JFP **12** (3): 229–292, May 2002.
DOI: 10.1017/S0956796801004270 Printed in the United Kingdom

Region analysis and a π-calculus with groups

SILVANO DAL ZILIO and ANDREW D. GORDON

Microsoft Research,
7 J J Thomson Avenue, Cambridge CB3 0FB, United Kingdom
(*e-mail:* adg@microsoft.com)

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.

Capsule Review

The paper presents a correctness proof for Tofte and Talpin's region calculus based on a translation into the pi-calculus. Generally, such a translation technique allows to reason about higher-order languages (like the region calculus) in a first- order setting (the pi-calculus), and to exploit the powerful proof-machinery that has been developed for the pi-calculus. I consider the work to be especially in line with pi-calculus encodings of the lambda-calculus by Milner and Sangiorgi, for the following reasons: (1) the region calculus is an extension of the lambda-calculus with regions; (2) for the encoding, an extension of the pi-calculus with groups is used; these groups correspond exactly to the regions; (3) the encoding inherits a lot from encodings of the lambda-calculus; in fact, regions and groups turn out to be orthogonal in the sense that adding them does not harm the validity of the methods applied to the lambda-calculus parts of the encoding.

1 Motivation

This paper reports a new proof of correctness of region-based memory management (Tofte & 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 *et al.*, 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.

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 & Walker, 2001; Walker, 1995), for the design of concurrent programming languages (Fournet & Gonthier, 1996; Pierce & Turner, 2000), and for the study of security protocols (Abadi & 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 & Sangiorgi, 1996), names may not occur in types.

We solve the problem of modelling regions by defining a typed π-calculus equipped with named groups and a new-group construct (Cardelli *et al.*, 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 & Walker, 2001). 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 group-based garbage collection principle expressed in the π-calculus.
- Secondly, the semantics provides a more abstract account of the behaviour of 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.

Overall, the paper makes a new connection between two programming language constructs, regions and groups, that were proposed independently and for different purposes. A benefit for the theory of the π-calculus is the discovery of a new confinement principle, Theorem 4.3. Although our proof of this theorem involves a substantial theoretical development, we anticipate it will be of use in other settings, such as the study of other source languages with regions. A benefit for the theory of the region calculus is that once we have proved this principle, we can write down a strikingly simple proof of the soundness of region-based memory management.

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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Appendix A reviews the untyped π-calculus. Appendix B describes proofs of all properties stated without proof in the main text.

An abridged version of this work appears as a conference paper (Dal Zilio & Gordon, 2000a).

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 & 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 $L = \{\ell_1, \ldots, \ell_n\}$ 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
$\quad x$	$\quad$ name
$\quad v \ at \ \rho$	$\quad$ allocation of v at ρ
$\quad x(y)$	$\quad$ application
$\quad let \ x = a \ in \ b$	$\quad$ sequencing
$\quad letregion \ \rho \ in \ b$	$\quad$ region allocation, de-allocation
$u, v ::=$	boxed value
$\quad \lambda(x{:}A)b$	$\quad$ 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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

let $f = b$ *in let* $x = a$ *in* $f(x)$, where $f \neq x$ and f is not free in a. This reduced syntax for application allows a more concise presentation of the operational semantics than the conventional syntax.

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

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

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 & 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
$S ::= \{\rho_1, \ldots, \rho_n\}$	stack of live regions

A region r is a finite map of the form $p_1 \mapsto v_1, \ldots, p_n \mapsto v_n$, 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 $\{p_1, \ldots, p_n\}$. We write $\varnothing$ 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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

$p = p_j$ 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)

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

(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)

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

(Eval Letregion)

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

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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

2.3 Static semantics

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

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

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 $\rho_1, \ldots, \rho_n$. The set of regions $\{\rho_1,\ldots,\rho_n\}$ 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 $(A \xrightarrow{e} B)\ at\ \rho$, 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:

$e ::= \{\rho_1,\ldots,\rho_n\}$	effect
$A, B ::=$	type of expressions
$\quad Lit$	$\quad$ type of literals
$\quad (A \xrightarrow{e} B)\ at\ \rho$	$\quad$ type of functions stored in ρ
$E ::=$	environment
$\quad \emptyset$	$\quad$ empty environment
$\quad E, \rho$	$\quad$ entry for a region ρ
$\quad E, x{:}A$	$\quad$ 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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Type and Effect Rules:

(Env $\emptyset$) $\dfrac{}{\emptyset \vdash \diamond}$

(Env x) (recall L is the set of literals) $\dfrac{E \vdash A \quad x \notin dom(E) \cup L}{E, x{:}A \vdash \diamond}$

(Env ρ) $\dfrac{E \vdash \diamond \quad \rho \notin dom(E)}{E, \rho \vdash \diamond}$

(Type Lit) $\dfrac{E \vdash \diamond}{E \vdash Lit}$

(Type $\rightarrow$) $\dfrac{E \vdash A \quad \{\rho\} \cup e \subseteq dom(E) \quad E \vdash B}{E \vdash (A \xrightarrow{e} B)\ at\ \rho}$

(Exp x) $\dfrac{E, x{:}A, E' \vdash \diamond}{E, x{:}A, E' \vdash x :^{\emptyset} A}$

(Exp ℓ) $\dfrac{E \vdash \diamond \quad \ell \in L}{E \vdash \ell :^{\emptyset} Lit}$

(Exp Appl) $\dfrac{E \vdash x :^{\emptyset} (B \xrightarrow{e} A)\ at\ \rho \quad E \vdash y :^{\emptyset} B}{E \vdash x(y) :^{\{\rho\} \cup e} A}$

(Exp Let) $\dfrac{E \vdash a :^{e} A \quad E, x{:}A \vdash b :^{e'} B}{E \vdash let\ x = a\ in\ b :^{e \cup e'} B}$

(Exp Letregion) $\dfrac{E, \rho \vdash a :^{e} A \quad \rho \notin fr(A)}{E \vdash letregion\ \rho\ in\ a :^{e - \{\rho\}} A}$

(Exp Fun) $\dfrac{E, x{:}A \vdash b :^{e} B \quad e \subseteq e' \quad \{\rho\} \cup e' \subseteq dom(E)}{E \vdash \lambda(x{:}A)b\ at\ \rho :^{\{\rho\}} (A \xrightarrow{e'} B)\ at\ \rho}$

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 $\rho \notin fr(A)$ 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 $e' \subseteq dom(E)$ 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:

$$\rho, \rho' \vdash (\lambda(x{:}Lit)x)\ at\ \rho' :^{\{\rho'\}} (Lit \xrightarrow{\emptyset} Lit)\ at\ \rho'$$
$$\rho, \rho', f{:}(Lit \xrightarrow{\emptyset} Lit)\ at\ \rho' \vdash (\lambda(x{:}Lit)f(x))\ at\ \rho :^{\{\rho\}} (Lit \xrightarrow{\{\rho'\}} Lit)\ at\ \rho$$
$$\rho, \rho', f{:}(Lit \xrightarrow{\emptyset} Lit)\ at\ \rho', g{:}(Lit \xrightarrow{\{\rho'\}} Lit)\ at\ \rho \vdash g(5) :^{\{\rho, \rho'\}} Lit$$

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,

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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 $(Lit \xrightarrow{\varnothing} Lit)\ at\ \rho'$. Let ex_2 be the following expression:

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

We have $\rho \vdash ex_2 :^{\{\rho\}} (Lit \xrightarrow{\{\rho\}} Lit)\ at\ \rho$. 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
$ptr(H) \triangleq R_1, \ldots, R_n$	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:

$E \vdash r\ at\ \rho : R$	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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Region, Heap, and Configuration Rules:

(Region Good)
$$\frac{E \vdash v_i \; at \; \rho :^{\{\rho\}} A_i \quad \forall i \in 1..n}{E \vdash (p_i \mapsto v_i)^{\,i\in 1..n} \; at \; \rho : (p_i{:}A_i)^{\,i\in 1..n}}$$

(Heap Typing Good)
$$\frac{env(H) \vdash \diamond}{H \models \diamond}$$

(Heap Good) (where $dom(H) = dom(h)$)
$$\frac{env(H) \vdash h(\rho) \; at \; \rho : H(\rho) \quad \forall \rho \in dom(H)}{H \models h}$$

(Config Good) (where $S \subseteq dom(H)$)
$$\frac{env(H) \vdash a :^{e} A \quad e \cup fr(A) \subseteq S \quad H \models h}{H \models S \cdot (a,h) : A}$$

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 $H \models S \cdot (v \; at \; \rho, h) : A$ 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 $env(H), x{:}B \vdash b :^{e} A$.

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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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 *et al.*, 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 *et al.*, 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.

Several authors have developed type and effect analyses for concurrent languages. The earliest include systems by Nielson & Nielson (1993; 1994) and Talpin (1993). There are several effect-based analyses of the π-calculus (Yoshida, 1996; Kobayashi, 1998). Our type and effect system is considerably simpler, and therefore less expressive, than some of these related systems. Unlike previous systems, it exploits the analogy between a function type with a latent effect in a functional calculus – such as the region calculus – and a channel type with a hidden effect. In our system, we may attach a hidden effect to a channel type; the effect is unleashed by an output and is masked by an input on the channel. This new idea is essential for our modelling of the region calculus.

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 & Gonthier, 1996; Merro & 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
$x(y_1{:}T_1, \ldots, y_n{:}T_n).P$	input (no $y_i \in inp(P)$)
$\overline{x}\langle y_1, \ldots, y_n\rangle$	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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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

In a process $x(y_1{:}T_1, \ldots, y_n{:}T_n).P$, the names $y_1, \ldots, y_n$ 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:

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

Free Groups, $fg(P)$, of Process P:

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

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

Names in Input Position, $inp(P)$, in Process P:

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

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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 | 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 **0** process represents inactivity.

3.2 Dynamic semantics

We formalize the semantics of our π-calculus using standard techniques. A reduction relation, $P \to Q$, 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)
$P \equiv Q \Rightarrow x(y_1{:}T_1,\ldots,y_n{:}T_n).P \equiv x(y_1{:}T_1,\ldots,y_n{:}T_n).Q$	(Struct Input)
$P \equiv Q \Rightarrow (\nu G)P \equiv (\nu G)Q$	(Struct GRes)

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

$P \equiv Q \Rightarrow (\nu x{:}T)P \equiv (\nu x{:}T)Q$	(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: $P \to Q$

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

$erase((\nu G)P) \stackrel{\Delta}{=} erase(P)$
$erase((\nu x{:}T)P) \stackrel{\Delta}{=} (\nu x)erase(P)$
$erase(\mathbf{0}) \stackrel{\Delta}{=} \mathbf{0}$
$erase(P \mid Q) \stackrel{\Delta}{=} erase(P) \mid erase(Q)$
$erase(!P) \stackrel{\Delta}{=} !erase(P)$
$erase(x(y_1{:}T_1, \ldots, y_n{:}T_n).P) \stackrel{\Delta}{=} x(y_1, \ldots, y_n).erase(P)$
$erase(\overline{x}\langle y_1, \ldots, y_n\rangle) \stackrel{\Delta}{=} \overline{x}\langle y_1, \ldots, y_n\rangle$

Proposition 3.1 (*Erasure*)

For all typed processes P and Q, if $P \to Q$ then $erase(P) \to erase(Q)$. If $erase(P) \to R$ then there is a typed process Q such that $P \to Q$ and $R \equiv erase(Q)$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

3.3 Static semantics

The main judgment $E \vdash P : \{G_1, \ldots, G_n\}$ 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 $G_1, \ldots, G_n$. A channel type takes the form $G[T_1, \ldots, T_n]\backslash\mathbf{H}$. 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 $T_1, \ldots, T_n$. 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 $G[T_1, \ldots, T_n]$.

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 $5 : Lit[]$ 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 $g(y,k').\overline{f}\langle y,k'\rangle : \{\rho,\rho'\}$ because the reads and writes of the process are on the channels g and f whose groups are ρ and ρ'. Similarly, we have $f(x,k').\overline{k'}\langle x\rangle : \{\rho', K\}$ and $\overline{g}\langle 5,k\rangle : \{\rho\}$. 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: $f : \rho'[Lit[], K[Lit[]]]\backslash\{K\}$ and $g : \rho[Lit[], K[Lit[]]]\backslash\{K,\rho'\}$. We then have $f(x,k').\overline{k'}\langle x\rangle : \{\rho'\}$, $g(y,k').\overline{f}\langle y,k'\rangle : \{\rho\}$, and $\overline{g}\langle 5,k\rangle : \{\rho,\rho',K\}$. The hidden effects are transferred from the function bodies to the process $\overline{g}\langle 5,k\rangle$ 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:

$$(\nu f{:}\rho')(\nu g{:}\rho)(!f(x,k').\overline{k'}\langle x\rangle \mid !g(y,k').\overline{f}\langle y,k'\rangle \mid \overline{g}\langle 5,k\rangle) : \{\rho,\rho',K\}$$

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,

$$(\nu\rho')(\nu f)(\nu g)(!f(x,k').\overline{k'}\langle x\rangle \mid !g(y,k').\overline{f}\langle y,k'\rangle \mid \overline{g}\langle 5,k\rangle) : \{\rho,K\}$$

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Syntax of Types and Environments, Typing Judgments:

$\mathbf{G}, \mathbf{H} ::= \{G_1, \ldots, G_n\}$	finite set of name groups
$T ::= G[T_1, \ldots, T_n]\backslash\mathbf{H}$	type of channel in group G with hidden effect $\mathbf{H}$
$E ::= \emptyset \mid E, G \mid E, x{:}T$	environment
$E \vdash \diamond$	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 $\mathbf{H}$

Typing Rules:

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

(Type Chan) $\dfrac{E \vdash \diamond \quad \{G\} \cup \mathbf{H} \subseteq dom(E) \quad E \vdash T_1 \quad \cdots \quad E \vdash T_n}{E \vdash G[T_1, \ldots, T_n]\backslash\mathbf{H}}$ (Exp x) $\dfrac{E', x{:}T, E'' \vdash \diamond}{E', x{:}T, E'' \vdash x : T}$

(Proc Input) $\dfrac{E \vdash x : G[T_1, \ldots, T_n]\backslash\mathbf{H} \quad E, y_1{:}T_1, \ldots, y_n{:}T_n \vdash P : \mathbf{G}}{E \vdash x(y_1{:}T_1, \ldots, y_n{:}T_n).P : \{G\} \cup (\mathbf{G} - \mathbf{H})}$

(Proc Output) $\dfrac{E \vdash x : G[T_1, \ldots, T_n]\backslash\mathbf{H} \quad E \vdash y_1 : T_1 \quad \cdots \quad E \vdash y_n : T_n}{E \vdash \overline{x}\langle y_1, \ldots, y_n\rangle : \{G\} \cup \mathbf{H}}$

