \rho_n, x:[B_1], k:K[\llbracket [B_2] \rrbracket] \vdash \llbracket [b] \rrbracket k : e' \cup \{K\}$$

By Lemma B.28:

$$\llbracket E \rrbracket, p:[A], \rho_1, \dots, \rho_n, x:[B_1], k:K[\llbracket [B_2] \rrbracket] \vdash \llbracket [b\{f \leftarrow p\}] \rrbracket k : e' \cup \{K\}$$

By (Exp x) and (Exp Unfold):

$$\llbracket E \rrbracket, p:[A] \vdash p : \rho(\rho_1, \dots, \rho_n)[\llbracket [B_1] \rrbracket, K[\llbracket [B_2] \rrbracket]] \setminus (e \cup \{K\})$$

By (Proc Input), $\llbracket E \rrbracket, p:[A] \vdash \llbracket [p \mapsto v] \rrbracket : \{\rho\}$. By (Proc Output) and Lemma B.27, $\llbracket E \rrbracket, k:K[\llbracket [A] \rrbracket], p:[A] \vdash \bar{k}\langle p \rangle : \{K\}$. By (Proc Par) and (Proc Res), $\llbracket E \rrbracket, k:K[\llbracket [A] \rrbracket] \vdash (\nu p:[A]) (\llbracket [p \mapsto v] \rrbracket \mid \bar{k}\langle p \rangle) : \{K, \rho\}$, as required.

(Exp Nil) Then $a = \text{nil} \text{ at } \rho$ and $e = \{\rho\}$ and $A = [B] \text{ at } \rho$, where $E \vdash [B] \text{ at } \rho$. By Lemma B.54 (2), $\llbracket E \rrbracket \vdash \llbracket [B] \text{ at } \rho \rrbracket$. By Lemma B.27:

$$\llbracket E \rrbracket, p:[A], z_1:\rho[], z_2:\rho[\llbracket [B] \rrbracket], \llbracket [B] \text{ at } \rho \rrbracket] \vdash \bar{z}_1\langle \rangle : \{\rho\}$$

By (Exp x) and (Exp Unfold):

$$\llbracket E \rrbracket, p:[A], z_1:\rho[], z_2:\rho[\llbracket [B] \rrbracket], \llbracket [B] \text{ at } \rho \rrbracket] \vdash p : \rho[\rho[], \rho[\llbracket [B] \rrbracket], \llbracket [B] \text{ at } \rho \rrbracket]]$$

By (Proc Input), $\llbracket E \rrbracket, p:[A] \vdash \llbracket [p \mapsto \text{nil}_{[B] \text{ at } \rho}] \rrbracket : \{\rho\}$. Assume $k \notin L \cup \text{dom}(E) \cup \{p\}$. By (Proc Output) and Lemma B.27, $\llbracket E \rrbracket, k:K[\llbracket [A] \rrbracket], p:[A] \vdash \bar{k}\langle p \rangle : \{K\}$. By (Proc Par) and (Proc Res), $\llbracket E \rrbracket, k:K[\llbracket [A] \rrbracket] \vdash (\nu p:[A]) (\llbracket [p \mapsto \text{nil}_A] \rrbracket \mid \bar{k}\langle p \rangle) : \{K, \rho\}$, as required.

(Exp Cons) Then $a = (x_1 :: x_2)$ at ρ and $e = \{\rho\}$ and $A = [B]$ at ρ , where $E \vdash x_1 :^{\varnothing} B$ and $E \vdash x_2 :^{\varnothing} [B]$ at ρ . By part (1), $\llbracket E \rrbracket \vdash x_1 : \llbracket B \rrbracket$ and $\llbracket E \rrbracket \vdash x_2 : \llbracket [B] \text{ at } \rho \rrbracket$. By (Exp x), (Exp Unfold) and Lemma B.27:

$$\llbracket E \rrbracket, p : \llbracket A \rrbracket, z_1 : \rho[], z_2 : \rho[\llbracket B \rrbracket, \llbracket [B] \text{ at } \rho \rrbracket] \vdash p : \rho[\rho[], \rho[\llbracket B \rrbracket, \llbracket [B] \text{ at } \rho \rrbracket]]$$

By (Proc Output):

$$\llbracket E \rrbracket, p : \llbracket A \rrbracket, z_1 : \rho[], z_2 : \rho[\llbracket B \rrbracket, \llbracket [B] \text{ at } \rho \rrbracket] \vdash \overline{z_2}(x_1, x_2) : \{\rho\}$$

By (Proc Input), $\llbracket E \rrbracket, p : \llbracket A \rrbracket \vdash \llbracket p \mapsto (x_1 :: x_2)_A \rrbracket : \{\rho\}$. Assume $k \notin L \cup \text{dom}(E) \cup \{\rho\}$. By (Proc Output) and Lemma B.27, $\llbracket E \rrbracket, k : K[\llbracket A \rrbracket], p : \llbracket A \rrbracket \vdash \overline{k}(p) : \{K\}$. By (Proc Par) and (Proc Res), $\llbracket E \rrbracket, k : K[\llbracket A \rrbracket] \vdash (\nu p : \llbracket A \rrbracket)(\llbracket p \mapsto (x_1 :: x_2)_A \rrbracket \mid \overline{k}(p)) : \{K, \rho\}$, as required. \square

Lemma B.56 *If $H \models h$ and $h(\rho)(p) = v$ then $\llbracket \text{env}(H) \rrbracket \vdash \llbracket p \mapsto v \rrbracket : \{\rho\}$.*

Proof Assume $H \models h$ and $h(\rho)(p) = v$. By (Heap Good) and (Region Good), there is a type, A , such that $H(\rho)(p) = A$. By (Exp x), $\text{env}(H) \vdash p :^{\varnothing} A$. By Lemma B.55 (1), $\llbracket \text{env}(H) \rrbracket \vdash p : \llbracket A \rrbracket$.

By (Heap Good), since $\rho \in \text{dom}(h)$, we get that $\text{env}(H) \vdash h(\rho)$ at $\rho : H(\rho)$. This must have been derived from (Region Good) with $\text{env}(H) \vdash v$ at $\rho : \{\rho\} A$. Only (Exp Fun), (Exp Nil) and (Exp Cons) can derive this judgment and so we have three possible cases.

(Exp Fun) Then v is the function $(\mu(f:A)\lambda[\rho_1, \dots, \rho_n](x)b)$ and A is the type $(\forall[\rho_1, \dots, \rho_n]B_1 \xrightarrow{e} B_2)$ at ρ , with $\text{env}(H), f:A, \rho_1, \dots, \rho_n, x:B_1 \vdash b :^{e'} B_2$ and $e' \subseteq e \subseteq \text{dom}(E, \rho_1, \dots, \rho_n)$. Assume $k \notin L \cup \text{dom}(E) \cup \{f, p, x\}$. By Lemma B.55 (2):

$$\llbracket \text{env}(H) \rrbracket, f : \llbracket A \rrbracket, \rho_1, \dots, \rho_n, x : \llbracket B_1 \rrbracket, k : K[\llbracket B_2 \rrbracket] \vdash \llbracket b \rrbracket k : e' \cup \{K\}$$

By Lemma B.28, since $\llbracket \text{env}(H) \rrbracket \vdash p : \llbracket A \rrbracket$:

$$\llbracket \text{env}(H) \rrbracket, \rho_1, \dots, \rho_n, x : \llbracket B_1 \rrbracket, k : K[\llbracket B_2 \rrbracket] \vdash \llbracket b\{f \leftarrow p\} \rrbracket k : e' \cup \{K\}$$

By (Exp Unfold), since $\llbracket \text{env}(H) \rrbracket \vdash p : \llbracket A \rrbracket$:

$$\llbracket \text{env}(H) \rrbracket \vdash p : \rho(\rho_1, \dots, \rho_n)[\llbracket B_1 \rrbracket, K[\llbracket B_2 \rrbracket]] \setminus (e \cup \{K\})$$

By (Proc Input), $\llbracket \text{env}(H) \rrbracket \vdash \llbracket p \mapsto v \rrbracket : \{\rho\}$, as required.

(Exp Nil) Then $v = \text{nil}$ and $A = [B]$ at ρ , where $\text{env}(H) \vdash [B]$ at ρ and $e = \{\rho\}$. Assume z_1, z_2 are fresh names. By (Proc Output) and Lemma B.27:

$$\llbracket \text{env}(H) \rrbracket, z_1 : \rho[], z_2 : \rho[\llbracket B \rrbracket, \llbracket [B] \text{ at } \rho \rrbracket] \vdash \overline{z_1}() : \{\rho\}$$

By (Exp Unfold), since $\llbracket \text{env}(H) \rrbracket \vdash p : \llbracket A \rrbracket$:

$$\llbracket \text{env}(H) \rrbracket \vdash p : \rho[\rho[], \rho[\llbracket B \rrbracket, \llbracket [B] \text{ at } \rho \rrbracket]]$$

By (Proc Input), $\llbracket \text{env}(H) \rrbracket \vdash \llbracket p \mapsto \text{nil}_{[B] \text{ at } \rho} \rrbracket : \{\rho\}$, as required.

(Exp Cons) Then $v = (x_1 :: x_2)$ and $A = [B]$ at ρ , where $env(H) \vdash x_1 :^{\varnothing} B$ and $env(H) \vdash x_2 :^{\varnothing} [B]$ at ρ . By Lemma B.55 (1), $\llbracket env(H) \rrbracket \vdash x_1 : \llbracket B \rrbracket$ and $\llbracket env(H) \rrbracket \vdash x_2 : \llbracket [B] \text{ at } \rho \rrbracket$. Assume z_1, z_2 are fresh names. By (Proc Output) and Lemma B.27:

$$\llbracket env(H) \rrbracket, z_1 : \rho[], z_2 : \rho[\llbracket B \rrbracket, \llbracket [B] \text{ at } \rho \rrbracket] \vdash \overline{z_2}(x_1, x_2) : \{\rho\}$$

By (Exp Unfold), since $\llbracket env(H) \rrbracket \vdash p : [A]$:

$$\llbracket env(H) \rrbracket \vdash p : \rho[\rho[], \rho[\llbracket B \rrbracket, \llbracket [B] \text{ at } \rho \rrbracket]]$$

By (Proc Input), $\llbracket env(H) \rrbracket \vdash [p \mapsto (x_1 :: x_2)_A] : \{\rho\}$, as required. \square

Lemma B.57 *If $H \models h$ and $\rho \in dom(H)$ then $\llbracket env(H) \rrbracket \vdash [h(\rho)] : \{\rho\}$.*

Proof Assume $H \models h$ and $\rho \in dom(H)$. The judgment $H \models h$ must have been derived from (Heap Good) with $env(H) \vdash h(\rho)$ at $\rho : H(\rho)$. This must have been derived from (Region Good) with $h(\rho) = (p_i \mapsto v_i)^{i \in 1..n}$ and $H(\rho) = (p_i : A_i)^{i \in 1..n}$ and $env(H) \vdash v_i$ at $\rho : \{\rho\} A_i$ for all $i \in 1..n$. By (Exp x), since $H(\rho) = (p_i : A_i)^{i \in 1..n}$, we get that $env(H) \vdash p_i :^{\varnothing} A_i$ for each $i \in 1..n$. By Lemma B.55 (1), $\llbracket env(H) \rrbracket \vdash p_i : [A_i]$. By Lemma B.56, $\llbracket env(H) \rrbracket \vdash [p_i \mapsto v_i] : \{\rho\}$ for each $i \in 1..n$. By (Proc Par), $\llbracket env(H) \rrbracket \vdash \prod_{i \in 1..n} [p_i \mapsto v_i] : \{\rho\}$. Hence, $\llbracket env(H) \rrbracket \vdash [h(\rho)] : \{\rho\}$. \square

Lemma B.58 *If $H \models S \cdot (a, h) : A$ and $k \notin dom_2(H) \cup L$ then:*

$$\llbracket env(H) \rrbracket, k : K[\llbracket A \rrbracket] \vdash [a]k \mid [h] : dom(H) \cup \{K\}$$

and also $\llbracket \varnothing \rrbracket, S, k : K[\llbracket A \rrbracket] \vdash [S \cdot (a, h_H)]k : S \cup \{K\}$.

Proof Assume $H \models S \cdot (a, h) : A$ and $k \notin dom_2(H) \cup L$. Only (Config Good) can derive this judgment and so $env(H) \vdash a :^e A$, $e \cup fg(A) \subseteq S$, $H \models h$, and $S \subseteq dom(H)$. By Lemma B.55 (2), $\llbracket env(H) \rrbracket, k : K[\llbracket A \rrbracket] \vdash [a]k : e \cup \{K\}$. By Lemma B.57 and (Proc Par), $\llbracket env(H) \rrbracket \vdash [h] : \bigcup_{\rho \in dom(H)} \{\rho\}$. By (Proc Par) and Lemma B.27, $\llbracket env(H) \rrbracket, k : K[\llbracket A \rrbracket] \vdash [a]k \mid [h] : dom(H) \cup e \cup \{K\}$. Since $e \subseteq dom(H)$, we get that $\llbracket env(H) \rrbracket, k : K[\llbracket A \rrbracket] \vdash [a]k \mid [h] : dom(H) \cup \{K\}$, as desired.

By (Proc Res) and (Proc GRes), since $env(H) = dom(H), ptr(H)$, we get that $\llbracket \varnothing \rrbracket, S, k : K[\llbracket A \rrbracket] \vdash (\nu(dom(H) - S))(\nu[ptr(H)])([a]k \mid [h]) : S \cup \{K\}$, that is, $\llbracket \varnothing \rrbracket, S, k : K[\llbracket A \rrbracket] \vdash [S \cdot (a, h_H)]k : S \cup \{K\}$. \square

The following asserts that the encoding of the extended region calculus in our extended π -calculus preserves the static semantics. This fact corresponds to Theorem 4.1 for the unextended calculi.

Theorem B.59 (Static Adequacy)

(1) *If $E \vdash \diamond$ then $\llbracket E \rrbracket \vdash \diamond$.*

(2) If $E \vdash A$ then $\llbracket E \rrbracket \vdash \llbracket A \rrbracket$.

(3) If $E \vdash a :^e A$ and $k \notin \text{dom}(\llbracket E \rrbracket)$ then

$$\llbracket E \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k : e \cup \{K\}$$

(4) If $H \models h$ and $\rho \in \text{dom}(H)$ then

$$\llbracket \text{env}(H) \rrbracket \vdash \llbracket h(\rho) \rrbracket : \{\rho\}$$

(5) If $H \models S \cdot (a, h) : A$ and $k \notin \llbracket \text{env}(H) \rrbracket$ then

$$\llbracket \text{env}(H) \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k \mid \llbracket h \rrbracket : \text{dom}(H) \cup \{K\}$$

and also

$$\llbracket \emptyset \rrbracket, S, k:K[\llbracket A \rrbracket] \vdash \llbracket S \cdot (a, h) \rrbracket k : S \cup \{K\}$$

Proof Combine Lemmas B.54, B.55, B.57 and B.58. \square

B.6.2 An Auxiliary Small Step Semantics

This section defines an auxiliary small step semantics for the region calculus. We prove Theorem B.62, that relates small step reductions to evaluations in the big step semantics.

Continuations and Control stack:

$c ::=$	continuations
$\text{popregion } \rho$	marker to deallocate region ρ
$(x:A)b$	continuation with argument x
$C ::= [c_1, \dots, c_n]$	stack of continuations

The reduction relation, $S \cdot (a, h, C) \rightarrow S' \cdot (a', h', C')$, may be read: in an initial heap h , with control stack C and live regions S , the expression a reduces to a' with updated heap h' , control stack C' , and live regions S' .

Reduction: $S \cdot (a, h, C) \rightarrow S' \cdot (a', h', C')$

(Red Alloc)

$$\frac{\rho \in S \quad p \notin \text{dom}_2(h)}{S \cdot (v \text{ at } \rho, h, C) \rightarrow S \cdot (p, h + (\rho \mapsto (h(\rho) + (p \mapsto v))), C)}$$

(Red Appl)

$$\frac{\rho \in S \quad h(\rho)(p) = \mu(f:A)\lambda[\rho_1, \dots, \rho_n](x)b}{S \cdot (p[\rho'_1, \dots, \rho'_n](q), h, C) \rightarrow S \cdot (b\{f \leftarrow p\}\{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}\{x \leftarrow q\}, h, C)}$$

(Red Let)

$$\frac{}{S \cdot (\text{let } x = a_A \text{ in } b, h, C) \rightarrow S \cdot (a, h, (x:A)b :: C)}$$

(Red Pop Let)

$$\frac{}{S \cdot (p, h, (x:A)b :: C) \rightarrow S \cdot (b\{x \leftarrow p\}, h, C)}$$

(Red Letregion)

$$\frac{\rho \notin (S \cup \text{dom}(h))}{S \cdot (\text{letregion } \rho \text{ in } a, h, C) \rightarrow (S \cup \{\rho\}) \cdot (a, h + \rho \mapsto \emptyset, C)}$$

(Red Pop Letregion)

$$\frac{\rho \in S}{S \cdot (p, h, \text{popregion } \rho :: C) \rightarrow (S - \{\rho\}) \cdot (p, h, C)}$$

(Red Case 1)

$$\frac{\rho \in S \quad h(\rho)(p) = \text{nil}}{S \cdot (\text{case } p \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) \rightarrow S \cdot (b_1, h, C)}$$

(Red Case 2)

$$\frac{\rho \in S \quad h(\rho)(p) = q_1 :: q_2}{S \cdot (\text{case } p \text{ of } \text{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) \rightarrow S \cdot (b_2\{y_1 \leftarrow q_1\}\{y_2 \leftarrow q_2\}, h, C)}$$

The static semantics defines new heap judgments used to type the elements in the control stack.