(Proc GRes) $\dfrac{E, G \vdash P : \mathbf{H}}{E \vdash (\nu G)P : \mathbf{H} - \{G\}}$ (Proc Res) $\dfrac{E, x{:}T \vdash P : \mathbf{H}}{E \vdash (\nu x{:}T)P : \mathbf{H}}$ (Proc Par) $\dfrac{E \vdash P : \mathbf{G} \quad E \vdash Q : \mathbf{H}}{E \vdash P \mid Q : \mathbf{G} \cup \mathbf{H}}$

(Proc Repl) $\dfrac{E \vdash P : \mathbf{H}}{E \vdash !P : \mathbf{H}}$ (Proc Zero) $\dfrac{E \vdash \diamond}{E \vdash \mathbf{0} : \emptyset}$ (Proc Subsum) $\dfrac{E \vdash P : \mathbf{G} \quad \mathbf{G} \subseteq \mathbf{H} \subseteq dom(E)}{E \vdash P : \mathbf{H}}$

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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

In the special case when the hidden effect **H** is ∅, (Proc Input) and (Proc Output) specialise to the following:

$$\frac{E \vdash x : G[T_1, \ldots, T_n]\backslash\varnothing \quad E, y_1{:}T_1, \ldots, y_n{:}T_n \vdash P : \mathbf{G}}{E \vdash x(y_1{:}T_1, \ldots, y_n{:}T_n).P : \{G\} \cup \mathbf{G}} \qquad \frac{E \vdash x : G[T_1, \ldots, T_n]\backslash\varnothing \quad E \vdash y_1 : T_1 \quad \cdots \quad E \vdash y_n : T_n}{E \vdash \overline{x}\langle y_1, \ldots, y_n\rangle : \{G\}}$$

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

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

$$\frac{E \vdash x : G[T_1, \ldots, T_n]\backslash\mathbf{H} \quad E, y_1{:}T_1, \ldots, y_n{:}T_n \vdash P : \mathbf{H}}{E \vdash x(y_1{:}T_1, \ldots, y_n{:}T_n).P : \{G\}} \qquad \frac{E \vdash x : G[T_1, \ldots, T_n]\backslash\mathbf{H} \quad E \vdash y_1 : T_1 \quad \cdots \quad E \vdash y_n : T_n}{E \vdash \overline{x}\langle y_1, \ldots, y_n\rangle : \{G\} \cup \mathbf{H}}$$

The effect of P is not attributed to the input $x(y_1{:}T_1, \ldots, y_n{:}T_n).P$ 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 $P \to Q$ 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 $\overline{x}$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Exhibition of a Barb:

$$\frac{}{x(y_1{:}T_1,\ldots,y_n{:}T_n).P \downarrow x}\ \text{(Barb Input)} \qquad \frac{}{\overline{x}\langle y_1,\ldots,y_n\rangle \downarrow \overline{x}}\ \text{(Barb Output)} \qquad \frac{P \downarrow \beta}{(\nu G)P \downarrow \beta}\ \text{(Barb GRes)}$$

$$\frac{P \downarrow \beta \quad \beta \notin \{x,\overline{x}\}}{(\nu x{:}T)P \downarrow \beta}\ \text{(Barb Res)} \qquad \frac{P \downarrow \beta}{P \mid Q \downarrow \beta}\ \text{(Barb Par)} \qquad \frac{P \equiv Q \quad Q \downarrow \beta}{P \downarrow \beta}\ \text{(Barb } \equiv\text{)}$$

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 $\beta \in \{x,\overline{x}\}$ then there is a type $G[T_1,\ldots,T_n]\backslash\mathbf{G}$ such that $E \vdash x : G[T_1,\ldots,T_n]\backslash\mathbf{G}$ 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 & Sangiorgi, 1996) of the barbed congruence of Milner & Sangiorgi (1992), an equivalence with a uniform definition for a variety of process calculi. See the textbook of Sangiorgi & Walker (2001) for further motivations and examples. 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*, $\mathscr{S}$, 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 $E \vdash P \ \mathscr{S}\ Q$ to mean $(E,P,Q) \in \mathscr{S}$.
- A relation on typed processes, $\mathscr{S}$, is *reflexive* if and only if $E \vdash P \ \mathscr{S}\ P$ whenever $E \vdash P$. It is *symmetric* if and only if $E \vdash Q \ \mathscr{S}\ P$ whenever $E \vdash P \ \mathscr{S}\ Q$. It is *transitive* if and only if $E \vdash P \ \mathscr{S}\ R$ whenever $E \vdash P \ \mathscr{S}\ Q$ and $E \vdash Q \ \mathscr{S}\ R$.
- For any relation on typed processes, $\mathscr{S}$, let $E \vdash P \equiv\mathscr{S}\equiv Q$ mean there are processes P' and Q' such that $P \equiv P'$, $E \vdash P' \ \mathscr{S}\ Q'$, 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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

- 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, $\mathscr{S}$, is a *barbed bisimulation* if and only if it is *symmetric* and $E \vdash P \mathscr{S} Q$ implies:
  (1) If $P \downarrow \overline{x}$ then $Q \Downarrow \overline{x}$.
  (2) If $P \rightarrow P'$ then there is Q' such that $Q \rightarrow^* Q'$ and $E \vdash P' \equiv \mathscr{S} \equiv Q'$.
- *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 $\mathscr{S}$ such that $E \vdash P \mathscr{S} Q$.

By definition of $E \vdash P \dot{\approx} Q$, 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 $n \geqslant 0$ and the names $x_1, \ldots, x_n$ are pairwise distinct. Let $dom(\sigma) = \{x_1, \ldots, x_n\}$ and $ran(\sigma) = \{x'_1, \ldots, x'_n\}$. If $x = x_j$ for some $j \in 1..n$, let $\sigma(x) = x'_j$. Otherwise, if $x \notin dom(\sigma)$, 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: $\varnothing\sigma \triangleq \varnothing$; $(E', G)\sigma \triangleq E'\sigma, G$; $(E', x{:}T)\sigma \triangleq E'\sigma, \sigma(x){:}T$ if $\sigma(x) \notin dom(E'\sigma)$, 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 $E, y_1{:}T_1, \ldots, y_n{:}T_n \vdash P \approx Q$ then $E \vdash x(y_1{:}T_1, \ldots, y_n{:}T_n).P \approx x(y_1{:}T_1, \ldots, y_n{:}T_n).Q$.
  - 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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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 $(\nu k')([\![a]\!]k' \mid k'(x).[\![b]\!]k)$. This process invents a fresh, intermediate continuation k'. The process $[\![a]\!]k'$ evaluates a returning a result on k'. The process $k'(x).[\![b]\!]k$ 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 $[\![S \cdot (a, h)]\!]k$ 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 $[\![\varnothing]\!]$ declares these groups and also a typing $\ell_i{:}Lit$ for each of the literals $\ell_1, \ldots, \ell_n$.

Translating of the Region Calculus to the π-Calculus:

$[\![A]\!]$	type modelling the type A
$[\![E]\!]$	environment modelling environment E
$[\![a]\!]k$	process modelling term a, answer on k
$[\![p \mapsto v]\!]$	process modelling value v at pointer p
$[\![r]\!]$	process modelling region r
$[\![S \cdot (a, h)]\!]k$	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 $I = \{i_1, \ldots, i_n\}$ is short for the composition $P_{i_1} \mid \cdots \mid P_{i_n} \mid \mathbf{0}$.

Translation Rules:

$$[\![Lit]\!] \triangleq Lit\,[]$$
$$[\![(A \xrightarrow{e} B)\ at\ \rho]\!] \triangleq \rho[[\![A]\!], K\,[[\![B]\!]]] \backslash (e \cup \{K\})$$

$$[\![\varnothing]\!] \triangleq K, Lit, \ell_1{:}Lit\,[], \ldots, \ell_n{:}Lit\,[]$$
$$[\![E, \rho]\!] \triangleq [\![E]\!], \rho$$
$$[\![E, x{:}A]\!] \triangleq [\![E]\!], x{:}[\![A]\!]$$

$$[\![x]\!]k \triangleq \overline{k}\langle x \rangle$$
$$[\![let\ x = a_A\ in\ b]\!]k \triangleq (\nu k'{:}K\,[[\![A]\!]])([\![a]\!]k' \mid k'(x{:}[\![A]\!]).[\![b]\!]k)$$
$$[\![p(q)]\!]k \triangleq \overline{p}\langle q, k \rangle$$
$$[\![letregion\ \rho\ in\ a]\!]k \triangleq (\nu\rho)[\![a]\!]k$$
$$[\![(v\ at\ \rho)_A]\!]k \triangleq (\nu p{:}[\![A]\!])([\![p \mapsto v]\!] \mid \overline{k}\langle p \rangle)$$

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

$[\![p \mapsto \lambda(x{:}A)b_B]\!] \triangleq\ !p(x{:}[\![A]\!], k{:}K\,[[\![B]\!]]).[\![b]\!]k$
$[\![(p_i \mapsto v_i)^{\,i\in 1..n}]\!] \triangleq \prod_{i\in 1..n}[\![p_i \mapsto v_i]\!]$
$[\![(\rho_i \mapsto r_i)^{\,i\in 1..n}]\!] \triangleq \prod_{i\in 1..n}[\![r_i]\!]$
$[\![S \cdot (a, h_H)]\!]k \triangleq (\nu\vec{\rho}_{\text{defunct}})(\nu\,[\![ptr(H)]\!])([\![a]\!]k \mid [\![h]\!])$ where $\{\vec{\rho}_{\text{defunct}}\} = dom(H) - S$

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 $[\![E]\!] \vdash \diamond$.
(2) If $E \vdash A$ then $[\![E]\!] \vdash [\![A]\!]$.
(3) If $E \vdash a :^e A$ and $k \notin dom([\![E]\!])$ then $[\![E]\!], k{:}K\,[[\![A]\!]] \vdash [\![a]\!]k : e \cup \{K\}$
(4) If $H \models h$ and $\rho \in dom(H)$ then $[\![env(H)]\!] \vdash [\![h(\rho)]\!] : \{\rho\}$
(5) If $H \models S \cdot (a, h) : A$ and $k \notin [\![env(H)]\!]$ then

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

and also: $[\![\varnothing]\!], S, k{:}K\,[[\![A]\!]] \vdash [\![S \cdot (a, h)]\!]k : S \cup \{K\}$

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 $k \notin dom_2(H + H') \cup L$, $[\![\varnothing]\!], S, k{:}K\,[[\![A]\!]] \vdash [\![S \cdot (a, h)]\!]k \approx [\![S \cdot (p', h')]\!]k$.

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):

$$[\![\{\rho\} \cdot (ex_1, h)]\!]k \approx (\nu\rho')(\nu f{:}\rho')(\nu g{:}\rho)([\![f \mapsto \lambda(x)x]\!] \mid [\![g \mapsto \lambda(y)f(y)]\!] \mid \overline{k}\langle 5\rangle)$$

$$[\![\{\rho\} \cdot (ex_2, h)]\!]k \approx (\nu\rho')(\nu f{:}\rho')(\nu g{:}\rho)(\nu j{:}\rho)\ ([\![f \mapsto \lambda(x)x]\!] \mid [\![g \mapsto \lambda(f)5]\!] \mid [\![j \mapsto \lambda(z)g(f)]\!] \mid \overline{k}\langle j\rangle)$$

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. Although various untyped equations to eliminate unused resources have been proposed, as far as we know none is applicable in our situation.

Suppose there are processes P and R such that 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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

We state the theorem in terms of the notation $(\nu E)P$ defined by the equations: $(\nu \varnothing)P \triangleq P$, $(\nu E, x{:}T)P \triangleq (\nu E)(\nu x{:}T)P$, 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:

$$(\nu\rho')(\nu f{:}\rho')(\nu g{:}\rho)([\![f \mapsto \lambda(x)x]\!] \mid [\![g \mapsto \lambda(y)f(y)]\!] \mid \overline{k}\langle 5\rangle) \approx (\nu\rho')(\nu f{:}\rho')(\nu g{:}\rho)([\![g \mapsto \lambda(y)f(y)]\!] \mid \overline{k}\langle 5\rangle)$$

$$(\nu\rho')(\nu f{:}\rho')(\nu g{:}\rho)(\nu j{:}\rho)([\![f \mapsto \lambda(x)x]\!] \mid [\![g \mapsto \lambda(f)5]\!] \mid [\![j \mapsto \lambda(z)g(f)]\!] \mid \overline{k}\langle j\rangle) \approx (\nu\rho')(\nu f{:}\rho')(\nu g{:}\rho)(\nu j{:}\rho)([\![g \mapsto \lambda(f)5]\!] \mid [\![j \mapsto \lambda(z)g(f)]\!] \mid \overline{k}\langle j\rangle)$$

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 $[\![g \mapsto \lambda(y)f(y)]\!] \mid \overline{k}\langle 5\rangle$ is simply $\{\rho, K\}$. This effect does not contain ρ' and so the theorem justifies deletion of the process $[\![f \mapsto \lambda(x)x]\!]$, whose effect is $\{\rho'\}$. In an effect system for the π-calculus without hidden effects, the effect of $[\![g \mapsto \lambda(y)f(y)]\!] \mid \overline{k}\langle 5\rangle$ 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 $k \notin dom_2(H) \cup L$. Let $\vec{\rho}_{\text{defunct}}$ be the sequence of groups in $dom(H) - S$. Then:

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

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 & Kennedy (1999) for their ML intermediate language (without regions) appear to

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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 $b(a) \triangleq \mathit{let}\ f = b\ \mathit{in}\ \mathit{let}\ x = a\ \mathit{in}\ f(x)$ where $f \notin \{x\} \cup \mathit{fn}(a)$. 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:

$$x\{z \leftarrow c\} \triangleq \begin{cases} c & \text{if } x = z \\ x & \text{otherwise} \end{cases}$$

$$x(y)\{z \leftarrow c\} \triangleq x\{z \leftarrow c\}(y\{z \leftarrow c\})$$

$$(\mathit{let}\ x = a\ \mathit{in}\ b)\{z \leftarrow c\} \triangleq \mathit{let}\ x = a\{z \leftarrow c\}\ \mathit{in}\ (b\{z \leftarrow c\}) \qquad \text{for } x \notin \{z\} \cup \mathit{fn}(c)$$

$$(\lambda(x{:}A)b)\{z \leftarrow c\} \triangleq \lambda(x{:}A)(b\{z \leftarrow c\}) \qquad \text{for } x \notin \{z\} \cup \mathit{fn}(c)$$

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, $\lambda(x)b\ \mathit{at}\ \rho$. 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 $(\lambda(x{:}A)b\ \mathit{at}\ \rho)(a)$ 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 $(\lambda(x{:}A)b\ \mathit{at}\ \rho)(a)$.