Heap Judgments:

$\vec{B} ::= [A_1, \dots, A_n]$	stack of types
$H \models S \cdot C : \vec{B}$	the control stack C has type \vec{B}
$H \models S \cdot (a, h, C) : A$	in H , the configuration (a, h, C) returns A

Region and Heap Rules:

(Control Good Empty)

$$\frac{\text{env}(H) \vdash A \quad \text{fg}(A) \subseteq S}{H \models S \cdot [] : [A]}$$

(Control Good Mark)

$$\frac{H \models S \cdot C : \vec{B} \quad \rho \notin S}{H \models (S \cup \{\rho\}) \cdot (\text{popregion } \rho :: C) : \vec{B}}$$

(Control Good Cont)

$$\frac{\text{env}(H), x:A \vdash b :^e B \quad \text{fg}(A) \cup e \subseteq S \quad H \models S \cdot C : (B :: \vec{B})}{H \models S \cdot ((x:A)b :: C) : (A :: B :: \vec{B})}$$

(Small Config Good)

$$\frac{H \models S \cdot (a, h) : A \quad H \models S \cdot C : (A :: \vec{B})}{H \models S \cdot (a, h, C) : \text{last}(A :: \vec{B})}$$

Lemma B.60 *If $H \models S \cdot (a, h, C) : A$ and $S \cdot (a, h) \Downarrow (p', h')$ then $S \cdot (a, h, C) \rightarrow^* S \cdot (p', h', C)$.*

Proof An easy induction on the derivation of $S \cdot (a, h) \Downarrow (p', h')$. \square

Lemma B.61 *If $H \models S \cdot (a, h, \square) : A$ and $S \cdot (a, h, \square) \rightarrow^* S_0 \cdot (p_0, h_0, \square)$ then $S \cdot (a, h) \Downarrow (p_0, h_0)$.*

Proof Assume $H \models S \cdot (a, h, C) : A$ and $S \cdot (a, h, C) \rightarrow^* S_0 \cdot (p_0, h_0, \square)$. The lemma follows by showing that:

- (1) If $C = \square$ then $S \cdot (a, h) \Downarrow (p_0, h_0)$.
- (2) If $C = (\text{popregion } \rho :: C')$ then $\rho \in S$ and there is p', h' such that $S \cdot (a, h) \Downarrow (p', h')$ and $S \cdot (p', h', C') \rightarrow^* S_0 \cdot (p_0, h_0, \square)$.
- (3) If $C = ((x:B)b :: C')$ then there is ρ, p', h' such that $S \cdot (a, h) \Downarrow (p', h')$ and $\rho \notin (S \cup \text{dom}(h'))$ and $S = S' \cup \{\rho\}$ and $S' \cdot (b\{x \leftarrow p'\}, h', C') \rightarrow^* S_0 \cdot (p_0, h_0, \square)$.

We proceed by induction on the derivation of $S \cdot (a, h, C) \rightarrow^* S_0 \cdot (p_0, h_0, \square)$.

The base case is for an empty reduction sequence, that is, $a = p_0$ and $h = h_0$ and $C = \square$. By (Eval Var), $S \cdot (a, h) \Downarrow (p_0, h_0)$, as required.

In the general case, there is S', a', h', C' such that $S \cdot (a, h, C) \rightarrow S' \cdot (a', h', C')$ and $S' \cdot (a', h', C') \rightarrow^* S_0 \cdot (p_0, h_0, \square)$. We proceed by case analysis on the derivation of $S \cdot (a, h, C) \rightarrow S' \cdot (a', h', C')$.

(Red Alloc) Then $a = (v \text{ at } \rho)$, and $a' = p$, and $h' = h + (\rho \mapsto (h(\rho) + (p \mapsto v)))$, and $S' = S$, and $C' = C$, where $\rho \in S$ and $p \notin \text{dom}_2(h)$. For part (1), by induction hypothesis (1), we get $S \cdot (p, h) \Downarrow (p_0, h_0)$. This must have been derived from (Eval Var) with $p = p_0$ and $h' = h_0$. By (Eval Alloc), $S \cdot (a, h) \Downarrow (p_0, h_0)$. Parts (2) and (3) are similar.

(Red Appl) Then $a = p[\rho'_1, \dots, \rho'_n](q)$, and $a' = b\{f \leftarrow p\}\{x \leftarrow q\}\sigma$, and $h' = h$, and $S' = S$, and $C' = C$, where $\rho \in S$, $h(\rho)(p)$ is the function $\mu(f:A)\lambda[\rho_1, \dots, \rho_n](x)b$ and σ is the substitution $\{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$. For part (1), by induction hypothesis (1), we get that $S' \cdot (a', h') \Downarrow (p_0, h_0)$. By (Eval Appl), $S \cdot (a, h) \Downarrow (p_0, h_0)$. Parts (2) and (3) are similar.

(Red Let) Then $a = (\text{let } x = a'_A \text{ in } b)$, and $C' = (x:A')b :: C$, and $h' = h$, and $S' = S$. By induction hypothesis (3), there is p'', h'' such that $S' \cdot (a', h') \Downarrow (p'', h'')$ and $S' \cdot (b\{x \leftarrow p''\}, h'') \Downarrow (p_0, h_0)$. For part (1), by (Eval Let), we get that $S \cdot (a, h) \Downarrow (p_0, h_0)$. Parts (2) and (3) are similar.

(Red Pop Let) Then $a = p$, and $a' = b\{x \leftarrow p\}$, and $C = (x:A')b :: C'$, and $h' = h$, and $S' = S$, and $S \cdot (b\{x \leftarrow p\}, h, C') \rightarrow^* S_0 \cdot (p_0, h_0, \square)$. We only have to consider part (3). By (Eval Var), $S \cdot (a, h) \Downarrow (p, h)$, as required.

(Red Letregion) Then $a = (\text{letregion } \rho \text{ in } a')$, and $h' = h + \rho \mapsto \emptyset$, and $S' = S \cup \{\rho\}$, where $\rho \notin (S \cup \text{dom}(h))$, and $C = C'$. For part (1), by induction hypothesis (1), we get that $(S \cup \{\rho\}) \cdot (a', h') \Downarrow (p_0, h_0)$. By (Eval Letregion), $S \cdot (a, h) \Downarrow (p_0, h_0)$. Parts (2) and (3) are similar.

(Red Pop Letregion) Then $a = a' = p$, and $h = h'$, and there is $\rho \in S$ such that $S' = S \setminus \{\rho\}$, and $C = (\text{popregion } \rho :: C')$, and $S' \cdot (p, h, C') \rightarrow^* S_0 \cdot (p_0, h_0, \square)$. We only have to consider part (2). By (Eval Var), $S \cdot (a, h) \Downarrow (p, h)$, as required.

(Red Case 1) Then $a = \text{case } p \text{ of nil} \Rightarrow a' \mid (y_1 :: y_2) \Rightarrow b_2$, and $h = h'$, and $C = C'$, where $\rho \in S$ and $h(\rho)(p) = \text{nil}$. For part (1), by induction hypothesis (1), we get that $S \cdot (a', h) \Downarrow (p_0, h_0)$. By (Eval Case 1), $S \cdot (a, h) \Downarrow (p_0, h_0)$. Parts (2) and (3) are similar.

(Red Case 2) Then $a = \text{case } p \text{ of nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2$, and $a' = b_2\{y_1 \leftarrow q_1\}\{y_2 \leftarrow q_2\}$, and $h = h'$, and $C = C'$, where $\rho \in S$ and $h(\rho)(p) = q_1 :: q_2$. For part (1), by induction hypothesis (1), we get that $S \cdot (a', h) \Downarrow (p_0, h_0)$. By (Eval Case 2), $S \cdot (a, h) \Downarrow (p_0, h_0)$. Parts (2) and (3) are similar. \square

Theorem B.62 *Suppose $H \models S \cdot (a, h) : A$. Then $S \cdot (a, h) \Downarrow (p', h')$ if and only if $S \cdot (a, h, \square) \rightarrow^* S \cdot (p', h', \square)$.*

Proof Assume $H \models S \cdot (a, h) : A$. By (Small Config Good) and (Control Good Empty), $H \models S \cdot (a, h, \square) : A$. If $S \cdot (a, h) \Downarrow (p', h')$ then, by Lemma B.60, $S \cdot (a, h, \square) \rightarrow^* S \cdot (p', h', \square)$. If $S \cdot (a, h, \square) \rightarrow^* S \cdot (p', h', \square)$ then, by Lemma B.61, $S \cdot (a, h) \Downarrow (p', h')$. \square

B.6.3 Proof of Dynamic Adequacy

- The length of a control stack, $\text{length}(C)$, is the number of continuations contained in C , that is, $\text{length}(\square) = 0$, and $\text{length}(\text{popregion } \rho :: C) = \text{length}(C)$, and $\text{length}((x:A)b :: C) = \text{length}(C) + 1$.
- The types of a control stack, $\text{types}(C)$, is the sequence of types inductively defined from C by the following rules, $\text{types}(\square)$ is the empty sequence, and $\text{types}(\text{popregion } \rho :: C) = \text{types}(C)$, and $\text{types}((x:A)b :: C) = A, \text{types}(C)$.

Translation rules:

Let \vec{k} be a stack, $[k_1, \dots, k_n]$, of n pairwise distinct names.

$$\llbracket (x:A)b :: C \rrbracket \vec{k} \triangleq k_1(x:K[\llbracket A \rrbracket]).\llbracket b \rrbracket k_2 \mid \llbracket C \rrbracket [k_2, \dots, k_n]$$

$$\llbracket \text{popregion } \rho :: C \rrbracket \vec{k} \triangleq \llbracket C \rrbracket \vec{k}$$

$$\llbracket \square \rrbracket \vec{k} \triangleq \mathbf{0}$$

Let $\{\vec{\rho}\} = \text{dom}(H) - S$, and $n = \text{length}(C)$, and $[A_1, \dots, A_n] = \text{types}(C)$, and k_1, \dots, k_{n+1} be a sequence of $n + 1$ pairwise distinct names.

$$\llbracket S \cdot (a, h_H, C) \rrbracket k_{n+1} \triangleq (\nu \vec{\rho})(\nu \llbracket \text{ptr}(H) \rrbracket)(\nu k_1 : K \llbracket [A_1] \rrbracket) \cdots (\nu k_n : K \llbracket [A_n] \rrbracket)$$

$$(\llbracket a \rrbracket k_1 \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [k_1, \dots, k_{n+1}])$$

In the case of an empty control stack, $C = []$, the translation of a small-step configuration, $S \cdot (a, h_H, C)$, equals the translation of the big-step configuration $S \cdot (a, h_H)$. That is, we have the following property:

Lemma B.63 $\llbracket S \cdot (a, h_H, []) \rrbracket k = \llbracket S \cdot (a, h_H) \rrbracket k$.

Lemma B.64 *If $H \models S \cdot C : [A_1, \dots, A_{n+1}]$ then $\llbracket \text{env}(H) \rrbracket, k_1 : K \llbracket [A_1] \rrbracket, \dots, k_{n+1} : K \llbracket [A_{n+1}] \rrbracket \vdash \llbracket C \rrbracket [k_1, \dots, k_{n+1}] : \text{dom}(H) \cup \{K\}$.*

Proof By induction on the structure of C . Let \vec{B} be the stack of types $[A_1, \dots, A_{n+1}]$.

(Empty) Then $C = []$ and $n = 0$. The judgment $H \models S \cdot C : \vec{B}$ must have been derived by (Control Good Empty) with $\vec{B} = [A_1]$ and $\text{env}(H) \vdash A_1$. By Lemma B.54, $\llbracket \text{env}(H) \rrbracket \vdash \llbracket [A_1] \rrbracket$. By (Proc Zero) and Lemma B.27, $\llbracket \text{env}(H) \rrbracket, k_1 : K \llbracket [A_1] \rrbracket \vdash \mathbf{0} : \emptyset$. By (Proc Subsum), $\llbracket \text{env}(H) \rrbracket, k_1 : K \llbracket [A_1] \rrbracket \vdash \llbracket [] \rrbracket [k_1] : \text{dom}(H) \cup \{K\}$.

(Cont) Then $C = (x:A_1)b :: C'$. The judgment $H \models S \cdot C : \vec{B}$ must have been derived by (Control Good Cont) with $\text{env}(H), x:A_1 \vdash b :^e A_2$ and $H \models S \cdot C' : (A_2, \dots, A_{n+1})$. By Lemma B.55, $\llbracket \text{env}(H) \rrbracket, x : [A_1], k_2 : K \llbracket [A_2] \rrbracket \vdash \llbracket b \rrbracket k_2 : e \cup \{K\}$. Therefore, by (Proc Input) and Lemma B.27, we get that $\llbracket \text{env}(H) \rrbracket, k_1 : K \llbracket [A_1] \rrbracket, k_2 : K \llbracket [A_2] \rrbracket \vdash k_1(x : [A_1]). \llbracket b \rrbracket k_2 : e \cup \{K\}$. By induction hypothesis, $\llbracket \text{env}(H) \rrbracket, k_2 : K \llbracket [A_2] \rrbracket, \dots, k_{n+1} : K \llbracket [A_{n+1}] \rrbracket \vdash \llbracket C' \rrbracket [k_2, \dots, k_{n+1}] : \text{dom}(H) \cup \{K\}$. By (Proc Par) and Lemma B.27, we finally obtain that $\llbracket \text{env}(H) \rrbracket, k_1 : K \llbracket [A_1] \rrbracket, \dots, k_{n+1} : K \llbracket [A_{n+1}] \rrbracket \vdash \llbracket C \rrbracket [k_1, \dots, k_{n+1}] : \text{dom}(H) \cup \{K\}$, as required.

(Pop) Then $C = \text{popregion } \rho :: C'$ and $\llbracket C \rrbracket \vec{k} = \llbracket C' \rrbracket \vec{k}$. The judgment $H \models S \cdot C : \vec{B}$ must have been derived by (Control Good Mark) with $H \models S \cdot C' : \vec{B}$. By induction hypothesis, $\llbracket \text{env}(H) \rrbracket, k_1 : K \llbracket [A_1] \rrbracket, \dots, k_{n+1} : K \llbracket [A_{n+1}] \rrbracket \vdash \llbracket C' \rrbracket [k_2, \dots, k_{n+1}] : \text{dom}(H) \cup \{K\}$, as required. \square

Proposition B.65 *If $H \models S \cdot (a, h, C) : A$ then:*

$$\llbracket \emptyset \rrbracket, S, k : K \llbracket [A] \rrbracket \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k : S \cup \{K\}$$

Proof Assume $H \models S \cdot (a, h, C) : A$ and $k \notin \text{dom}_2(H) \cup L$. Only (Small Config Good) can derive this judgment and so $H \models S \cdot (a, h) : A_1$ and $H \models S \cdot C : [A_1, \dots, A_{n+1}]$ with $n = \text{length}(C)$ and $A = A_{n+1}$. Let \vec{k} be the stack $[k_1, \dots, k_{n+1}]$ and \vec{T} be the sequence $K \llbracket [A_1] \rrbracket, \dots, K \llbracket [A_{n+1}] \rrbracket$.

By Lemma B.58, $\llbracket \text{env}(H) \rrbracket, k_1:T_1 \vdash \llbracket a \rrbracket k_1 \mid \llbracket h \rrbracket : \text{dom}(H) \cup \{K\}$. By Lemma B.64, $\llbracket \text{env}(H) \rrbracket, \vec{k}:\vec{T} \vdash \llbracket C \rrbracket \vec{k} : \text{dom}(H) \cup \{K\}$. By (Proc Par) and Lemma B.27, $\llbracket \text{env}(H) \rrbracket, \vec{k}:\vec{T} \vdash \llbracket a \rrbracket k_1 \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket \vec{k} : \text{dom}(H) \cup \{K\}$.

By (Proc Res) and (Proc GRes), $\llbracket \emptyset \rrbracket, S, k_{n+1}:K[\llbracket A \rrbracket] \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k : S \cup \{K\}$. \square

Lemma B.66 *If $H \models S \cdot (a, h, C) : A$ and $S \cdot (a, h, C) \rightarrow S' \cdot (a', h', C')$, then there is a heap typing H' , with $H \simeq H'$, such that $H + H' \models S' \cdot (a', h', C') : A$ and that, for all channel k with $k \notin (\text{dom}_2(H + H') \cup L)$, we have:*

$$\llbracket \emptyset \rrbracket, S, k:K[\llbracket A \rrbracket] \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$$

Proof By induction on the derivation of $S \cdot (a, h, C) \rightarrow S' \cdot (a', h', C')$. Assume $H \models S \cdot (a, h, C) : A$. Let $\vec{\rho} = \text{dom}(H) - S$, and $\mathcal{H} = \llbracket \text{ptr}(H) \rrbracket$, and \vec{k} be a stack, $[k_1, \dots, k_n]$, made of pairwise distinct variables such that $\{\vec{k}\} \cap (\text{dom}_2(H) \cup L \cup \{k\}) = \emptyset$. Let A_1, \dots, A_n be the sequence $\text{types}(C)$ and let T_i denote the π -calculus type $K[\llbracket A_i \rrbracket]$ for each $i \in 1..n$. In the remainder of this proof, we will use the notation $(\nu \vec{k}:\vec{T})P$ for the process $(\nu k_1:T_1) \cdots (\nu k_n:T_n)P$ and we will sometimes use k_{n+1} and A_{n+1} instead of k and A . We will also omit type annotations when they can be easily inferred from the environment. Let $E \triangleq \llbracket \emptyset \rrbracket, S, k:K[\llbracket B \rrbracket]$.

(Red Alloc) We have $a = (v \text{ at } \rho)$ and $a' = p$, where $\rho \in S$ and $p \notin \text{dom}_2(h)$, and $h' = h + (\rho \mapsto (h(\rho) + (p \mapsto v)))$, and $C' = C$. By (Config Good) and (Region Good), since $H \models S \cdot (a, h, C) : A$, there is a type B such that $\text{env}(H) \vdash a : \{\rho\} B$. Take $H' = (\rho \mapsto [p \mapsto v])$. Then:

$$\begin{aligned} H + H' &\models S' \cdot (a', h', C') : A \\ \llbracket S \cdot (a, h_H, C) \rrbracket k &= (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k}:\vec{T}) \\ &\quad ((\nu p:[A])(\llbracket p \mapsto v \rrbracket \mid \overline{k_1}(p)) \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \\ \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k &\equiv (\nu \vec{\rho})(\nu \mathcal{H})(\nu p:[A])(\nu \vec{k}:\vec{T}) \\ &\quad (\overline{k_1}(p) \mid (\llbracket h \rrbracket \mid \llbracket p \mapsto v \rrbracket) \mid \llbracket C \rrbracket [\vec{k}, k]) \end{aligned}$$

Therefore $\llbracket S \cdot (a, h_H, C) \rrbracket k \equiv \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$. By Lemma B.40 and Proposition B.65, $E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$.