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

(Eq Refl)
$$\frac{E \vdash a :^{e} A}{E \vdash a \leftrightarrow a : A}$$

(Eq Symm)
$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)
$$\frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

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

(Eq Fun β) (where a is a name or an allocation)
$$\frac{E \vdash a :^{e_1} A \quad E, x{:}A \vdash b :^{e_2} B \quad E \vdash b\{x \leftarrow a\} :^{e_3} B \quad \rho \in \mathit{dom}(E)}{E \vdash (\lambda(x{:}A)b\ \mathit{at}\ \rho)(a) \leftrightarrow b\{x \leftarrow a\} : B}$$

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Equational Theory: *let*

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

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

(Eq Let β) (where a is a name or an allocation)
$$\frac{E \vdash a :^{e_1} A \quad E, x{:}A :^{e_2} b : B \quad E \vdash b\{x{\leftarrow}a\} :^{e_3} B}{E \vdash let\ x = a\ in\ b \leftrightarrow b\{x{\leftarrow}a\} : B}$$

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

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

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

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

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 $\rho \notin dom(E)$, rather than the weaker condition $\rho \notin e \cup fr(A)$, 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 lemma justifies our intention that if a_1 and a_2 are related by the equational theory then in fact they have the same type, although they need not have the same effect.

Lemma 5.1
If $E \vdash a_1 \leftrightarrow a_2 : A$ then there are e_1, e_2 such that for each $i \in 1..2$, $e_i \subseteq dom(E)$ and $E \vdash a_i :^{e_i} A$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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 $k \notin dom(E) \cup L$. Then $[\![E]\!], k{:}K[\![A]\!] \vdash [\![a]\!]k \approx [\![b]\!]k$.

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.

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).

(Eq Appl $x(y)$)

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

(Eq Appl) (where $\rho \notin fr((A \xrightarrow{e_2} B)\ at\ \rho')$)

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

Other examples of derivable rules are:

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

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

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

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

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

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

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

The following are special cases of (Eq Drop):

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

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

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.

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

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. As we discussed at the beginning of section 2, this extended calculus includes all the features of Tofte and Talpin's original calculus, except type polymorphism. We include lists as a simple example of a boxed data structure. We conjecture that our treatment of lists would easily extend to general algebraic types, that is, recursive sums-of-products, with no further extensions to the typed π-calculus of this section. Likewise, a treatment of mutable storage would require no extensions. On the other hand, type or effect polymorphism would require an extended π-calculus; we leave this study as future work. The results of this section show that our π-calculus model of the region calculus is robust with respect to extensions that do not directly manipulate regions.

We describe our 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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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
$\quad x$	$\quad$ variable or pointer or literal
$\quad v \ at \ \rho$	$\quad$ allocation of v at ρ
$\quad x[\rho_1, \ldots, \rho_n](y)$	$\quad$ application
$\quad let \ x = a \ in \ b$	$\quad$ sequencing
$\quad letregion \ \rho \ in \ b$	$\quad$ region allocation and de-allocation
$\quad case \ x \ of \ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2$	$\quad$ list case
$u, v ::=$	boxed value
$\quad \mu(f{:}A)\lambda[\rho_1, \ldots, \rho_n](x)b$	$\quad$ recursive function
$\quad nil$	$\quad$ empty list
$\quad x_1 :: x_2$	$\quad$ 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 $\mu(f{:}A)\lambda[\rho_1, \ldots, \rho_n](x)b$, the names f and x and the region variables $\rho_1, \ldots, \rho_n$ 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 $\rho_1, \ldots, \rho_n$ 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 & Talpin, 1997). Other kinds 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, $x[\rho_1, \ldots, \rho_n](y)$, applies the function pointed to by x to the region parameters $\rho_1, \ldots, \rho_n$, and the value parameter y. The other new expression, $case \ x \ of \ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2$, 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
$S ::= \{\rho_1, \ldots, \rho_n\}$	stack of live regions

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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)

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

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

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

(Eval Let)

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

(Eval Letregion)

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

(Eval Case 1)

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

(Eval Case 2)

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

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 $(\mu(f{:}F)\lambda[\rho_1, \ldots, \rho_n](x)b)$ *at* ρ, will have a type $F = (\forall[\rho_1, \ldots, \rho_n]A \xrightarrow{e} B)$ *at* ρ, where A is the type of the function argument x, and the regions $\rho_1, \cdots, \rho_n$ 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 is stored in the same region than the list itself.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Effects, Types, and Environments:

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

In the type $(\forall[\rho_1, \ldots, \rho_n]A \xrightarrow{e} B) \ at \ \rho$, the regions $\rho_1, \ldots, \rho_n$ 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 $fr((\forall[\rho_1, \ldots, \rho_n]A \xrightarrow{e} B) \ at \ \rho) = (fr(A) \cup fr(B) \cup e) - \{\rho_1, \ldots, \rho_n\} \cup \{\rho\}$, and $fr([A] \ at \ \rho) = fr(A) \cup \{\rho\}$. 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 $e = \{\rho_1, \cdots, \rho_n\}$, the effect $e\sigma$ is the set of regions $\{\sigma(\rho_1), \cdots, \sigma(\rho_n)\}$.

Typing Rules:

(Env $\emptyset$) $\dfrac{}{\emptyset \vdash \diamond}$ $\qquad$ (Env x) $\dfrac{E \vdash A \quad x \notin dom(E) \cup L}{E, x{:}A \vdash \diamond}$ $\qquad$ (Env ρ) $\dfrac{E \vdash \diamond \quad \rho \notin dom(E)}{E, \rho \vdash \diamond}$ $\qquad$ (Type Lit) $\dfrac{E \vdash \diamond}{E \vdash Lit}$

(Type $\rightarrow$) (where $E' = E, \rho_1, \ldots, \rho_n$) $\dfrac{E' \vdash A \quad e \subseteq dom(E') \quad E' \vdash B \quad \rho \in dom(E)}{E \vdash (\forall[\rho_1, \ldots, \rho_n]A \xrightarrow{e} B) \ at \ \rho}$

(Type List) $\dfrac{E \vdash A \quad \rho \in dom(E)}{E \vdash [A] \ at \ \rho}$

(Exp x) $\dfrac{E, x{:}A, E' \vdash \diamond}{E, x{:}A, E' \vdash x :^{\emptyset} A}$ $\qquad$ (Exp ℓ) $\dfrac{E \vdash \diamond \quad \ell \in L}{E \vdash \ell :^{\emptyset} Lit}$

(Exp Appl) (where $\sigma = \{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$ and $\{\rho'_1, \ldots, \rho'_n\} \subseteq dom(E)$)
$$\frac{E \vdash x :^{\emptyset} (\forall[\rho_1, \ldots, \rho_n]A \xrightarrow{e} B) \ at \ \rho \quad E \vdash y :^{\emptyset} A\sigma}{E \vdash x[\rho'_1, \ldots, \rho'_n](y) :^{\{\rho\} \cup (e\sigma)} B\sigma}$$

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

(Exp Let)
$$\frac{E \vdash a :^{e} A \quad E, x{:}A \vdash b :^{e'} B}{E \vdash \mathit{let}\ x = a\ \mathit{in}\ b :^{e \cup e'} B}$$

(Exp Letregion)
$$\frac{E, \rho \vdash a :^{e} A \quad E \vdash A}{E \vdash \mathit{letregion}\ \rho\ \mathit{in}\ a :^{e - \{\rho\}} A}$$

(Exp Case)
$$\frac{E \vdash x :^{\emptyset} [A]\ \mathit{at}\ \rho \quad E \vdash b_1 :^{e_1} B \quad E, y_1{:}A, y_2{:}[A]\ \mathit{at}\ \rho \vdash b_2 :^{e_2} B}{E \vdash \mathit{case}\ x\ \mathit{of}\ \mathit{nil} \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2 :^{\{\rho\} \cup e_1 \cup e_2} B}$$

(Exp Fun) (where $F = (\forall[\rho_1, \ldots, \rho_n]A \xrightarrow{e} B)\ \mathit{at}\ \rho$)
$$\frac{E, f{:}F, \rho_1, \ldots, \rho_n, x{:}A \vdash b :^{e'} B \quad e' \subseteq e \subseteq \mathit{dom}(E, \rho_1, \ldots, \rho_n)}{E \vdash (\mu(f{:}F)\lambda[\rho_1, \ldots, \rho_n](x)b)\ \mathit{at}\ \rho :^{\{\rho\}} F}$$

(Exp Nil)
$$\frac{E \vdash [A]\ \mathit{at}\ \rho}{E \vdash \mathit{nil}\ \mathit{at}\ \rho :^{\{\rho\}} [A]\ \mathit{at}\ \rho}$$

(Exp Cons)
$$\frac{E \vdash x_1 :^{\emptyset} A \quad E \vdash x_2 :^{\emptyset} [A]\ \mathit{at}\ \rho}{E \vdash (x_1 :: x_2)\ \mathit{at}\ \rho :^{\{\rho\}} [A]\ \mathit{at}\ \rho}$$

The definitions of region and heap typings, R and H, respectively, and of the judgments $E \vdash r\ \mathit{at}\ \rho : R$, $H \models \diamond$, $H \models h$, 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 $G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{H}$, where G is the group of the channel, $G_1, \ldots, G_m$ are group parameters, $T_1, \ldots, T_n$ are the types of the name parameters, and $\mathbf{H}$ is the hidden effect. The types $T_1, \ldots, T_n$ and the effect $\mathbf{H}$ may depend on the group parameters $G_1, \ldots, G_m$. An output process takes the form $\overline{x}\langle G_1, \ldots, G_m, y_1, \ldots, y_n\rangle$, where $G_1, \ldots, G_m$ are the group parameters, and $y_1, \ldots, y_n$ are the name parameters. An input process takes the form $x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P$ where $G_1, \ldots, G_m$ receive the group parameters, and $y_1, \ldots, y_n$ 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 & 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 $\mu(X)T$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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

In the type $G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{H}$, the groups $G_1, \ldots, G_m$ are bound with scope $T_1, \ldots, T_n$ and $\mathbf{H}$. In the type $\mu(X)T$, the type variable X is bound with scope T. In a process $x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P$, the groups $G_1, \ldots, G_m$ and the names $y_1, \ldots, y_n$ 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:

$$fg(X) \triangleq \emptyset$$
$$fg(G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{H}) \triangleq \{G\} \cup ((fg(T_1) \cup \cdots \cup fg(T_n) \cup \mathbf{H}) - \{G_1, \ldots, G_m\})$$
$$fg(\mu(X)T) \triangleq fg(T)$$

Free Groups, $fg(P)$, of Process P:

$$fg(x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P) \triangleq (fg(T_1) \cup \cdots \cup fg(T_n) \cup fg(P)) - \{G_1, \ldots, G_m\}$$
$$fg(\overline{x}\langle G_1, \ldots, G_m, y_1, \ldots, y_n\rangle) \triangleq \{G_1, \ldots, G_m\}$$
$$fg((\nu G)P) \triangleq fg(P) - \{G\}$$
$$fg((\nu x{:}T)P) \triangleq fg(T) \cup fg(P)$$

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

$fg(P \mid Q) \triangleq fg(P) \cup fg(Q)$
$fg(!P) \triangleq fg(P)$
$fg(\mathbf{0}) \triangleq \varnothing$

We identify types and processes up to consistent renaming of bound groups, names, and type variables. We write $P\{x \leftarrow y\}$ for the outcome of substituting y for each free occurrence of the name x in the process P. We write $P\{G \leftarrow H\}$ and $T\{G \leftarrow H\}$ for the outcomes of substituting H for each free occurrence of the group G in the process P and the type T, respectively. We write $P\{X \leftarrow T\}$ and $T'\{X \leftarrow T\}$ for the outcomes of substituting T for each free occurrence of the type variable X in the process P and the type T', respectively.

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

$$P \equiv Q \Rightarrow x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P \equiv x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).Q \qquad \text{(Struct Input)}$$

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

Reduction:

$\overline{x}\langle G'_1, \ldots, G'_m, y'_1, \ldots, y'_n\rangle \mid x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P \rightarrow P\{G_1 \leftarrow G'_1\} \cdots \{G_m \leftarrow G'_m\}\{y_1 \leftarrow y'_1\} \cdots \{y_n \leftarrow y'_n\}$	(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 $\equiv$)

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 $dom(\varnothing) = \varnothing$, $dom(E, \rho) = dom(E) \cup \{\rho\}$, $dom(E, x{:}A) = dom(E) \cup \{x\}$ and $dom(E, X) = dom(E) \cup \{X\}$.

Environments:

E ::=	environment
$\varnothing$	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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Typing Rules:

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

$$\text{(Env } G\text{)} \quad \frac{E \vdash \diamond \quad G \notin dom(E)}{E, G \vdash \diamond} \qquad \text{(Env } X\text{)} \quad \frac{E \vdash \diamond \quad X \notin dom(E)}{E, X \vdash \diamond}$$

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

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

$$\text{(Exp Unfold)} \quad \frac{E \vdash x : \mu(X)T}{E \vdash x : T\{X \leftarrow \mu(X)T\}} \qquad \text{(Exp Fold)} \quad \frac{E \vdash x : T\{X \leftarrow \mu(X)T\}}{E \vdash x : \mu(X)T}$$

$$\text{(Proc Input) (where } (\mathbf{G} - \mathbf{H}) \cap \{G_1, \ldots, G_m\} = \varnothing\text{)} \quad \frac{\begin{array}{c} E \vdash x : G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{H} \\ E, G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n \vdash P : \mathbf{G} \end{array}}{E \vdash x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P : \{G\} \cup (\mathbf{G} - \mathbf{H})}$$

$$\text{(Proc Output) (where } \sigma = \{G_1 \leftarrow G'_1\} \cdots \{G_m \leftarrow G'_m\}\text{)} \quad \frac{\begin{array}{l} E \vdash x : G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{H} \\ \{G'_1, \ldots, G'_m\} \subseteq dom(E) \quad E \vdash y'_i : T_i\sigma \quad \forall i \in 1..n \end{array}}{E \vdash \overline{x}\langle G'_1, \ldots, G'_m, y'_1, \ldots, y'_n\rangle : \{G\} \cup \mathbf{H}\sigma}$$

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

$$\text{(Proc Repl)} \quad \frac{E \vdash P : \mathbf{H}}{E \vdash !P : \mathbf{H}} \qquad \text{(Proc Zero)} \quad \frac{E \vdash \diamond}{E \vdash \mathbf{0} : \varnothing} \qquad \text{(Proc Subsum)} \quad \frac{E \vdash P : \mathbf{G} \quad \mathbf{G} \subseteq \mathbf{H} \subseteq dom(E)}{E \vdash P : \mathbf{H}}$$

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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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 $\beta \in \{x, \overline{x}\}$, is defined much as before.