(Red Appl) We have $a = p[\rho'_1, \dots, \rho'_n](q)$ and $a' = b\{f \leftarrow p\}\{x \leftarrow q\}\sigma$, where $h(\rho)(p)$ is a function $v = \mu(f:F)\lambda[\rho_1, \dots, \rho_n](x)b$, $S = S'$, $h = h'$, $C = C'$ and σ is the substitution $\{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$. Take $H' = \emptyset$. By Lemma B.11 and (Small Config Good), since $H \models S \cdot (a, h, C) : A$, we get that $H + H' \models S' \cdot (a', h', C') : A$. Moreover:

$$\begin{aligned} \llbracket S \cdot (a, h_H, C) \rrbracket k &= (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k}:\vec{T}) \\ &\quad (\overline{p}(\rho_1, \dots, \rho_n, q, k_1) \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \end{aligned}$$

where $\llbracket h \rrbracket \equiv (\llbracket (p \mapsto v) \rrbracket \mid Q)$, for some Q such that $p \notin \text{inp}(Q)$. By Proposition A.8 (5):

$$E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k}:\vec{T}) \\ (\llbracket b\{f \leftarrow p\} \rrbracket k_1 \{ \rho_1 \leftarrow \rho'_1 \} \cdots \{ \rho_n \leftarrow \rho'_n \} \{ x \leftarrow q \} \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k])$$

Hence, $E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$.

(Red Let) We have $a = (\text{let } x = a'_{A_0} \text{ in } b)$, and $C' = ((x:A_0)b :: C)$, and $S = S'$, and $h = h'$. Take $H' = \emptyset$. By (Control Good Cont) and (Small Config Good), $H + H' \models S' \cdot (a', h', C') : A$. Moreover:

$$\llbracket S \cdot (a, h_H, C) \rrbracket k = (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k}:\vec{T})((\nu k_0:K[\llbracket A \rrbracket])(\llbracket a' \rrbracket k_0 \mid \\ k_0(x:A_0).\llbracket b \rrbracket k_1) \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \\ \equiv (\nu \vec{\rho})(\nu \mathcal{H})(\nu k_0:K[\llbracket A_0 \rrbracket])(\nu \vec{k}:\vec{T}) \\ (\llbracket a' \rrbracket k_0 \mid \llbracket h \rrbracket \mid (k_0(x:A_0).\llbracket b \rrbracket k_1 \mid \llbracket C \rrbracket [\vec{k}, k]))$$

If $T_0 = K[\llbracket A_0 \rrbracket]$ then:

$$\llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k = (\nu \vec{\rho})(\nu \mathcal{H})(\nu k_0:T_0, \vec{k}:\vec{T}) \\ (\llbracket a' \rrbracket k_0 \mid \llbracket h \rrbracket \mid \llbracket (x:A_0)b :: C \rrbracket [k_0, \vec{k}, k])$$

Therefore, $\llbracket S \cdot (a, h_H, C) \rrbracket k \equiv \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$. By Lemma B.40, $E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$.

(Red Pop Let) We have $a = p$, and $a' = b\{x \leftarrow p\}$, and $C = (x:A_1)b :: C'$, and $S = S'$, and $h = h'$. Take $H' = \emptyset$. By Lemma B.11, we get that $H + H' \models S' \cdot (a', h', C') : A$. Let \vec{k}' be the sequence $[k_2, \dots, k_n]$.

$$\llbracket S \cdot (a, h_H, C) \rrbracket k = (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k}:\vec{T}) \\ (\overline{k_1}(p) \mid \llbracket h \rrbracket \mid (k_1(x:A_1).\llbracket b \rrbracket k_2) \mid \llbracket C \rrbracket [\vec{k}', k]))$$

Let \vec{T}' denote the sequence T_2, \dots, T_n . By Lemma A.5:

$$E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k}:\vec{T}') \\ (\llbracket b\{x \leftarrow p\} \rrbracket k_2 \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket (\vec{k}', k))$$

Hence, $E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$.

(Red Letregion) We have $a = (\text{letregion } \rho \text{ in } a')$, where $\rho \notin (S \cup \text{dom}(h))$, and $S' = S \cup \{\rho\}$, and $h' = h + \rho \mapsto \emptyset$, and $C = C'$. Take $H' = (\rho \mapsto \emptyset)$. Then:

$$\llbracket S \cdot (a, h_H, C) \rrbracket k = (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k}:\vec{T}) \\ (((\nu \rho)\llbracket a' \rrbracket k_1) \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \\ \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k = (\nu \rho)(\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k}:\vec{T}) \\ (\llbracket a' \rrbracket k_1 \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k])$$

Since the name ρ is bound, we may assume that it is different from the names in \mathcal{H} and \vec{k} . Therefore $\llbracket S \cdot (a, h_H, C) \rrbracket k \equiv \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$. By Lemma B.40, $E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$.

(Red Pop Letregion) We have $a = a' = p$, and $C = \text{popregion } \rho :: C'$, and $\rho \in S$, and $S' = S \setminus \{\rho\}$, and $h = h'$. Take $H' = \emptyset$. Then $H + H' \models S' \cdot (a', h', C') : A$, and:

$$\llbracket S \cdot (a, h_H, C) \rrbracket k = (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k} : \vec{T})(\overline{k_1} \langle p \rangle \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k])$$

where $\llbracket C \rrbracket [\vec{k}, k] = \llbracket C' \rrbracket [\vec{k}, k]$. Therefore, $\llbracket S \cdot (a, h_H, C) \rrbracket k \equiv \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$. By Lemma B.40, $E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$.

(Red Case 1) Then $a = \text{case } p \text{ of nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2$ and $a' = b_1$ where $\rho \in S$ and $h(\rho)(p) = \text{nil}$ and $S = S'$ and $h = h'$. Take $H' = \emptyset$. Therefore $H + H' \models S' \cdot (a', h', C') : A$ and:

$$\begin{aligned} \llbracket S \cdot (a, h_H, C) \rrbracket k &= (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k} : \vec{T}) \\ &((\nu z_1)(\nu z_2)(\overline{p} \langle z_1, z_2 \rangle \mid z_1().\llbracket b_1 \rrbracket k \mid z_2(y_1, y_2).\llbracket b_2 \rrbracket k) \\ &\mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \end{aligned}$$

where $\llbracket h \rrbracket \equiv (\llbracket p \mapsto \llbracket \text{nil} \rrbracket \rrbracket \mid Q)$ for some Q such that $p \notin \text{inp}(Q)$. By Proposition A.8 (5):

$$\begin{aligned} E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k &\approx (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k} : \vec{T}) \\ &((\nu z_1)(\nu z_2)(\overline{\text{nil}} \langle \rangle \mid z_1().\llbracket b_1 \rrbracket k \mid z_2(y_1, y_2).\llbracket b_2 \rrbracket k) \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \end{aligned}$$

By Lemma A.5,

$$\begin{aligned} E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k &\approx (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k} : \vec{T}) \\ &((\nu z_2)(z_2(y_1, y_2).\llbracket b_2 \rrbracket k) \mid \llbracket b_1 \rrbracket k \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \end{aligned}$$

By Lemma A.7, $E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$.

(Red Case 2) Then $a = \text{case } p \text{ of nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2$ and $a' = b_2 \{y_1 \leftarrow q_1\} \{y_2 \leftarrow q_2\}$ where $\rho \in S$ and $h(\rho)(p) = q_1 :: q_2$ and $S = S'$ and $h = h'$. Take $H' = \emptyset$. Therefore $H + H' \models S' \cdot (a', h', C') : A$ and:

$$\begin{aligned} \llbracket S \cdot (a, h_H, C) \rrbracket k &= (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k} : \vec{T}) \\ &((\nu z_1)(\nu z_2)(\overline{p} \langle z_1, z_2 \rangle \mid z_1().\llbracket b_1 \rrbracket k \mid z_2(y_1, y_2).\llbracket b_2 \rrbracket k) \\ &\mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \end{aligned}$$

where $\llbracket h \rrbracket \equiv (\llbracket p \mapsto [q_1 :: q_2] \rrbracket \mid Q)$ for some Q such that $p \notin \text{inp}(Q)$. By Proposition A.8 (5):

$$\begin{aligned} E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k &\approx (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k} : \vec{T})(\nu z_1)(\nu z_2) \\ &(\overline{\text{nil}} \langle q_1, q_2 \rangle \mid z_1().\llbracket b_1 \rrbracket k \mid z_2(y_1, y_2).\llbracket b_2 \rrbracket k \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]) \end{aligned}$$

By Lemma A.5,

$$E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx (\nu \vec{\rho})(\nu \mathcal{H})(\nu \vec{k} : \vec{T})(\nu z_1) \\ (z_1(). \llbracket b_1 \rrbracket k) \mid \llbracket b_2 \rrbracket k \{y_1 \leftarrow q_1\} \{y_2 \leftarrow q_2\} \mid \llbracket h \rrbracket \mid \llbracket C \rrbracket [\vec{k}, k]$$

By Lemma A.7, $E \vdash \llbracket S \cdot (a, h_H, C) \rrbracket k \approx \llbracket S' \cdot (a', h'_{H+H'}, C') \rrbracket k$. \square

The following asserts that the encoding of the extended region calculus preserves the dynamic semantics. A proof of Theorem 4.2, dynamic adequacy for the unextended calculi, can be obtained by defining an auxiliary (unextended) small-step semantics for the region calculus and simplifying the following proof.

Theorem B.67 *If $H \models S \cdot (a, h) : A$ and $S \cdot (a, h) \Downarrow (p', h')$ then there is H' such that $H \simeq H'$ and $H + H' \models S \cdot (p', h') : A$ and for all $k \notin \text{dom}_2(H + H') \cup L$, $\llbracket \emptyset \rrbracket, S, k : K \llbracket [A] \rrbracket \vdash \llbracket S \cdot (a, h_H) \rrbracket k \approx \llbracket S \cdot (p', h'_{H+H'}) \rrbracket k$.*

Proof Assume $H \models S \cdot (a, h) : A$ and $S \cdot (a, h) \Downarrow (p', h')$. By Theorem B.62, we have $S \cdot (a, h, \square) \rightarrow^* S \cdot (p', h', \square)$. By rule (Control Good Empty) and (Small Config Good), we have $H \models S \cdot (a, h, \square) : A$. By Lemma B.66, there is a heap typing H' , with $H \simeq H'$, such that $H + H' \models S \cdot (p', h', \square) : A$ and that, for all channel k with $k \notin (\text{dom}_2(H + H') \cup L)$, we have:

$$\llbracket \emptyset \rrbracket, S, k : K \llbracket [A] \rrbracket \vdash \llbracket (H, S, a, h, \square) \rrbracket k \approx \llbracket (H + H', S, p', h', \square) \rrbracket k$$

By Lemma B.63, $\llbracket \emptyset \rrbracket, S, k : K \llbracket [A] \rrbracket \vdash \llbracket S \cdot (a, h_H) \rrbracket k \approx \llbracket (S, H + H', p', h') \rrbracket k$. \square

B.6.4 Proof of Garbage Collection for the λ -Calculus

The following property asserts that defunct regions make no difference to the behaviour of a program. It corresponds to Theorem 4.4 for the unextended calculi.

Theorem B.68 *Suppose $H \models S \cdot (a, h) : A$ and $k \notin \text{dom}_2(H) \cup L$. Let $\{\vec{\rho}_{\text{defunct}}\} = \text{dom}(H) - S$. Then:*

$$\llbracket \emptyset \rrbracket, S, k : K \llbracket [A] \rrbracket \vdash \llbracket S \cdot (a, h) \rrbracket k \\ \approx (\nu \vec{\rho}_{\text{defunct}})(\nu \llbracket ptr(H) \rrbracket)(\llbracket a \rrbracket k \mid \prod_{\rho \in S} \llbracket H(\rho) \rrbracket)$$

Proof Let $\vec{\rho}$ be a sequence of groups, ρ_1, \dots, ρ_m , such that $\{\vec{\rho}\} = S$. Let $\vec{\rho}'_{\text{defunct}}$ be a sequence of groups, ρ'_1, \dots, ρ'_n , such that $\{\vec{\rho}'_{\text{defunct}}\} = \text{dom}(H) - S$. For the sake of brevity, we use the symbol $\vec{\rho}'_{\times}$ instead of $\vec{\rho}'_{\text{defunct}}$ in the remainder of this proof. In particular $(\{\vec{\rho}\} \cup \{K\}) \cap \{\vec{\rho}'_{\times}\} = \emptyset$. Let:

$$\begin{aligned} h &= \vec{\rho} \mapsto \vec{r}, \vec{\rho}'_{\times} \mapsto \vec{r}'_{\times} \\ H &= \vec{\rho} \mapsto \vec{R}, \vec{\rho}'_{\times} \mapsto \vec{R}'_{\times} \\ \text{env}(H) &= \vec{\rho}, \vec{\rho}'_{\times}, \vec{r} \text{ at } \vec{\rho}, \vec{r}'_{\times} \text{ at } \vec{\rho}'_{\times} \end{aligned}$$

By (Config Good), $H \models S \cdot (a, h) : A$ implies $env(H) \vdash a :^e A$ and $e \cup fg(A) \subseteq S$ and $H \models S \cdot h$. By Theorem 4.1, we have that:

$$\begin{aligned} & \llbracket \emptyset \rrbracket, \vec{\rho}, \vec{\rho}_\times, \llbracket \vec{r} \text{ at } \vec{\rho} \rrbracket, \llbracket \vec{r}_\times \text{ at } \vec{\rho}_\times \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k : e \cup \{K\} \\ & \llbracket \emptyset \rrbracket, \vec{\rho}, \vec{\rho}_\times, \llbracket \vec{r} \text{ at } \vec{\rho} \rrbracket, \llbracket \vec{r}_\times \text{ at } \vec{\rho}_\times \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket \vec{\rho} \mapsto \vec{r} \rrbracket : \{\vec{\rho}\} \\ & \llbracket \emptyset \rrbracket, \vec{\rho}, \vec{\rho}_\times, \llbracket \vec{r} \text{ at } \vec{\rho} \rrbracket, \llbracket \vec{r}_\times \text{ at } \vec{\rho}_\times \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket \vec{\rho}_\times \mapsto \vec{r}_\times \rrbracket : \{\vec{\rho}_\times\} \end{aligned}$$

Let $P = \llbracket a \rrbracket k \mid \llbracket \vec{\rho} \mapsto \vec{r} \rrbracket$. By an exchange lemma, we get:

$$\begin{aligned} & \llbracket \emptyset \rrbracket, \vec{\rho}, k:K[\llbracket A \rrbracket], \vec{\rho}_\times, \llbracket \vec{r} \text{ at } \vec{\rho} \rrbracket, \llbracket \vec{r}_\times \text{ at } \vec{\rho}_\times \rrbracket \vdash P : \{\vec{\rho}, K\} \\ & \llbracket \emptyset \rrbracket, \vec{\rho}, k:K[\llbracket A \rrbracket], \vec{\rho}_\times, \llbracket \vec{r} \text{ at } \vec{\rho} \rrbracket, \llbracket \vec{r}_\times \text{ at } \vec{\rho}_\times \rrbracket \vdash \llbracket \vec{\rho}_\times \mapsto \vec{r}_\times \rrbracket : \{\vec{\rho}_\times\} \end{aligned}$$

By Theorem B.53 several times, we get:

$$\begin{aligned} & \llbracket \emptyset \rrbracket, S, k:K[\llbracket A \rrbracket] \vdash (\nu \vec{\rho}_\times)(\nu \llbracket \vec{r} \text{ at } \vec{\rho} \rrbracket)(\nu \llbracket \vec{r}_\times \text{ at } \vec{\rho}_\times \rrbracket) \\ & (P \mid \llbracket \vec{\rho}_\times \mapsto \vec{r}_\times \rrbracket) \approx (\nu \vec{\rho}_\times)(\nu \llbracket \vec{r} \text{ at } \vec{\rho} \rrbracket)(\nu \llbracket \vec{r}_\times \text{ at } \vec{\rho}_\times \rrbracket)P \end{aligned}$$

But this is:

$$\begin{aligned} & \llbracket \emptyset \rrbracket, S, k:K[\llbracket A \rrbracket] \vdash \llbracket S \cdot (a, h) \rrbracket k \\ & \approx (\nu \vec{\rho}_{\text{defunct}})(\nu \llbracket ptr(H) \rrbracket)(\llbracket a \rrbracket k \mid \prod_{\rho \in S} \llbracket H(\rho) \rrbracket) \end{aligned}$$

□

B.7 An Equational Theory

We now prove that the equational theory for the region calculus is sound with respect to our encoding in the π -calculus with groups. This property is given by Theorem 5.2, that the encoding of equivalent expressions are (barbed) equivalent processes. In this appendix we consider the simple region calculus introduced in Section 2. For the sake of brevity we have not considered the details of how to extend this theory to the polymorphic region calculus.

We start by proving Lemma B.69, that the encoding of a term obtained by substituting an allocation v at ρ , to a variable x , in a term b , is equivalent to the process obtained by substituting to x in $\llbracket b \rrbracket$, a private link to a replicated copy of the process $\llbracket v \rrbracket$. This property is used in the proof of Theorem 5.2. More precisely, it is needed in the proof that the encoding of β -equivalent terms are equivalent processes.

Note that this is the only result of this paper that relies on the locality restriction imposed on the π -calculus.