Exhibition of a Barb:

(Barb Input) $\quad x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P \downarrow x$

(Barb Output) $\quad \overline{x}\langle G_1, \ldots, G_m, y_1, \ldots, y_n\rangle \downarrow \overline{x}$

(Barb GRes) $\quad \dfrac{P \downarrow \beta}{(\nu G)P \downarrow \beta}$

(Barb Res) $\quad \dfrac{P \downarrow \beta \quad \beta \notin \{x, \overline{x}\}}{(\nu x{:}T)P \downarrow \beta}$

(Barb Par) $\quad \dfrac{P \downarrow \beta}{P \mid Q \downarrow \beta}$

(Barb $\equiv$) $\quad \dfrac{P \equiv Q \quad Q \downarrow \beta}{P \downarrow \beta}$

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), $\mathscr{S}$, 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 $\rho(\rho_1, \ldots, \rho_m)[T_1, \ldots, T_n]\backslash\mathbf{H}$. In the common case when $m = 0$, that is, the value is monomorphic, and has no hidden effect, we abbreviate the type to $\rho[T_1, \ldots, T_n]$.

Translating of the Region Calculus to the π-Calculus:

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

Translation Rules:

$[\![Lit]\!] \triangleq Lit\,[]$

$[\![(\forall[\rho_1, \ldots, \rho_n]A \xrightarrow{e} B)\ at\ \rho]\!] \triangleq \rho(\rho_1, \ldots, \rho_n)[[\![A]\!], K\,[[\![B]\!]]]\backslash(e \cup \{K\})$

$[\![[A]\ at\ \rho]\!] \triangleq \mu(X)\rho[\rho[], \rho[[\![A]\!], X]]$

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

$[\![\varnothing]\!] \triangleq K, Lit, \ell_1{:}Lit\,[], \ldots, \ell_n{:}Lit\,[]$
$[\![E, \rho]\!] \triangleq [\![E]\!], \rho$
$[\![E, x{:}A]\!] \triangleq [\![E]\!], x{:}[\![A]\!]$

$[\![x]\!]k \triangleq \overline{k}\langle x\rangle$
$[\![let\ \ x = a_A\ in\ b]\!]k \triangleq (\nu k'{:}K\,[[\![A]\!]])([\![a]\!]k' \mid k'(x{:}[\![A]\!]).[\![b]\!]k)$
$[\![p[\rho_1, \ldots, \rho_n](q)]\!]k \triangleq \overline{p}\langle \rho_1, \ldots, \rho_n, q, k\rangle$
$[\![(v\ at\ \rho)_A]\!]k \triangleq (\nu p{:}[\![A]\!])([\![p \mapsto v]\!] \mid \overline{k}\langle p\rangle)$
$[\![letregion\ \rho\ in\ a]\!]k \triangleq (\nu \rho)[\![a]\!]k$
$[\![case\ p_{[A]at\,\rho}\ of\ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2]\!]k \triangleq (\nu z_1{:}\rho[])(\nu z_2{:}\rho[[\![A]\!], [\![[A]\ at\ \rho]\!]])$
$\quad (\overline{p}\langle z_1, z_2\rangle \mid z_1().[\![b_1]\!]k \mid z_2(y_1{:}[\![A]\!], y_2{:}[\![[A]\ at\ \rho]\!]).[\![b_2]\!]k)$

$[\![p \mapsto \mu(f{:}F)\lambda[\rho_1, \ldots, \rho_n](x)b]\!] \triangleq\ !p(\rho_1, \ldots, \rho_n, x{:}[\![A]\!], k{:}K\,[[\![B]\!]]).[\![b\{f \leftarrow p\}]\!]k$
$\qquad\qquad$ where $F = (\forall[\rho_1, \ldots, \rho_n]A \xrightarrow{e} B)\ at\ \rho$
$[\![p \mapsto nil_{[A]at\,\rho}]\!] \triangleq\ !p(z_1{:}\rho[], z_2{:}\rho[[\![A]\!], [\![[A]\ at\ \rho]\!]]).\overline{z_1}\langle\rangle$
$[\![p \mapsto (x_1 :: x_2)_{[A]at\,\rho}]\!] \triangleq\ !p(z_1{:}\rho[], z_2{:}\rho[[\![A]\!], [\![[A]\ at\ \rho]\!]]).\overline{z_2}\langle x_1, x_2\rangle$

$[\![(p_i \mapsto v_i)^{\,i\in 1..n}]\!] \triangleq \prod_{i\in 1..n}[\![p_i \mapsto v_i]\!]$
$[\![(\rho_i \mapsto r_i)^{\,i\in 1..n}]\!] \triangleq \prod_{i\in 1..n}[\![r_i]\!]$
$[\![S \cdot (a, h_H)]\!]k \triangleq (\nu \vec{\rho}_{\text{defunct}})(\nu [\![ptr(H)]\!])([\![a]\!]k \mid [\![h]\!])$ where $\{\vec{\rho}_{\text{defunct}}\} = dom(H) - S$

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 & 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 types generated by *letregion*, just as the standard new-name construct of the π-calculus accounts for dynamic generation of values.

Banerjee, Heintze & Riecke (1999) give an alternative proof of the soundness of

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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.

Other alternative proofs of soundness are contained in some papers published after this work first appeared (Helsen & Thiemann, 2000; Calcagno, 2001; Talpin, 2001). These papers give concrete, purely syntactic proofs of soundness for various region calculi. Still, these specific proofs do not expose the fact that the soundness of region de-allocation follows from a more abstract principle such as our Theorem 4.3.

A line of future work is to consider the semantics of other region calculi (Aiken *et al.*, 1995; Crary *et al.*, 1999; Hughes & Pareto, 1999) in terms of the π-calculus. Finally, various researchers (Moggi & Palumbo, 1999; Semmelroth & Sabry, 1999) have noted a connection between the monadic encapsulation of state in Haskell (Launchbury & 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 paper. Luca Cardelli, Tony Hoare, and Andy Moran commented on a draft.

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 & 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 & Gonthier, 1996; Pierce & Turner, 2000). The additional algebraic laws obtained in the local variant of π, such as, for example, the replication laws listed subsequently in Proposition A.6, 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
$x(y_1, \ldots, y_n).P$	input (no $y_i \in inp(P)$)
$\overline{x}\langle y_1, \ldots, y_n\rangle$	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, $x(y_1, \ldots, y_n).P$, namely that no parameter y_i is in $inp(P)$, where $inp(P)$ is the set of names x such that an input $x(z_1, \ldots, z_m).P'$ 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)

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow x(y_1,\ldots,y_n).P \equiv x(y_1,\ldots,y_n).Q$	(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:

$\overline{x}\langle y_1,\ldots,y_n\rangle \mid x(z_1,\ldots,z_n).P \rightarrow P\{z_1 \leftarrow y_1\}\cdots\{z_n \leftarrow y_n\}$	(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 appears to be difficult to prove directly Lemma B.29, a property essential in the proof of the garbage collection principle given in Appendix B.5, without proving first an equivalent result, Lemma B.28, for the 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 & 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 names $x_1,\ldots,x_n$ such that $n \geqslant 0$ and $x_1, \ldots, x_n$ 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 names $z_1, \ldots, z_m$, and $y_1, \ldots, y_n$, respectively, such that $m, n \geqslant 0$, and $\{\vec{z}\} \subseteq \{\vec{y}\}$, and $z_1, \ldots, z_m$ 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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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

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

$$(\nu y)(\vec{x})P \stackrel{\Delta}{=} (\vec{x})(\nu y)P \quad \text{and} \quad R \mid (\vec{x})P \stackrel{\Delta}{=} (\vec{x})(R \mid P)$$

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

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

$$(\nu x)(\nu\vec{z})\langle\vec{y}\rangle Q \stackrel{\Delta}{=} \begin{cases} (\nu x,\vec{z})\langle\vec{y}\rangle Q & \text{if } x \in \{\vec{y}\} \\ (\nu\vec{z})\langle\vec{y}\rangle(\nu x)Q & \text{otherwise} \end{cases}$$

$$R \mid (\nu\vec{z})\langle\vec{y}\rangle Q \stackrel{\Delta}{=} (\nu\vec{z})\langle\vec{y}\rangle(R \mid Q)$$

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

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 $\vec{x} = x_1,\ldots,x_n$ and C is the concretion $(\nu\vec{z})\langle\vec{y}\rangle Q$ where $\vec{y} = y_1,\ldots,y_n$ and $\{\vec{z}\} \cap fn(P) = \emptyset$, we define the *interactions* $F@C$ and $C@F$ to be the processes given by:

$$F@C \stackrel{\Delta}{=} (\nu\vec{z})(P\{x_1 \leftarrow y_1\}\cdots\{x_n \leftarrow y_n\} \mid Q)$$

$$C@F \stackrel{\Delta}{=} (\nu\vec{z})(Q \mid P\{x_1 \leftarrow y_1\}\cdots\{x_n \leftarrow y_n\})$$

An *action* is either a barb or the distinguished *silent action* τ. The *labelled transition system* is written $P \stackrel{\alpha}{\longrightarrow} A$, 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)
$$\frac{}{x(y_1,\ldots,y_n).P \stackrel{x}{\longrightarrow} (y_1,\ldots,y_n)P}$$

(Trans Out)
$$\frac{}{\overline{x}\langle y_1,\ldots,y_n\rangle \stackrel{\overline{x}}{\longrightarrow} (\nu)\langle y_1,\ldots,y_n\rangle\mathbf{0}}$$

(Trans Inter 1) (with $|F| = |C|$)
$$\frac{P \stackrel{x}{\longrightarrow} F \quad Q \stackrel{\overline{x}}{\longrightarrow} C}{P \mid Q \stackrel{\tau}{\longrightarrow} F@C}$$

(Trans Inter 2) (with $|F| = |C|$)
$$\frac{P \stackrel{\overline{x}}{\longrightarrow} C \quad Q \stackrel{x}{\longrightarrow} F}{P \mid Q \stackrel{\tau}{\longrightarrow} C@F}$$

(Trans Par 1)
$$\frac{P \stackrel{\alpha}{\longrightarrow} A}{P \mid Q \stackrel{\alpha}{\longrightarrow} A \mid Q}$$

(Trans Par 2)
$$\frac{Q \stackrel{\alpha}{\longrightarrow} A}{P \mid Q \stackrel{\alpha}{\longrightarrow} P \mid A}$$

(Trans Res)
$$\frac{P \stackrel{\alpha}{\longrightarrow} A \quad \alpha \notin \{x,\overline{x}\}}{(\nu x)P \stackrel{\alpha}{\longrightarrow} (\nu x)A}$$

(Trans Repl)
$$\frac{P \mid !P \stackrel{\alpha}{\longrightarrow} A}{!P \stackrel{\alpha}{\longrightarrow} A}$$

The following is a basic result that states that modulo structural congruence, the reduction relation exactly represents the silent action of the transition semantics.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

A proof of this property can be obtained by adapting the detailed proof of an equivalent result for the π-calculus (Abadi & Gordon, 1999).

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

A.4 Barbed congruence

The notion of equivalence between untyped terms that we consider in this paper is barbed congruence (Milner & 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 $\overline{x}$. We write $P \Downarrow \beta$ if there exists P' such that $P \to^* P'$ and $P' \downarrow \beta$, where the relation $\downarrow$ is defined in the following table.

Exhibition of a Barb:

(Barb Input) $\quad x(y_1,\ldots,y_n).P \downarrow x$

(Barb Output) $\quad \overline{x}\langle y_1,\ldots,y_n\rangle \downarrow \overline{x}$

(Barb Res) $\quad \dfrac{P \downarrow \beta \quad \beta \notin \{x,\overline{x}\}}{(\nu x)P \downarrow \beta}$

(Barb Par) $\quad \dfrac{P \downarrow \beta}{P \mid Q \downarrow \beta}$

(Barb $\equiv$) $\quad \dfrac{P \equiv Q \quad Q \downarrow \beta}{P \downarrow \beta}$

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.2
$P \downarrow \beta$ 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 $P \downarrow \overline{x}$ then $Q \Downarrow \overline{x}$.
  (2) If $P \to P'$ then there is Q' such that $Q \to^* Q'$ 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 $n \geqslant 0$ and the names $x_1, \ldots, x_n$ are pairwise distinct. Let $dom(\sigma) = \{x_1,\ldots,x_n\}$ and $ran(\sigma) = \{x'_1,\ldots,x'_n\}$. If $x = x_j$ for some $j \in 1..n$, let $\sigma(x) = x'_j$. Otherwise, if $x \notin dom(\sigma)$, 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 $P \mathcal{S} Q$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

- *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.3

(1) Barbed congruence is reflexive, transitive, and symmetric.
(2) Barbed congruence satisfies the congruence properties.
   - If $P \approx Q$ then $x(y_1, \ldots, y_n).P \approx x(y_1, \ldots, y_n).Q$.
   - 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 $x \notin fn(P)$ then $(\nu x)P \approx P$.

Next, we state 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.4 (*Non-Interference*)
If $k \notin \{z\} \cup fn(P)$, then $(\nu k)(\bar{k}\langle z\rangle \mid k(x).P) \approx P\{x \leftarrow z\}$.

We also make use of the following algebraic laws.

Lemma A.5

(1) If $k \notin \{x, y_1, \ldots, y_n\}$ then $x(y_1, \ldots, y_n).(\nu k)P \approx (\nu k)x(y_1, \ldots, y_n).P$.
(2) If $k \notin fn(Q)$ then $(\nu k)(k(x_1, \ldots, x_n).P \mid Q) \approx Q$ and $(\nu k)(!k(x_1, \ldots, x_n).P \mid Q) \approx Q$.
(3) If $x \notin fn(k'(y).Q)$ then $(\nu k')(k(x).(P \mid k'(y).Q)) \approx (\nu k')(k(x).P \mid k'(y).Q)$.

Assume p does not appear free in input position in P, Q, that is, $p \notin inp(P) \cup inp(Q)$, let the operator $def\ \ p(x,k) = P\ \ in\ Q$ 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, $erase([\![\lambda(x{:}A)b\ \ at\ \ \rho]\!]k)$ can be rewritten $def\ \ p(x,k) = [\![b]\!]k\ \ in\ \bar{k}\langle p\rangle$.

Replicated Resources:

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

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:

$$def\ \ p(y_1, \ldots, y_n) = P\ \ in\ (Q \mid R) \approx (def\ \ p(y_1, \ldots, y_n) = P\ \ in\ Q) \mid (def\ \ p(y_1, \ldots, y_n) = P\ \ in\ R)$$

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

We state below Proposition A.6, which lists a more complete set of replication laws. Milner (1999) proves an equivalent property for the full π-calculus, where the equivalence used is strong ground congruence (Milner *et al.*, 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. Merro & Sangiorgi (1998) prove the same equation, without the first side condition, for barbed congruence in the local π-calculus. Dal Zilio (1999) proves similar laws for a local variant of the blue calculus.