Lemma B.69 *Consider two expressions a and b such that a is an allocation, v at ρ , with $E \vdash a :^{\{\rho\}} A$ and $E, x:A \vdash b :^e B$ and $E \vdash b\{x \leftarrow a\} :^{e'} B$. If $p \notin fv(v) \cup fv(b)$ and $k \notin dom(E) \cup L$ then:*

$$\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu p:\llbracket A \rrbracket)(\llbracket p \mapsto v \rrbracket \mid \llbracket b\{x \leftarrow p\} \rrbracket k)$$

Proof By induction on the structure of b . Let a be an allocation, v at ρ . Assume $E \vdash a :^{\{\rho\}} A$ and $E, x:A \vdash b :^e B$ and $E \vdash b\{x \leftarrow a\} :^{e'} B$. Let $k \notin dom(E) \cup L$. By Theorem 4.1 (3), $\llbracket E \rrbracket, x:\llbracket A \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b \rrbracket k : e \cup \{K\}$

and $\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k : e' \cup \{K\}$. By Lemma B.56, $\llbracket E \rrbracket, p:[A] \vdash \llbracket p \mapsto v \rrbracket : \{\rho\}$. By (Proc Par), (Proc Res) and Lemma B.28, $\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash (\nu p:[A])(\llbracket p \mapsto v \rrbracket \mid \llbracket b\{x \leftarrow p\} \rrbracket k)$. Hence $\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k$ and $\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash (\nu p:[A])(\llbracket p \mapsto v \rrbracket \mid \llbracket b\{x \leftarrow p\} \rrbracket k)$.

For the sake of brevity, we omit the type annotations in the encoding of region calculus terms in the remainder of this proof.

(Variable) Then $b \triangleq y$. If $y = x$ then $\llbracket b\{x \leftarrow a\} \rrbracket k = \llbracket v \text{ at } \rho \rrbracket k = (\nu p)(\llbracket p \mapsto v \rrbracket \mid \llbracket p \rrbracket k)$, as required. If $y \neq x$ then $\llbracket b\{x \leftarrow a\} \rrbracket k = \llbracket y \rrbracket k$. By Lemma A.7, since $p \notin \text{fv}(\llbracket y \rrbracket k)$, we get that $\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash (\nu p)(\llbracket p \mapsto v \rrbracket \mid \llbracket y \rrbracket k) \approx \llbracket y \rrbracket k$, as required.

(Allocation) Then $b \triangleq (\lambda(y)c \text{ at } \rho')$ and $\llbracket b\{x \leftarrow a\} \rrbracket k$ is the replicated resource $\text{def } q(y, k) = \llbracket c\{x \leftarrow a\} \rrbracket k \text{ in } \bar{k}\langle q \rangle$. By induction hypothesis, and since \approx is a congruence, we get that:

$$\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx \text{def } q(y, k) = (\nu p)(\llbracket p \mapsto v \rrbracket \mid \llbracket c\{x \leftarrow p\} \rrbracket k) \text{ in } \bar{k}\langle q \rangle$$

By Propositions A.8 (1) and A.8 (3), we get that

$$\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu p)(\llbracket p \mapsto v \rrbracket \mid \llbracket b\{x \leftarrow p\} \rrbracket k)$$

(Application) Then $b \triangleq y(z)$. Assume $y = x$. Since the term b is well-typed, it must be the case that $z \neq x$. Hence, $\llbracket b\{x \leftarrow a\} \rrbracket k = (\nu k', p)(\llbracket p \mapsto v \rrbracket \mid \bar{k}'\langle p \rangle \mid k'(x').\llbracket x'(z) \rrbracket k)$. By Lemma A.5, $\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu p)(\llbracket p \mapsto v \rrbracket \mid \llbracket p(z) \rrbracket k)$, as required. Assume $z = x$. Since the term b is well-typed, it must be the case that $y \neq x$. Hence, $\llbracket b\{x \leftarrow a\} \rrbracket k = (\nu k', p)(\llbracket p \mapsto v \rrbracket \mid \bar{k}'\langle p \rangle \mid k'(x').\llbracket y(x') \rrbracket k)$. By Lemma A.5, $\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu p)(\llbracket p \mapsto v \rrbracket \mid \llbracket y(p) \rrbracket k)$, as required.

(Sequencing) Then $b \triangleq \text{let } y = c_1 \text{ in } c_2$ and $\llbracket b\{x \leftarrow a\} \rrbracket k$ is the process $(\nu k')(\llbracket c_1\{x \leftarrow a\} \rrbracket k' \mid k'(y).\llbracket c_2\{x \leftarrow a\} \rrbracket k)$. By induction hypothesis, Proposition A.8 (4), and since \approx is a congruence, we get that:

$$\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu k')((\nu p_1)(\llbracket p_1 \mapsto v \rrbracket \mid \llbracket c_1\{x \leftarrow p_1\} \rrbracket k') \mid (\nu p_2)(\llbracket p_2 \mapsto v \rrbracket \mid k'(y).\llbracket c_2\{x \leftarrow p_2\} \rrbracket k))$$

Assume $p \notin \text{fv}(v) \cup \text{fv}(b)$. Since $p_1, p_2 \notin \text{fv}(v)$, we get that:

$$\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu k')((\nu p)(\llbracket p \mapsto v \rrbracket \mid \llbracket c_1\{x \leftarrow p\} \rrbracket k') \mid (\nu p)(\llbracket p \mapsto v \rrbracket \mid k'(y).\llbracket c_2\{x \leftarrow p\} \rrbracket k))$$

By Proposition A.8 (2), $\llbracket E \rrbracket, k:K[\llbracket B \rrbracket] \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu p)(\llbracket p \mapsto v \rrbracket \mid (\nu k')(\llbracket c_1\{x \leftarrow p\} \rrbracket k' \mid k'(y).\llbracket c_2\{x \leftarrow p\} \rrbracket k))$, as required

(Letregion) Then $b \triangleq \text{letregion } \rho' \text{ in } c$. Hence, $\llbracket b\{x \leftarrow a\} \rrbracket k = (\nu \rho') \llbracket c\{x \leftarrow a\} \rrbracket k$. Assume $p \notin \text{fv}(v) \cup \text{fv}(b)$. By induction hypothesis, and since \approx is a congruence, we get that: $\llbracket E \rrbracket, k : K \llbracket \llbracket B \rrbracket \rrbracket \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu \rho') (\nu p) (\llbracket p \mapsto v \rrbracket \mid \llbracket c\{x \leftarrow p\} \rrbracket k)$. By Proposition 3.4 (4), since $\rho' \notin \text{fr}(a)$ and $\text{fg}(\llbracket v \rrbracket k) = \text{fr}(a) \cup \{K\}$, we get that $\llbracket E \rrbracket, k : K \llbracket \llbracket B \rrbracket \rrbracket \vdash \llbracket b\{x \leftarrow a\} \rrbracket k \approx (\nu p) (\llbracket p \mapsto v \rrbracket \mid (\nu \rho') \llbracket c\{x \leftarrow p\} \rrbracket k)$, as required. \square

Proof of Lemma 5.1 If $E \vdash a_1 \leftrightarrow a_2 : A$ then there is $e \subseteq \text{dom}(E)$ such that for each $i \in 1..2$, there is $e_i \subseteq e$ with $E \vdash a_i :^{e_i} A$.

Proof By induction on the derivation of $E \vdash a_1 \leftrightarrow a_2 : A$.

(Eq Refl) and (Eq Symm) Trivial.

(Eq Trans) Then $E \vdash a_1 \leftrightarrow b : A$ and $E \vdash b \leftrightarrow a_2 : A$. By induction hypothesis, there are effects $f^1, f^2 \subseteq \text{dom}(E)$ such that $E \vdash a_1 :^{f^1} A$, $E \vdash b :^{f^2} A$, $E \vdash a_2 :^{f^1} A$ and $E \vdash b :^{f^2} A$, where for each $i, j \in 1..2$, $f_j^i \subseteq f^i$. Take $e = f^1 \cup f^2$ and $e_i = f_1^i$ for each $i \in 1..2$.

(Eq Fun) Then $E, x : A' \vdash b_i :^{e_i} B'$ for each $i \in 1..2$ and $A \triangleq (A' \xrightarrow{e'} B')$ at ρ , where $a_i \triangleq (\lambda(x:A') b_i)$ at ρ and $e_i \subseteq e'$ and $E \vdash A$. Take $e = \{\rho\}$. By (Type \rightarrow), since $E \vdash A$, we get that $e \subseteq \text{dom}(E)$. By (Exp Fun), $E \vdash a_i :^e A$ for each $i \in 1..2$, as required.

(Eq Fun β) and (Eq Let β) Then $a_1 \triangleq \text{let } y = (\lambda(x:B) b \text{ at } \rho) \text{ in } y(a)$ and $a_2 \triangleq b\{x \leftarrow a\}$ where a is a name or an allocation, and $y \notin \text{fv}(a)$, and $E \vdash a :^{e_1} B$, and $E, x : B \vdash b :^{e_2} A$, and $E \vdash b\{x \leftarrow a\} :^{e_3} A$ (that is, $E \vdash a_2 :^{e_3} A$), and $\rho \in \text{dom}(E)$. Take $e = \{\rho\} \cup e_1 \cup e_2$. By (Exp Appl) and (Exp Let), we get that $E \vdash a_1 :^e A$, as required. The case for (Eq Let β) is similar.

(Eq Let) and (Eq Letregion Let) Then $a_1 \triangleq \text{let } x = a \text{ in } b$ and $a_2 \triangleq \text{let } x = a' \text{ in } b'$ where $E \vdash a \leftrightarrow a' : B$ and $E, x : B \vdash b \leftrightarrow b' : A$. By induction hypothesis, there is $e^1, e^2 \subseteq \text{dom}(E)$ such that for each $i, j \in 1..2$ there is $e_i^j \subseteq e^j$ with $E \vdash a :^{e_1^1} B$, $E \vdash a' :^{e_2^1} B$, $E, x : B \vdash b :^{e_1^2} A$ and $E, x : B \vdash b' :^{e_2^2} A$. Take $e_i = e_1^i \cup e_2^i$ for each $i \in 1..2$ and $e = e_1 \cup e_2$. By (Exp Let), we get that $E \vdash a_i :^{e_i} A$ and $e_i \subseteq e$ for each $i \in 1..2$, as required. The case for (Eq Letregion Let) is similar.

(Eq Let Assoc) Then $a_1 \triangleq \text{let } x = a \text{ in } (\text{let } y = b \text{ in } c)$ and $a_2 \triangleq \text{let } y = (\text{let } x = a \text{ in } b) \text{ in } c$ where $E \vdash a :^{e_a} A$ and $E, x : A \vdash b :^{e_b} B$ and $E, y : B \vdash c :^{e_c} C$. In particular, since $x \notin \text{dom}(E, y : B)$, we get that $x \notin \text{fv}(c)$ and $E, y : B, x : A \vdash \diamond$. Take $e = e_1 = e_2 = e_a \cup e_b \cup e_c$. By (Exp Let) and Lemma B.6, we get that $E \vdash a_i :^{e_i} C$ for each $i \in 1..2$, as required.

(Eq Letregion) and (Eq Swap) Then $a_i = (\nu\rho)b_i$ for each $i \in 1..2$ where $E, \rho \vdash b_1 \leftrightarrow b_2 : A$ and $\rho \notin fr(A)$. By induction hypothesis, there is $f \subseteq dom(E, \rho)$ such that for each $i \in 1..2$, there is $f_i \subseteq f$ with $E, \rho \vdash b_i :^{f_i} A$. Take $e = f - \{\rho\}$ and $e_i = f_i - \{\rho\}$ for each $i \in 1..2$. By (Exp Letregion), $E \vdash a_i :^{e_i} A$ for each $i \in 1..2$, as required. Case (Eq Swap) is similar.

(Eq Drop) Then $a_1 = (\nu\rho)a_2$ where $E \vdash a :^e A$ and $\rho \notin dom(E)$. Take $e_1 = e_2 = e$. Since $e \subseteq dom(E)$, we get that $e = e - \{\rho\}$ and $\rho \notin fr(A)$. By (Exp Letregion), $E \vdash a_1 :^e A$, as required. \square

Proof of Theorem 5.2 Suppose $E \vdash a \leftrightarrow b : A$ and $k \notin dom(E) \cup L$. Then $\llbracket E \rrbracket, k : K \llbracket [A] \rrbracket \vdash \llbracket a \rrbracket k \approx \llbracket b \rrbracket k$.

Proof By induction on the derivation of $E \vdash a \leftrightarrow b : A$. By Lemma 5.1, there is $e \subseteq dom(E)$ and e_1, e_2 such that $e_i \subseteq e$ for each $i \in 1..2$ with $E \vdash a :^{e_1} A$ and $E \vdash b :^{e_2} A$. Let $k \notin dom(E) \cup L$. By Theorem 4.1 (3), $\llbracket E \rrbracket, k : K \llbracket [A] \rrbracket \vdash \llbracket a \rrbracket k : e_1 \cup \{K\}$ and $\llbracket E \rrbracket, k : K \llbracket [A] \rrbracket \vdash \llbracket b \rrbracket k : e_2 \cup \{K\}$. Hence, $\llbracket E \rrbracket, k : K \llbracket [A] \rrbracket \vdash \llbracket a \rrbracket k, \llbracket b \rrbracket k$.

For the sake of brevity, we omit the type annotations in the encoding of region calculus terms in the remainder of this proof.

(Eq Refl), (Eq Symp) and (Eq Trans) Trivial, since \approx is an equivalence relation.

(Eq Fun), (Eq Let) and (Eq Letregion) Trivial, since \approx is a congruence.

(Eq Fun β) and (Eq Let β) Then $a \triangleq let\ y = (\lambda(x:B)b' \text{ at } \rho) \text{ in } y(a')$ and $b \triangleq b'\{x \leftarrow a'\}$ where a' is a name or an allocation, and $y \notin fv(a')$. Hence, $\llbracket a \rrbracket k \triangleq (\nu k') def\ p(x, k) = \llbracket b' \rrbracket k \text{ in } (\overline{k'}\langle p \mid k'(y). \llbracket y(a') \rrbracket k)$ where k' and p are fresh names. By Lemma A.5, $\llbracket E \rrbracket, k : K \llbracket [A] \rrbracket \vdash \llbracket a \rrbracket k \approx def\ p(x, k) = \llbracket b' \rrbracket k \text{ in } \llbracket p(a') \rrbracket k$. We have two possible cases depending on the shape of a' .

Assume a' is a name, say q . Hence, $\llbracket E \rrbracket, k : K \llbracket [A] \rrbracket \vdash \llbracket a \rrbracket k \approx def\ p(x, k) = \llbracket b' \rrbracket k \text{ in } \overline{p}\langle q, k \rangle$. By Proposition A.8 (5):

$$\llbracket E \rrbracket, k : K \llbracket [A] \rrbracket \vdash \llbracket a \rrbracket k \approx def\ p(x, k) = \llbracket b' \rrbracket k \text{ in } \llbracket b'\{x \leftarrow a'\} \rrbracket k$$

By Proposition A.8 (1), since $p \notin fv(b'\{x \leftarrow a'\})$, we get that:

$$\llbracket E \rrbracket, k : K \llbracket [A] \rrbracket \vdash \llbracket a \rrbracket k \approx \llbracket b'\{x \leftarrow a'\} \rrbracket k$$

Assume a' is an allocation, say $(\lambda(y)c \text{ at } \rho')$. Hence:

$$\begin{aligned}
\llbracket E \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k &\approx \text{def } p(x, k) = \llbracket b' \rrbracket k \text{ in} \\
&\quad (\nu k')(\text{def } q(y, k) = \llbracket c \rrbracket k \text{ in} \\
&\quad \quad (\overline{k'}\langle q \rangle \mid k'(y).\overline{p}\langle y, k \rangle)) \\
&\quad \quad \quad \text{(By Lemma A.5)} \\
&\approx \text{def } p(x, k) = \llbracket b' \rrbracket k \text{ in} \\
&\quad (\text{def } q(y, k) = \llbracket c \rrbracket k \text{ in } \overline{p}\langle q, k \rangle) \\
&\quad \quad \quad \text{(By Proposition A.8 (5))} \\
&\approx \text{def } p(x, k) = \llbracket b' \rrbracket k \text{ in} \\
&\quad (\text{def } q(y, k) = \llbracket c \rrbracket k \text{ in } \llbracket b' \{x \leftarrow q\} \rrbracket k) \\
&\quad \quad \quad \text{(By Lemma B.69)} \\
&\approx \text{def } p(x, k) = \llbracket b' \rrbracket k \text{ in } \llbracket b' \{x \leftarrow a'\} \rrbracket k
\end{aligned}$$

Case (Eq Let β) is similar.

(Eq Let Assoc) Then $a \triangleq \text{let } x = a' \text{ in } (\text{let } y = b' \text{ in } c')$ and $b \triangleq \text{let } y = (\text{let } x = a' \text{ in } b') \text{ in } c'$ where $E \vdash a' :^{e_1} A$ and $E, x : A \vdash b' :^{e_2} B$ and $E, y : B \vdash c' :^{e_3} C$. In particular, since $x \notin \text{dom}(E, y : B)$, we get that $x \notin \text{fv}(c')$. Hence, $\llbracket a \rrbracket k \triangleq (\nu k_1)(\llbracket a' \rrbracket k_1 \mid k_1(x).(\nu k_2)(\llbracket b' \rrbracket k_2 \mid k_2(y).\llbracket c' \rrbracket k))$ and $\llbracket b \rrbracket k \triangleq (\nu k_1)(\llbracket a' \rrbracket k_1 \mid (\nu k_2)(k_1(x).\llbracket b' \rrbracket k_2 \mid k_2(y).\llbracket c' \rrbracket k))$, where k_1, k_2 are two fresh names. By Lemma A.6, $\llbracket E \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k \approx (\nu k_1)(\llbracket a' \rrbracket k_1 \mid (\nu k_2)k_1(x).(\llbracket b' \rrbracket k_2 \mid k_2(y).\llbracket c' \rrbracket k))$. By Lemma A.9, we get that $\llbracket E \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k \approx \llbracket b \rrbracket k$, as desired.

(Eq Drop), (Eq Swap) and (Eq Letregion Let) In each of these cases we have $\text{erase}(\llbracket a \rrbracket k) = \text{erase}(\llbracket b \rrbracket k)$, $\llbracket E \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k$ and $\llbracket E \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket b \rrbracket k$. By Proposition B.39, $\llbracket E \rrbracket, k:K[\llbracket A \rrbracket] \vdash \llbracket a \rrbracket k \approx \llbracket b \rrbracket k$. \square