Proposition A.6 (*Replication Laws*)

(1) If $p \notin fn(P)$ then $def\ \ p(\vec{y}) = R\ in\ P \approx P$

(2) $def\ \ p(\vec{y}) = R\ in\ (P \mid Q) \approx (def\ \ p(\vec{y}) = R\ in\ P) \mid (def\ \ p(\vec{y}) = R\ in\ Q)$

(3) If $p \neq q$ and $q \notin fn(R)$ then:

$$def\ \ p(\vec{y}) = R\ in\ (def\ \ q(\vec{z}) = S\ in\ P) \approx def\ \ q(\vec{z}) = (def\ \ p(\vec{y}) = R\ in\ S)\ in\ (def\ \ p(\vec{y}) = R\ in\ P)$$

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

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

(5) $def\ \ p(y_1, \ldots, y_n) = P\ in\ (\overline{p}\langle z_1, \ldots, z_n\rangle \mid Q) \approx def\ \ p(y_1, \ldots, y_n) = P\ in\ (P\{y_1 \leftarrow z_1\} \cdots \{y_n \leftarrow z_n\} \mid Q)$

B Proofs

In this appendix, we prove all the propositions stated without proof in the main body of the paper. Proofs for auxiliary results can be found in a technical report (Dal Zilio & Gordon, 2000b). 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.5, the subject reduction property for the extended region calculus, and Proposition B.6, 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.17, the subject reduction property for our extended π-calculus, and Proposition B.18, 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.22, 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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

(4) In Appendix B.4, we prove Proposition B.27, 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.30, 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.32, 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.35, which asserts that the encoding preserves the dynamic semantics.
Appendix B.6.4 proves Theorem B.36, which asserts that defunct regions make no difference to the behaviour of a program.
The three theorems proved in this appendix correspond to Theorems 4.1, 4.2, and 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.5, that in the extended region calculus reduction preserves types. We also prove Proposition B.6, that well-typed values are allocated in live regions and that well-typed function applications invoke closures stored in live regions. Combining these two properties implies that a well-typed expression cannot yield a runtime errors.

The proof of these properties uses a standard series of simple intermediate results, like weakening properties, Lemmas B.1 and B.2, or substitution lemmas, Lemmas B.3 to B.4.

In the type and effect system introduced in section 2.3 and section 6.1, each judgment has the form $E \vdash \mathscr{J}$, where E is a typing judgment and $\mathscr{J}$ 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 the section, we use the symbol $\mathscr{J}$ to denote such an assertion.

Lemma B.1
If $E \vdash \mathscr{J}$ and $E, E' \vdash \diamond$ then $E, E' \vdash \mathscr{J}$.

Lemma B.2
If $env(H) \vdash \mathscr{J}$ and $H \asymp H'$ and $H' \models \diamond$ then $env(H + H') \vdash \mathscr{J}$.

Lemma B.3
If $E, x{:}A, E' \vdash \mathscr{J}$ and $E \vdash p :^{\varnothing} A$ then $E, E' \vdash \mathscr{J}\{x{\leftarrow}p\}$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Lemma B.4
If $E, \rho, E' \vdash \mathscr{J}$ and ρ' is a region defined in $dom(E)$ then $E, E'\{\rho \leftarrow \rho'\} \vdash \mathscr{J}\{\rho \leftarrow \rho'\}$.

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.5
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 $S \cdot (v \ at \ \rho, h) \Downarrow (p, h + (\rho \mapsto (h(\rho) + (p \mapsto v))))$ with $\rho \in S$ and $p \notin dom_2(h)$.
By (Config Good), $H \models S \cdot (v \ at \ \rho, h) : A$ means that $env(H) \vdash v \ at \ \rho :^{e} A$ for some $e \subseteq S$, and that $H \models h$ and $S \subseteq dom(H)$. Since only (Exp Nil), (Exp Cons) or (Exp Fun) can derive $env(H) \vdash v \ at \ \rho :^{e} A$, we have $A = V \ at \ \rho$, for some V, and $e = \{\rho\}$.
Let H' be the heap typing $\rho \mapsto (p{:}A)$. Since $p \notin dom_2(H) = dom_2(h)$, we have $env(H + H') \vdash \diamond$. Hence, $env(H + H') \vdash p :^{\emptyset} A$. Moreover $S \subseteq dom(H + H')$.
By (Heap Good), $H \models h$ and $\rho \in S$ imply that $env(H) \vdash h(\rho) \ at \ \rho : H(\rho)$. Therefore, $env(H + H') \vdash h(\rho) \ at \ \rho + (p \mapsto v \ at \ \rho) : H(\rho) + (p{:}A)$. Hence, $H + H' \models h + \rho \mapsto (h(\rho) + p \mapsto v)$.
We have $env(H + H') \vdash p \ at \ \rho : A$ and $S \subseteq dom(H + H')$ 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 $S \cdot (p[\rho'_1, \ldots, \rho'_n](q), h) \Downarrow (p', h')$ 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 $\mu(f{:}F)\lambda[\rho_1, \ldots, \rho_n](x)b$ and $\sigma = \{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$ and $F = \forall[\rho_1, \ldots, \rho_n](B_1 \xrightarrow{e'} B_2) \ at \ \rho$.
By (Config Good), $H \models S \cdot (p[\rho'_1, \ldots, \rho'_n](q), h) : A$ means that $env(H) \vdash p[\rho'_1, \ldots, \rho'_n](q) :^{e} A$ for some $e \subseteq S$, and that $H \models h$ and $S \subseteq dom(H)$.
Only (Exp Appl) can derive $env(H) \vdash p[\rho'_1, \ldots, \rho'_n](q) :^{e} A$ and so we have $env(H) \vdash p :^{\emptyset} F$ and $env(H) \vdash q :^{\emptyset} B_1\sigma$ and $A = B_2\sigma$, where $e = \{\rho\} \cup e'\sigma$ and $\{\rho'_1, \ldots, \rho'_n\} \subseteq dom(H)$. Since $H \models h$ and $\rho \in dom(H)$, we have that $env(H) \vdash h(\rho) \ at \ \rho : H(\rho)$, and in particular, $env(H) \vdash \mu(f{:}F)\lambda[\rho_1, \ldots, \rho_n](x)b \ at \ \rho :^{\{\rho\}} F$.
Only (Exp Fun) can derive $env(H) \vdash \mu(f{:}F)\lambda[\rho_1, \ldots, \rho_n](x)b \ at \ \rho :^{\{\rho\}} F$, and so $env(H), f{:}F, \rho_1, \ldots, \rho_n, x{:}B_1 \vdash b :^{e''} B_2$ where $e'' \subseteq e' \subseteq dom(E, \rho_1, \ldots, \rho_n)$. By Lemma B.3 and B.4, since $env(H) \vdash q :^{\emptyset} B_1\sigma$ and $env(H) \vdash p :^{\emptyset} F$, we get that $env(H) \vdash b\{f \leftarrow p\}\{x \leftarrow q\}\sigma :^{e''\sigma} B_2\sigma$.
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 $S \cdot (let \ x = b \ in \ a, h) \Downarrow (p'', h'')$ derives from $S \cdot (b, h) \Downarrow (p', h')$ and $S \cdot (a\{x \leftarrow p'\}, h') \Downarrow (p'', h'')$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

By (Config Good), $H \models S \cdot (let\ x = b\ in\ a, h) : A$ means that $env(H) \vdash let\ x = b\ in\ a :^{e} A$ for some $e \subseteq S$, and that $H \models h$ and $S \subseteq dom(H)$.
Only (Exp Let) can derive $env(H) \vdash let\ x = b\ in\ a :^{e} A$ and so we have $env(H) \vdash b :^{e_b} B$ and $env(H), x{:}B \vdash a :^{e_a} A$ for some e_b, e_a and B, such that $e = e_a \cup e_b$. By (Config Good), $env(H) \vdash b :^{e_b} B$ and $e_b \subseteq S$ and $H \models h$ and $S \subseteq dom(H)$ 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 $env(H + H') \vdash p' :^{e'} B$ for some $e' \subseteq S$, and that $H + H' \models h'$ and $S \subseteq dom(H + H')$.
Only (Exp x) or (Exp l) can derive this and so it must be that $e' = \emptyset$.
By Lemma B.2, since $env(H), x{:}B \vdash a :^{e_a} A$, we get that $env(H + H'), x{:}B \vdash a :^{e_a} A$.
By Lemma B.3, since $env(H + H') \vdash p' :^{\emptyset} B$, we get that $env(H + H') \vdash a\{x \leftarrow p'\} :^{e_a} A$. Therefore, by (Config Good), $env(H + H') \models S \cdot (a\{x \leftarrow p'\}, h') : A$.
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 $S \cdot (letregion\ \rho\ in\ a, h) \Downarrow (p', h')$ derives from $S \cup \{\rho\} \cdot (a, h + \rho \mapsto \emptyset) \Downarrow (p', h')$ with $\rho \notin dom(h)$. By (Config Good), $H \models S \cdot (letregion\ \rho\ in\ a, h) : A$ means that $env(H) \vdash letregion\ \rho\ in\ a :^{e} A$ and $H \models h$ for some $e \subseteq S$.
Only (Exp Letregion) can derive this and so we have $env(H), \rho \vdash a :^{e'} A$ with $env(H) \vdash A$ and $e = e' - \{\rho\}$. In particular $\rho \notin dom(H)$.
Let H' be the heap typing $(\rho \mapsto \emptyset)$. We have $H \asymp H'$ and $env(H + H') = env(H), \rho$.
By (Config Good), $env(H + H') \vdash a :^{e'} A$, and $e' \subseteq S \cup \{\rho\}$ and $S \cup \{\rho\} \subseteq dom(H + H')$ 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 $H + H' \asymp H''$ and $(H + H') + H'' \models (S \cup \{\rho\}) \cdot (p', h') : A$. To complete the case, note that $H \asymp H' + H''$ and $H + (H' + H'') \models S \cdot (p', h') : A$.

(Eval Case 1) Then $S \cdot (case\ p\ of\ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) \Downarrow (p', h')$. This derives from $S \cdot (b_1, h) \Downarrow (p', h')$ with $\rho \in S$ and $h(\rho)(p) = nil$.
By (Config Good), $H \models S \cdot (case\ p\ of\ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) : A$ means that $env(H) \vdash case\ p\ of\ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2 :^{e} A$ and $H \models h$ for some $e \subseteq S$.
Only (Exp Case) can derive this and so we have $env(H) \vdash p :^{\emptyset} [B]\ at\ \rho$ and $env(H) \vdash b_1 :^{e_1} A$ and $env(H), y_1{:}B, y_2{:}[B]\ at\ \rho \vdash b_2 :^{e_2} A$.
By induction hypothesis, since $S \cdot (b_1, h) \Downarrow (p', h')$, we get that there is H' with $H \asymp H'$ and $H + H' \models S \cdot (p', h') : A$, as required.

(Eval Case 2) Then $S \cdot (case\ p\ of\ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) \Downarrow (p', h')$. 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), $H \models S \cdot (case\ p\ of\ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2, h) : A$ means that $env(H) \vdash case\ p\ of\ nil \Rightarrow b_1 \mid (y_1 :: y_2) \Rightarrow b_2 :^{e} A$ and $H \models h$ for some $e \subseteq S$.
Only (Exp Case) can derive this and so we have $env(H) \vdash p :^{\emptyset} [B]\ at\ \rho$ and $env(H) \vdash b_1 :^{e_1} A$ and $env(H), y_1{:}B, y_2{:}[B]\ at\ \rho \vdash b_2 :^{e_2} A$.
Therefore, by hypothesis, $h(\rho)(p) = q_1 :: q_2$ and $env(H) \vdash p :^{\emptyset} [B]\ at\ \rho$. Hence, by (Config Good), $env(H) \vdash (q_1 :: q_2)\ at\ \rho :^{\{\rho\}} [B]\ at\ \rho$. Only (Exp Cons) can derive this and so we have $E \vdash q_1 :^{\emptyset} B$ and $E \vdash q_2 :^{\emptyset} [B]\ at\ \rho$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

By Lemma B.3, since $env(H), y_1{:}B, y_2{:}[B]$ *at* $\rho \vdash b_2 :^{e_2} A$, we get that $env(H) \vdash b_2\{y_1 \leftarrow q_1\}\{y_2 \leftarrow q_2\} :^{e_2} A$.
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 $H \asymp H'$ and $H + H' \models S \cdot (p', h') : A$, as required. □

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.6

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

Proof

For part (1), assume $H \models S \cdot (v \ at \ \rho, h) : A$. By (Config Good) we get that $env(H) \vdash (v \ at \ \rho) :^{e} A$ for some effect e, with $e \cup fg(A) \subseteq S$. 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 $H \models S \cdot (p[\rho'_1, \ldots, \rho'_n](q), h) : A$. By (Config Good) we get that $H \models h$ and that $env(H) \vdash p[\rho'_1, \ldots, \rho'_n](q) :^{e} A$ for some effects e, with $e \cup fg(A) \subseteq S$. Only (Exp Appl) can derive this and so we have $env(H) \vdash p :^{\varnothing} F$ for some region ρ, with $F = (\forall[\rho_1, \ldots, \rho_n](B \xrightarrow{e'} A))$ *at* ρ and $e = \{\rho\} \cup e'\{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$. Only (Exp x) can derive $env(H) \vdash p :^{\varnothing} F$ and so there is a region ρ' such that $H(\rho')(p) = F$. By (Heap Good), $\rho' = \rho$ and $\rho \in dom(H)$ and there is a value v such that $H(\rho)(p) = v$. By (Region Good), $env(H) \vdash v \ at \ \rho :^{\{\rho\}} F$. Only (Exp Fun) can derive this. Hence, v is a function of the form $\mu(f{:}F)\lambda[\rho_1, \ldots, \rho_n](x)b$ and there is e'' such that $e'' \subseteq e' \subseteq dom(E, \rho_1, \ldots, \rho_n)$ and $env(H), f{:}F, \rho_1, \ldots, \rho_n, x{:}B \vdash b :^{e''} A$. □

B.2 Subject reduction for the π-calculus

We show that reduction in the π-calculus preserves types and effects. As in the previous section on subject reduction for the λ-calculus, we use intermediate results whose proofs can be found in a technical report (Dal Zilio & Gordon, 2000b). Again, we use the symbol $\mathscr{J}$ 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.7
If $E \vdash P : \mathbf{H}$ then $\mathbf{H} \subseteq dom(E)$.

Lemma B.8
If $E \vdash \mathscr{J}$ then $fg(\mathscr{J}) \subseteq dom(E)$.

Lemma B.9
If $E \vdash x : T_1$ and $E \vdash x : T_2$, where T_1 and T_2 are channel types of the form $G(H_1, \ldots, H_m)[T'_1, \ldots, T'_n]\backslash\mathbf{H}$, then $T_1 = T_2$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Lemma B.10
If $E, x_1{:}T_1, x_2{:}T_2, E' \vdash \mathscr{J}$ then $E, x_2{:}T_2, x_1{:}T_1, E' \vdash \mathscr{J}$.

Lemma B.11
If $E, x{:}T, G, E' \vdash \mathscr{J}$ then $E, G, x{:}T, E' \vdash \mathscr{J}$.

Lemma B.12
If $E, x{:}T, E' \vdash \mathscr{J}$ and $x \notin fn(\mathscr{J})$ then $E, E' \vdash \mathscr{J}$.

Lemma B.13
If $E \vdash \mathscr{J}$ and $E, E' \vdash \diamond$ then $E, E' \vdash \mathscr{J}$.

Lemma B.14
If $E, x{:}T, E' \vdash \mathscr{J}$ and $E \vdash y : T$ then $E, E' \vdash \mathscr{J}\{x \leftarrow y\}$.

Lemma B.15
If $E, G, E' \vdash \mathscr{J}$ and $H \in dom(E)$ then we have $E, E'\{G \leftarrow H\} \vdash \mathscr{J}\{G \leftarrow H\}$, where $E'\{G \leftarrow H\}$ is the result of applying the substitution $\{G \leftarrow H\}$ to each of the types in E'.

Lemma B.16
If $E \vdash P : \mathbf{H}$ and $P \equiv Q$ then $E \vdash Q : \mathbf{H}$.

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.17
If $E \vdash P : \mathbf{H}$ and $P \to Q$ then $E \vdash Q : \mathbf{H}$.

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

(Red Interact) Then P is a parallel composition $\overline{x}\langle G'_1, \ldots, G'_m, y'_1, \ldots, y'_n\rangle \mid x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P'$ 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.7, $\mathbf{H} \subseteq dom(E)$. The judgment $E \vdash P : \mathbf{H}$ must have been derived from (Proc Par), with $E \vdash \overline{x}\langle G'_1, \ldots, G'_m, y'_1, \ldots, y'_n\rangle : \mathbf{H}_1$, and $E \vdash x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P' : \mathbf{H}_2$ where $\mathbf{H} = \mathbf{H}_1 \cup \mathbf{H}_2$. The former must have been derived from a number of subsumption steps implying $E \vdash \overline{x}\langle G'_1, \ldots, G'_m, y'_1, \ldots, y'_n\rangle : \mathbf{H}_3$, where $\mathbf{H}_3 \subseteq \mathbf{H}_1$, followed by (Proc Output), with $E \vdash y'_1 : T_1\sigma, \cdots, E \vdash y'_n : T_n\sigma$, and $E \vdash x : G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{G}$, and:

$$\{G\} \cup \mathbf{G}\sigma = \mathbf{H}_3 \tag{B 1}$$

By Lemma B.9, the latter must have been derived from (Proc Input), with $E, G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n \vdash P' : \mathbf{H}_4$ and $(\mathbf{H}_4 - \mathbf{G}) \cap \{G_1, \ldots, G_m\} = \emptyset$, followed by a number of subsumption steps implying $\{G\} \cup (\mathbf{H}_4 - \mathbf{G}) \subseteq \mathbf{H}_2 \subseteq dom(E)$ by transitivity. In particular, we have that:

$$(\mathbf{H}_4 - \mathbf{G})\sigma = (\mathbf{H}_4 - \mathbf{G}) \subseteq \mathbf{H}_2 \tag{B 2}$$

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

By Lemma B.8, since $E \vdash \overline{x}\langle G'_1, \ldots, G'_m, y'_1, \ldots, y'_n\rangle$, we have that $\{G'_1, \ldots, G'_m\} \subseteq dom(E)$. Then, by Lemma B.15 several times, it follows that $E, y_1{:}T_1\sigma, \ldots, y_n{:}T_n\sigma \vdash P'\sigma : \mathbf{H}_4\sigma$. By Lemma B.14, $E \vdash P'\sigma\ \{y_1{\leftarrow}y'_1\} \cdots \{y_n{\leftarrow}y'_n\} : \mathbf{H}_4\sigma$.

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 (B 1) and (B 2), we get that $\mathbf{H}_4\sigma \subseteq (\{G\} \cup (\mathbf{H}_4 - \mathbf{G})\sigma) \cup (\{G\} \cup \mathbf{G}\sigma) \subseteq (\mathbf{H}_2 \cup \mathbf{H}_3) \subseteq (\mathbf{H}_2 \cup \mathbf{H}_1) = \mathbf{H} \subseteq dom(E)$. 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 $P' \to Q'$. 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 $P' \to Q'$. 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 $P' \to Q'$. 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 $P' \to Q'$. Assume $E \vdash P : \mathbf{H}$. By Lemma B.16, $E \vdash P' : \mathbf{H}$. By induction hypothesis, $E \vdash Q' : \mathbf{H}$. By Lemma B.16, $E \vdash Q : \mathbf{H}$. □

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.18

If $E \vdash P : \mathbf{H}$ and $P \downarrow \beta$ with $\beta \in \{x, \overline{x}\}$ then there is a type polymorphic channel type $T \triangleq G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{G}$ such that $E \vdash x : T$ and $G \in \mathbf{H}$.

Proof

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

(Barb Input) Then $P = x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P'$ 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 $E \vdash x : G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{G}$ 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 $P = \overline{x}\langle G'_1, \ldots, G'_m, y'_1, \ldots, y'_n\rangle$ and $\beta = \overline{x}$. Assume $E \vdash P : \mathbf{H}$. This must have been derived from (Proc Output) with $E \vdash P : \mathbf{H}_1$ and $E \vdash x : G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{G}$ 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 $W \triangleq G(G_1, \ldots, G_m)[T_1, \ldots, T_n]\backslash\mathbf{G}$ such that $E \vdash x : W$ and $G \in \mathbf{H}_2$. Hence, $G \in \mathbf{H}$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

(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 $W \triangleq G(G_1, \ldots, G_m)\ [T_1, \ldots, T_n]\backslash\mathbf{G}$ such that $E, y{:}T \vdash x : W$ and $G \in \mathbf{H}_1$. Hence $G \in \mathbf{H}$. By Lemma B.12, $E \vdash x : W$.

(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 $W \triangleq G(G_1, \ldots, G_m)\ [T_1, \ldots, T_n]\backslash\mathbf{G}$ such that $E \vdash x : W$ and $G \in \mathbf{H}'$. Hence, $G \in \mathbf{H}$.

(Barb ≡) Then $P \equiv P'$ for some P' such that $P' \downarrow \beta$. Assume $E \vdash P : \mathbf{H}$. By Lemma B.16, $E \vdash P' : \mathbf{H}$.

By induction hypothesis, there is a type $W \triangleq G(G_1, \ldots, G_m)\ [T_1, \ldots, T_n]\backslash\mathbf{G}$ such that $E \vdash x : W$ and $G \in \mathbf{H}$. □

B.3 Correctness of type erasure

In this section, we study the relations between the typed and untyped versions of the π-calculus defined in this paper. We prove Proposition B.22, which gives a simple correspondence between the reductions of a typed process, P, and the reductions of the untyped process obtained by erasing all type information 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 processes. 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.19
For all typed processes P and Q, if $P \equiv Q$ then $erase(P) \equiv erase(Q)$.

Lemma B.20
For all typed processes P, $P \downarrow \beta$ if and only if $erase(P) \downarrow \beta$.

Lemma B.21
If $E \vdash P$ and $erase(P) \xrightarrow{\tau} R$ then there exists a typed process, Q, such that $P \rightarrow Q$ and $erase(Q) = R$.

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.22
For all typed processes P and Q, if $P \rightarrow Q$ then $erase(P) \rightarrow erase(Q)$. If $E \vdash P$ and $erase(P) \rightarrow R$ then there is a typed process Q such that $P \rightarrow Q$ and $R \equiv erase(Q)$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Proof
The first implication is proved by a simple induction on the derivation of $P \to Q$, with appeal to Lemma B.19.

Assume $E \vdash P$ and $\mathit{erase}(P) \to R$. By Proposition A.1, there is an untyped process S such that $\mathit{erase}(P) \xrightarrow{\tau} S$ and $S \equiv R$. By Lemma B.21, there exists a typed process Q such that $P \to Q$ and $\mathit{erase}(Q) = S$. By (Struct Trans), $\mathit{erase}(Q) \equiv R$, as required. □

Using Proposition B.22, it is possible to prove that two typed processes with equivalent erasures, according to the untyped barbed congruence defined in Appendix A.4, are barbed congruent.

Proposition B.23
If $E \vdash P$ and $E \vdash Q$ and $\mathit{erase}(P) \approx \mathit{erase}(Q)$ then $E \vdash P \approx Q$.

B.4 Properties of Barbed Congruence

In this section we study some properties of typed barbed congruence. We prove 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.27, that barbed congruence is indeed a compositional equivalence relation. The proof of this property relies on Lemma B.26, 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 respect the type information.

Lemma B.24
If $P \equiv Q$ and $E \vdash P$ then $E \vdash P \approx Q$

Proof
Let $\mathscr{S}$ be the smallest relation on typed processes that contains $\approx$ and such that $E \vdash P \;\mathscr{S}\; Q$ if $P \equiv Q$ and $E \vdash P$. The relation $\mathscr{S}$ 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.17, $E \vdash Q$. Note that $\mathscr{S}$ is symmetric. We prove that $\mathscr{S}$ 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 $P \to P'$. By (Red $\equiv$), $Q \to P'$ and, since $\approx$ is reflexive, $P' \;\mathscr{S}\; P'$, as required.

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

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.14 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, $E\sigma, E' \vdash P\sigma \mid R \mathrel{\dot{\approx}} Q\sigma \mid R$. Hence, $E \vdash P \approx Q$, as required. □

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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Lemma B.25
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.24 and transitivity of $\approx$, since $P\sigma \mid \mathbf{0} \equiv P\sigma$ and $Q\sigma \mid \mathbf{0} \equiv Q$, we get that $E\sigma \vdash P\sigma \approx Q\sigma$. □

We introduce some new notations to simplify the presentation of the following properties. If E is a type environment $G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n$, let $x(E).P$ be the process $x(G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n).P$. 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 $\sigma(G) \notin dom(E'\sigma)$, and $E'\sigma$ otherwise. We have:

Lemma B.26
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.27
(1) Let E' be the environment $G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n$. 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.24, 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 $x(E').P \mid R \downarrow \overline{x}$. By Lemma B.20, Proposition A.2 and inspection of the possible transitions, it must be the case that $R \downarrow \overline{x}$. By (Barb Par) and (Barb ≡), since $R \downarrow \overline{x}$, we get that $x(E').Q \mid R \downarrow \overline{x}$, as required.
(2) Assume $x(E').P \mid R \rightarrow P'$. Suppose E' is the type environment $G_1, \ldots, G_m, y_1{:}T_1, \ldots, y_n{:}T_n$. By Propositions B.22 and A.1 and inspection of the possible transitions, either (1) $R \rightarrow R'$ and $P' \equiv x(E').P \mid R'$, or (2) $R \equiv (\nu E'')(\overline{x}\langle H_1, \ldots, H_m, z_1, \ldots, z_n\rangle \mid R')$ 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 $\sigma_y = \{y_1 \leftarrow z_1\} \cdots \{y_n \leftarrow z_n\}$.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

For (1), by (Red Par), $x(E').Q \mid R \to x(E').Q \mid R'$. By Proposition B.17, $E \vdash R'$. Hence, $E \vdash x(E').P \mid R' \mathrel{\mathscr{S}} x(E').Q \mid R'$, 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 $\equiv$), $x(E').Q \mid R \to Q'$. By Lemma B.26, 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 $dom(E') \cap dom(E'') = \emptyset$. By Lemma B.13 several times and since $E, E'' \vdash \overline{x}\langle H_1, \ldots, H_m, z_1, \ldots, z_n\rangle \mid R'$, we get that $E''' \vdash P\sigma_G \approx Q\sigma_G$, where E''' is the type environment $E, E'', y_1{:}T_1\sigma_G, \ldots, y_n{:}T_n\sigma_G$. Since $E \vdash x(E').P \mid R$, $E \vdash x(E').Q \mid R$ and $E, E'' \vdash \overline{x}\langle H_1, \ldots, H_m, z_1, \ldots, z_n\rangle \mid R'$, the substitution σ_y is an E'''-renaming. By Lemma B.25, since $(E, E')\sigma_G \vdash P\sigma_G \approx Q\sigma_G$ and $z_i \in dom(E, E'')$ 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 $\mathscr{S}$, we get that $E \vdash P' \mathrel{\mathscr{S}} Q'$, as required.

Therefore $\mathscr{S}$ is a barbed bisimulation and if $E, E' \vdash P \approx Q$ then $E \vdash x(E').P \mid R \mathrel{\dot{\approx}} x(E').Q \mid R$ 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 $dom(E') \cap dom(E'') = \emptyset$. Therefore, by Lemmas B.13 and B.14, $E\sigma, E''$, $E' \vdash P\sigma \approx Q\sigma$, and then $E\sigma, E'' \vdash x(E').P\sigma \mid R \mathrel{\dot{\approx}} x(E').Q\sigma \mid R$. Hence, $E \vdash x(E').P \approx x(E').Q$, as desired. □

B.5 Garbage collection for the π-calculus

In this section we prove Theorem B.30, 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.28 shows that these processes cannot synchronize. In the sense that their parallel composition do not introduce new silent transitions. This property is essential in the proof of Lemma B.29, an equivalent of the garbage collection property for the barbed bisimilarity relation.

Lemma B.28
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 $erase(P \mid R) \xrightarrow{\alpha} A$ then there is an agent A' such that $erase(P) \xrightarrow{\alpha} A'$ and $A = A' \mid erase(R)$, or such that $erase(R) \xrightarrow{\alpha} A'$ and $A = erase(P) \mid A'$.

Lemma B.29
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: $E \vdash (\nu G)(\nu E')(P \mid R) \mathrel{\dot{\approx}} (\nu G)(\nu E')P$.

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.

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Theorem B.30
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 $dom(G, E')$ are bound, we may assume that $dom(G, E') \cap (dom(E'') \cup dom(\sigma) \cup ran(\sigma)) = \emptyset$. Hence, since $fn(Q) \subseteq dom(E\sigma, E'')$ and $\mathbf{G} \subseteq dom(E\sigma, E'')$, we get that $fn(Q) \cap dom(G, E') = \emptyset$ and $G \notin \mathbf{G}$. By Lemma B.14 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.13, $E\sigma, E'', G, E' \vdash Q : \mathbf{G}$. 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 $dom(G, E') \cap (dom(E'') \cup dom(\sigma) \cup ran(\sigma)) = \emptyset$, 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.29, $E\sigma, E' \vdash (\nu G)(\nu E')(P \mid R)\sigma \mid Q \dot{\approx} ((\nu G)(\nu E')P)\sigma \mid Q$, as required. □

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 section 6.1.

In section B.6.3, we prove Theorem B.35, 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.33, which relates the small-step and big-step semantics.

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 heap all the references stored in defunct regions.

B.6.1 Proof of static adequacy

We prove Theorem B.32, 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. The proof of this property uses an intermediate

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

result, Lemma B.31, that the encoding of a well-typed value stored in region ρ is a well-typed process with effect $\{\rho\}$.

Lemma B.31
If $H \models h$ and $h(\rho)(p) = v$ then $[\![env(H)]\!] \vdash [\![p \mapsto v]\!] : \{\rho\}$.

Theorem B.32 (*Static Adequacy*)
(1) If $E \vdash \diamond$ then $[\![E]\!] \vdash \diamond$.
(2) If $E \vdash A$ then $[\![E]\!] \vdash [\![A]\!]$.
(3) If $E \vdash a :^e A$ and $k \notin dom([\![E]\!])$ then $[\![E]\!], k{:}K\,[[\![A]\!]] \vdash [\![a]\!]k : e \cup \{K\}$
(4) If $H \models h$ and $\rho \in dom(H)$ then $[\![env(H)]\!] \vdash [\![h(\rho)]\!] : \{\rho\}$
(5) If $H \models S \cdot (a, h) : A$ and $k \notin [\![env(H)]\!]$ then

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

and also $[\![\varnothing]\!], S, k{:}K\,[[\![A]\!]] \vdash [\![S \cdot (a, h)]\!]k : S \cup \{K\}$

Proof
Parts (1) and (2) follow easily by induction on the structure of E. We prove part (3) by induction on the derivation of $E \vdash a :^e A$. Recall that $G[T_1, \ldots, T_n]$ is a shorthand for the type $G()[T_1, \ldots, T_n]\backslash\varnothing$.

(Exp x) Then $a = x$ and $E = E_1, x{:}A, E_2$ and $e = \varnothing$. Assume $k \notin L \cup dom(E)$. Then $[\![E]\!] \vdash x : [\![A]\!]$. By (Proc Output) and Lemma B.13, $[\![E]\!], k{:}K\,[[\![A]\!]] \vdash \overline{k}\langle x\rangle : \{K\}$, as required.

(Exp l) Then $a = l$, where $l \in L$ and $A = Lit$. By definition, $[\![E]\!] \vdash l : Lit\,[]$. Assume $k \notin L \cup dom(E)$. By (Proc Output) and Lemma B.13, $[\![E]\!], k{:}K\,[[\![A]\!]] \vdash \overline{k}\langle l\rangle : \{K\}$, as required.

(Exp Appl) Then $a = x[\rho'_1, \ldots, \rho'_n](y)$ and $e = \{\rho\} \cup e'\sigma$, with $E \vdash x :^{\varnothing} F$ and $E \vdash y :^{\varnothing} B_1\sigma$ and $F = (\forall[\rho_1, \ldots, \rho_n]B_1 \xrightarrow{e'} B_2)\ at\ \rho$ and $\sigma = \{\rho_1 \leftarrow \rho'_1\} \cdots \{\rho_n \leftarrow \rho'_n\}$ and $A = B_2\sigma$ and $\{\rho'_1, \ldots, \rho'_n\} \subseteq dom(E)$. Then $[\![E]\!] \vdash x : [\![F]\!]$ and $[\![E]\!] \vdash y : [\![B_1\sigma]\!]$. Assume $k \notin L \cup dom(E)$. By (Exp Unfold) and Lemma B.13:

$$[\![E]\!], k{:}K\,[[\![A]\!]] \vdash x : \rho(\rho_1, \ldots, \rho_n)[[\![B_1]\!], K\,[[\![B_2]\!]]]\backslash(e' \cup \{K\})$$

By (Proc Output), $[\![E]\!], k{:}K\,[[\![A]\!]] \vdash \overline{x}\langle\rho'_1, \ldots, \rho'_n, y, k\rangle : e'\sigma \cup \{K\}$, 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:

$$\begin{cases} [\![E]\!], k'{:}K\,[[\![B]\!]] \vdash [\![b]\!]k' : e' \cup \{K\} \\ [\![E]\!], x{:}[\![B]\!], k{:}K\,[[\![A]\!]] \vdash [\![c]\!]k : e'' \cup \{K\} \end{cases}$$

By (Proc Input) and Lemmas B.12 and B.10:

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

By (Proc Par) and (Proc Res):

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

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

(Exp Letregion) Then $a = \mathit{letregion}\ \rho\ \mathit{in}\ b$ and $e = e' - \{\rho\}$, with $E, \rho \vdash b :^{e'} A$ and $E \vdash A$. Assume $k \notin L \cup \mathit{dom}(E)$. By induction hypothesis, $[\![E]\!], \rho, k{:}K[[\![A]\!]] \vdash [\![b]\!]k : e' \cup \{K\}$. By part (2), $[\![E]\!] \vdash [\![A]\!]$. Therefore, since $\rho \notin \mathit{dom}(E)$, we have that $\rho \notin \mathit{fg}(K[[\![A]\!]])$ and, by Lemma B.11, $[\![E]\!], k{:}K[[\![A]\!]], \rho \vdash [\![b]\!]k : e' \cup \{K\}$. By (Proc GRes), $[\![E]\!], k{:}K[[\![A]\!]] \vdash (\nu\rho)[\![b]\!]k : (e' \cup \{K\}) - \{\rho\}$, as required.

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

$$\begin{cases} [\![E]\!] \vdash x : [\![[B]\ at\ \rho]\!] \\ [\![E]\!], k{:}K[[\![A]\!]] \vdash [\![b_1]\!]k : e_1 \cup \{K\} \\ [\![E]\!], y_1{:}[\![B]\!], y_2{:}[\![[B]\ at\ \rho]\!], k{:}K[[\![A]\!]] \vdash [\![b_2]\!]k : e_2 \cup \{K\} \end{cases}$$

By (Exp Unfold), $[\![E]\!] \vdash x : \rho[\rho[], \rho[[\![B]\!], [\![[B]\ at\ \rho]\!]]]$. By Lemma B.13 and (Proc Output):

$$\begin{cases} [\![E]\!], k{:}K[[\![A]\!]], z_1{:}\rho[], z_2{:}\rho[[\![B]\!], [\![[B]\ at\ \rho]\!]] \vdash \overline{x}\langle z_1, z_2\rangle : \{\rho\} \\ [\![E]\!], k{:}K[[\![A]\!]], z_1{:}\rho[], z_2{:}\rho[[\![B]\!], [\![[B]\ at\ \rho]\!]] \vdash z_1().[\![b_1]\!]k : e_1 \cup \{K\} \\ [\![E]\!], k{:}K[[\![A]\!]], z_1{:}\rho[], z_2{:}\rho[[\![B]\!], [\![[B]\ at\ \rho]\!]] \vdash \\ \qquad z_2(y_1{:}[\![B]\!], y_2{:}[\![[B]\ at\ \rho]\!]).[\![b_2]\!]k : e_2 \cup \{K\} \end{cases}$$

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

(Exp Fun) Then $a = v\ at\ \rho$ and $e = \{\rho\}$ and $A = (\forall[\rho_1, \ldots, \rho_n]B_1 \xrightarrow{e} B_2)\ at\ \rho$, where v is the function $(\mu(f{:}A)\lambda[\rho_1, \ldots, \rho_n](x)b)$ and $E, f{:}A,\ \rho_1, \ldots, \rho_n, x{:}B_1 \vdash b :^{e'} B_2$ and $e' \subseteq e \subseteq \mathit{dom}(E, \rho_1, \ldots, \rho_n)$. Assume $k \notin L \cup \mathit{dom}(E) \cup \{p\}$. Since f and p are bound names, we can also assume that $k \notin \{f, p\}$. By induction hypothesis and Lemma B.13:

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

By Lemma B.14:

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

By (Exp x) and (Exp Unfold):

$$[\![E]\!], p{:}[\![A]\!] \vdash p : \rho(\rho_1, \ldots, \rho_n)[[\![B_1]\!], K[[\![B_2]\!]]] \backslash (e \cup \{K\})$$

By (Proc Input), $[\![E]\!], p{:}[\![A]\!] \vdash [\![p \mapsto v]\!] : \{\rho\}$. By (Proc Output) and Lemma B.13, $[\![E]\!], k{:}K[[\![A]\!]], p{:}[\![A]\!] \vdash \overline{k}\langle p\rangle : \{K\}$. By (Proc Par) and (Proc Res), $[\![E]\!], k{:}K[[\![A]\!]] \vdash (\nu p{:}[\![A]\!])\ ([\![p \mapsto v]\!] \mid \overline{k}\langle p\rangle) : \{K, \rho\}$, as required.

(Exp Nil) Then $a = \mathit{nil}\ at\ \rho$ and $e = \{\rho\}$ and $A = [B]\ at\ \rho$, where $E \vdash [B]\ at\ \rho$. By part (2), $[\![E]\!] \vdash [\![[B]\ at\ \rho]\!]$. By Lemma B.13:

$$[\![E]\!], p{:}[\![A]\!], z_1{:}\rho[], z_2{:}\rho[[\![B]\!], [\![[B]\ at\ \rho]\!]] \vdash \overline{z_1}\langle\rangle : \{\rho\}$$

By (Exp x) and (Exp Unfold):

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

By (Proc Input), $[\![E]\!], p{:}[\![A]\!] \vdash [\![p \mapsto nil_{[B]at\,\rho}]\!] : \{\rho\}$. Assume $k \notin L \cup dom(E) \cup \{p\}$. By (Proc Output) and Lemma B.13, $[\![E]\!], k{:}K\,[\![A]\!], p{:}[\![A]\!] \vdash \overline{k}\langle p\rangle : \{K\}$. By (Proc Par) and (Proc Res), $[\![E]\!], k{:}K\,[\![A]\!] \vdash (\nu p{:}[\![A]\!])([\![p \mapsto nil_A]\!] \mid \overline{k}\langle p\rangle) : \{K, \rho\}$, as required.

(Exp Cons) Then $a = (x_1 :: x_2)\ at\ \rho$ and $e = \{\rho\}$ and $A = [B]\ at\ \rho$, where $E \vdash x_1 :^{\varnothing} B$ and $E \vdash x_2 :^{\varnothing} [B]\ at\ \rho$. By part (1), $[\![E]\!] \vdash x_1 : [\![B]\!]$ and $[\![E]\!] \vdash x_2 : [\![[B]\ at\ \rho]\!]$. By (Exp x), (Exp Unfold) and Lemma B.13:

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

By (Proc Output):

$$[\![E]\!], p{:}[\![A]\!], z_1{:}\rho[\,], z_2{:}\rho[[\![B]\!], [\![[B]\ at\ \rho]\!]] \vdash \overline{z_2}\langle x_1, x_2\rangle : \{\rho\}$$

By (Proc Input), $[\![E]\!], p{:}[\![A]\!] \vdash [\![p \mapsto (x_1 :: x_2)_A]\!] : \{\rho\}$. Assume $k \notin L \cup dom(E) \cup \{p\}$. By (Proc Output) and Lemma B.13, $[\![E]\!], k{:}K\,[\![A]\!], p{:}[\![A]\!] \vdash \overline{k}\langle p\rangle : \{K\}$. By (Proc Par) and (Proc Res), $[\![E]\!], k{:}K\,[\![A]\!] \vdash (\nu p{:}[\![A]\!])([\![p \mapsto (x_1 :: x_2)_A]\!] \mid \overline{k}\langle p\rangle) : \{K, \rho\}$, as required.

We prove part (4) by case analysis. 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 $i \in 1..n$. Then $[\![env(H)]\!] \vdash p_i : [\![A_i]\!]$. By Lemma B.31, $[\![env(H)]\!] \vdash [\![p_i \mapsto v_i]\!] : \{\rho\}$ for each $i \in 1..n$. By (Proc Par), $[\![env(H)]\!] \vdash \prod_{i\in 1..n} [\![p_i \mapsto v_i]\!] : \{\rho\}$. Hence, $[\![env(H)]\!] \vdash [\![h(\rho)]\!] : \{\rho\}$.

We prove part (5) by case analysis. 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 part (3), $[\![env(H)]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k : e \cup \{K\}$. By part (4) and (Proc Par), $[\![env(H)]\!] \vdash [\![h]\!] : \bigcup_{\rho\in dom(H)}\{\rho\}$. By (Proc Par) and Lemma B.13, $[\![env(H)]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \mid [\![h]\!] : dom(H) \cup e \cup \{K\}$. Since $e \subseteq dom(H)$, we get that $[\![env(H)]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \mid [\![h]\!] : dom(H) \cup \{K\}$, as desired.

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

B.6.2 An auxiliary small-step semantics

This section defines an auxiliary small-step semantics for the region calculus. We prove Theorem B.33, that relates small-step reductions to evaluations in the big-step semantics.

Continuations and Control Stack:

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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)

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

(Red Appl)

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

(Red Let)

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

(Red Pop Let)

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

(Red Letregion)

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

(Red Pop Letregion)

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

(Red Case 1)

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

(Red Case 2)

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

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Heap Judgments:

$\vec{B} ::= [A_1, \ldots, A_n]$	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)
$$\frac{env(H) \vdash A \quad fg(A) \subseteq S}{H \models S \cdot [] : [A]}$$

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

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

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

Theorem B.33
Suppose $H \models S \cdot (a, h) : A$. Then $S \cdot (a, h) \Downarrow (p', h')$ if and only if $S \cdot (a, h, []) \rightarrow^{*} S \cdot (p', h', [])$.

B.6.3 Proof of dynamic adequacy

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

Translation Rules:

Let $\vec{k}$ be a stack, $[k_1, \ldots, k_n]$, of n pairwise distinct names.
$[\![(x{:}A)b :: C]\!]\vec{k} \triangleq k_1(x{:}K\,[\![A]\!]).[\![b]\!]k_2 \mid [\![C]\!][k_2, \ldots, k_n]$
$[\![popregion\ \rho :: C]\!]\vec{k} \triangleq [\![C]\!]\vec{k}$
$[\![[\,]]\!]\vec{k} \triangleq \mathbf{0}$

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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

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

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 $[\![S \cdot (a, h_H, [])]\!]k = [\![S \cdot (a, h_H)]\!]k$.

Lemma B.34
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 $H \asymp H'$, such that $H + H' \models S' \cdot (a', h', C') : A$ and that, for all channel k with $k \notin (dom_2(H + H') \cup L)$, we have $[\![\varnothing]\!], S, k{:}K[[\![A]\!]] \vdash [\![S \cdot (a, h_H, C)]\!]k \approx [\![S' \cdot (a', h'_{H+H'}, C')]\!]k$.

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.35
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 $k \notin dom_2(H + H') \cup L$, $[\![\varnothing]\!], S, k{:}K[[\![A]\!]] \vdash [\![S \cdot (a, h_H)]\!]k \approx [\![S \cdot (p', h'_{H+H'})]\!]k$.

Proof
Assume $H \models S \cdot (a, h) : A$ and $S \cdot (a, h) \Downarrow (p', h')$. By Theorem B.33, we have $S \cdot (a, h, []) \rightarrow^* S \cdot (p', h', [])$. By rule (Control Good Empty) and (Small Config Good), we have $H \models S \cdot (a, h, []) : A$. By Lemma B.34, there is a heap typing H', with $H \asymp H'$, such that $H + H' \models S \cdot (p', h', []) : A$ and that, for all channel k with $k \notin (dom_2(H + H') \cup L)$, we have $[\![\varnothing]\!], S, k{:}K[[\![A]\!]] \vdash [\![(H, S, a, h, [])]\!]k \approx [\![(H + H', S, p', h', [])]\!]k$.

Since for all big-step configuration $S{\cdot}(a, h_H)$ we have $[\![S{\cdot}(a, h_H, [])]\!]k = [\![S{\cdot}(a, h_H)]\!]k$, it follows that $[\![\varnothing]\!], S, k{:}K[[\![A]\!]] \vdash [\![S \cdot (a, h_H)]\!]k \approx [\![(S, H + H', p', h')]\!]k$. □

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.36
Suppose $H \models S \cdot (a, h) : A$ and $k \notin dom_2(H) \cup L$. Let $\{\vec{\rho}_{\text{defunct}}\} = dom(H) - S$. Then $[\![\varnothing]\!], S, k{:}K[[\![A]\!]] \vdash [\![S \cdot (a, h)]\!]k \approx (\nu\vec{\rho}_{\text{defunct}})(\nu[\![ptr(H)]\!])([\![a]\!]k \mid \prod_{\rho \in S}[\![H(\rho)]\!])$.

Proof
Let $\vec{\rho}$ be a sequence of groups, $\rho_1, \ldots, \rho_m$, such that $\{\vec{\rho}\} = S$. Let $\vec{\rho}_{\text{defunct}}$ be a sequence of groups, $\rho'_1, \ldots, \rho'_n$, such that $\{\vec{\rho}_{\text{defunct}}\} = dom(H) - S$. For the sake of

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

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:

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

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:

$$\begin{array}{l} [\![\emptyset]\!], \vec{\rho}, \vec{\rho}_\times, [\![\vec{r}\ at\ \vec{\rho}]\!], [\![\vec{r}_\times\ at\ \vec{\rho}_\times]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k : e \cup \{K\} \\ [\![\emptyset]\!], \vec{\rho}, \vec{\rho}_\times, [\![\vec{r}\ at\ \vec{\rho}]\!], [\![\vec{r}_\times\ at\ \vec{\rho}_\times]\!], k{:}K\,[\![A]\!] \vdash [\![\vec{\rho} \mapsto \vec{r}]\!] : \{\vec{\rho}\} \\ [\![\emptyset]\!], \vec{\rho}, \vec{\rho}_\times, [\![\vec{r}\ at\ \vec{\rho}]\!], [\![\vec{r}_\times\ at\ \vec{\rho}_\times]\!], k{:}K\,[\![A]\!] \vdash [\![\vec{\rho}_\times \mapsto \vec{r}_\times]\!] : \{\vec{\rho}_\times\} \end{array}$$

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

$$\begin{array}{l} [\![\emptyset]\!], \vec{\rho}, k{:}K\,[\![A]\!], \vec{\rho}_\times, [\![\vec{r}\ at\ \vec{\rho}]\!], [\![\vec{r}_\times\ at\ \vec{\rho}_\times]\!] \vdash P : \{\vec{\rho}, K\} \\ [\![\emptyset]\!], \vec{\rho}, k{:}K\,[\![A]\!], \vec{\rho}_\times, [\![\vec{r}\ at\ \vec{\rho}]\!], [\![\vec{r}_\times\ at\ \vec{\rho}_\times]\!] \vdash [\![\vec{\rho}_\times \mapsto \vec{r}_\times]\!] : \{\vec{\rho}_\times\} \end{array}$$

By Theorem B.30 several times, we get:

$$\begin{array}{l} [\![\emptyset]\!], S, k{:}K\,[\![A]\!] \vdash (\nu\vec{\rho}_\times)(\nu\,[\![\vec{r}\ at\ \vec{\rho}]\!])(\nu\,[\![\vec{r}_\times\ at\ \vec{\rho}_\times]\!]) \\ \quad (P \mid [\![\vec{\rho}_\times \mapsto \vec{r}_\times]\!]) \approx (\nu\vec{\rho}_\times)(\nu\,[\![\vec{r}\ at\ \vec{\rho}]\!])(\nu\,[\![\vec{r}_\times\ at\ \vec{\rho}_\times]\!])P \end{array}$$

But this is:

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

□

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.37, that the encoding of an expression obtained by substituting an allocation $v\ at\ \rho$ for a variable x in an expression b is equivalent to the process obtained by substituting for x in $[\![b]\!]$ a private link to a replicated copy of the process $[\![v]\!]$. This property is used in the proof of Theorem 5.2. More precisely, it is needed to show that the encoding of β-equivalent terms are equivalent processes.

Lemma B.37 is the only result of this article that directly depends on the locality restriction imposed on the π-calculus.

Lemma B.37
Consider two expressions a and b such that a is an allocation, $v\ at\ \rho$, with $E \vdash a :^{\{\rho\}} A$ and $E, x{:}A \vdash b :^e B$. If $p \notin fn(v) \cup fn(b)$ and $k \notin dom(E) \cup L$ then:

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Proof of Lemma 5.1 *If $E \vdash a_1 \leftrightarrow a_2 : A$ then there are e_1, e_2 such that for each $i \in 1..2$, $e_i \subseteq dom(E)$ and $E \vdash a_i :^{e_i} A$.*

Proof
By induction on the derivation of $E \vdash a_1 \leftrightarrow a_2 : A$.

(Eq Refl), (Eq Symm) and (Eq Trans) Trivial.

(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 $a_i \triangleq (\lambda(x{:}A')b_i \ at \ \rho)$ and $e'_i \subseteq e'$ and $E \vdash A$. Take $e_1 = e_2 = \{\rho\}$. By (Type →), since $E \vdash A$, we get that $e_i \subseteq dom(E)$ for each $i \in 1..2$. By (Exp Fun), $E \vdash a_i :^{e_i} A$ for each $i \in 1..2$, as required.

(Eq Fun β) and (Eq Let β) Then $a_1 \triangleq let \ y = (\lambda(x{:}B)b \ at \ \rho) \ in \ y(a)$ and $a_2 \triangleq b\{x{\leftarrow}a\}$ where a is a name or an allocation, and $y \notin fn(a)$, 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 $\rho \in dom(E)$. By (Exp Appl) and (Exp Let), we get that $E \vdash a_1 :^{e_1 \cup \{\rho\}} A$, as required. The case for (Eq Let β) is similar.

(Eq Let) and (Eq Letregion Let) Then $a_1 \triangleq let \ x = a \ in \ b$ and $a_2 \triangleq let \ x = a' \ in \ b'$ where $E \vdash a \leftrightarrow a' : B$ and $E, x : B \vdash b \leftrightarrow b' : A$. By induction hypothesis, there are $e^1_1, e^1_2, e^2_1, e^2_2 \subseteq dom(E)$ such that $E \vdash a :^{e^1_1} B$, $E \vdash a' :^{e^1_2} B$, $E, x : B \vdash b :^{e^2_1} A$ and $E, x : B \vdash b' :^{e^2_2} A$. Take $e_i = e^i_1 \cup e^i_2$ for each $i \in 1..2$. By (Exp Let), we get that $E \vdash a_i :^{e_i} A$ and $e_i \subseteq dom(E)$ for each $i \in 1..2$, as required. The case for (Eq Letregion Let) is similar.

(Eq Let Assoc) Then $a_1 \triangleq let \ x = a \ in \ (let \ y = b \ in \ c)$ and $a_2 \triangleq let \ y = (let \ x = a \ in \ b) \ in \ c$ 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 $x \notin dom(E, y : B)$, we get that $x \notin fn(c)$ and $E, y{:}B, x{:}A \vdash \diamond$. Take $e_1 = e_2 = e_a \cup e_b \cup e_c$. By (Exp Let) and Lemma B.1, 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 are f_1, f_2 such that $f_i \subseteq dom(E, \rho)$ and $E, \rho \vdash b_i :^{f_i} A$ for each $i \in 1..2$. Take $e_i = f_i - \{\rho\}$ 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. □

Proof of Theorem 5.2 *Suppose $E \vdash a \leftrightarrow b : A$ and $k \notin dom(E) \cup L$. Then $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \approx [\![b]\!]k$.*

Proof
By induction on the derivation of $E \vdash a \leftrightarrow b : A$. By Lemma 5.1, there are e_1, e_2 such that $e_i \subseteq dom(E)$ for each $i \in 1..2$ and $E \vdash a :^{e_1} A$ and $E \vdash b :^{e_2} A$. Let $k \notin dom(E) \cup L$. By Theorem 4.1(3), $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k : e_1 \cup \{K\}$ and $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![b]\!]k : e_2 \cup \{K\}$. Hence, $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k, [\![b]\!]k$.

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

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

(Eq Refl), (Eq Symm) 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 $a \stackrel{\Delta}{=} \mathit{let}\ y = (\lambda(x{:}B)b'\ \mathit{at}\ \rho)\ \mathit{in}\ y(a')$ and $b \stackrel{\Delta}{=} b'\{x{\leftarrow}a'\}$ where a' is a name or an allocation, and $y \notin \mathit{fn}(a')$. Hence, $[\![a]\!]k \stackrel{\Delta}{=} (\nu k')\ \mathit{def}\ p(x,k) = [\![b']\!]k\ \mathit{in}\ (\overline{k'}\langle p\rangle \mid k'(y).[\![y(a')]\!]k)$ where k' and p are fresh names. By Lemma A.4, $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \approx \mathit{def}\ p(x,k) = [\![b']\!]k\ \mathit{in}\ [\![p(a')]\!]k$. We have two possible cases depending on the shape of a'.

Assume a' is a name, say q. Hence, $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \approx \mathit{def}\ p(x,k) = [\![b']\!]k\ \mathit{in}\ \overline{p}\langle q,k\rangle$. By Proposition A.6(5):

$$[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \approx \mathit{def}\ p(x,k) = [\![b']\!]k\ \mathit{in}\ [\![b'\{x{\leftarrow}a'\}]\!]k$$

By Proposition A.6(1), since $p \notin \mathit{fn}(b\{x{\leftarrow}a'\})$, we get that:

$$[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \approx [\![b'\{x{\leftarrow}a'\}]\!]k$$

Assume a' is an allocation, say $(\lambda(y)c\ \mathit{at}\ \rho')$. Hence:

$$\begin{aligned}
[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \quad &\approx \quad \mathit{def}\ p(x,k) = [\![b']\!]k\ \mathit{in} \\
&\qquad (\nu k')(\mathit{def}\ q(y,k) = [\![c]\!]k\ \mathit{in} \\
&\qquad\quad (\overline{k'}\langle q\rangle \mid k'(y).\overline{p}\langle y,k\rangle)) \\
&\qquad\qquad \text{(By Lemma A.4)} \\
&\approx \quad \mathit{def}\ p(x,k) = [\![b']\!]k\ \mathit{in} \\
&\qquad (\mathit{def}\ q(y,k) = [\![c]\!]k\ \mathit{in}\ \overline{p}\langle q,k\rangle) \\
&\qquad\qquad \text{(By Proposition A.6(5))} \\
&\approx \quad \mathit{def}\ p(x,k) = [\![b']\!]k\ \mathit{in} \\
&\qquad (\mathit{def}\ q(y,k) = [\![c]\!]k\ \mathit{in}\ [\![b'\{x{\leftarrow}q\}]\!]k) \\
&\qquad\qquad \text{(By Lemma B.37)} \\
&\approx \quad \mathit{def}\ p(x,k) = [\![b']\!]k\ \mathit{in}\ [\![b'\{x{\leftarrow}a'\}]\!]k
\end{aligned}$$

Case (Eq Let β) is similar.

(Eq Let Assoc) Then $a \stackrel{\Delta}{=} \mathit{let}\ x = a'\ \mathit{in}\ (\mathit{let}\ y = b'\ \mathit{in}\ c')$ and $b \stackrel{\Delta}{=} \mathit{let}\ y = (\mathit{let}\ x = a'\ \mathit{in}\ b')\ \mathit{in}\ c'$ 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 $x \notin \mathit{dom}(E, y : B)$, we get that $x \notin \mathit{fn}(c')$. Hence, $[\![a]\!]k \stackrel{\Delta}{=} (\nu k_1)([\![a']\!]k_1 \mid k_1(x).(\nu k_2)([\![b']\!]k_2 \mid k_2(y).[\![c']\!]k))$ and $[\![b]\!]k \equiv (\nu k_1)([\![a']\!]k_1 \mid (\nu k_2)(k_1(x).[\![b']\!]k_2 \mid k_2(y).[\![c']\!]k))$, where k_1, k_2 are two fresh names.

By Lemma A.5(1), $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \approx (\nu k_1)([\![a']\!]k_1 \mid (\nu k_2)k_1(x).([\![b']\!]k_2 \mid k_2(y).[\![c']\!]k))$. By Lemma A.5(3), we get that $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \approx [\![b]\!]k$, as desired.

(Eq Drop), (Eq Swap) and (Eq Letregion Let) In each of this cases we have the following three relations: $\mathit{erase}([\![a]\!]k) = \mathit{erase}([\![b]\!]k)$, and $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k$ and $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![b]\!]k$. By Proposition B.23, $[\![E]\!], k{:}K\,[\![A]\!] \vdash [\![a]\!]k \approx [\![b]\!]k$. □

References

Abadi, M. & Gordon, A. D. (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.)

https://doi.org/10.1017/S0956796801004270 Published online by Cambridge University Press

Aiken, A., Faehndrich, M. & Levien, R. (1995) Better static memory management: Improvements to region-based analysis of higher-order languages. *Proc. PLDI'95*, pp. 174–185. ACM Press.

Banerjee, A., Heintze, N. & Riecke, J. (1999) Region analysis and the polymorphic lambda calculus. *Proc. LICS'99*.

Benton, N. & Kennedy, A. (1999) Monads, effects and transformations. *Proc. HOOTS'99*, Electronic Notes in Theoretical Computer Science, **26**, pp. 1–18. Elsevier.

Birkedal, L., Tofte, M. & Vejlstrup, M. (1996) From region inference to von Neumann machines via region representation inference. *Proc. POPL'96*, pp. 171–183. ACM Press.

Boudol, G. (1992) *Asynchrony and the π-calculus*. Research Report 1702, INRIA.

Calcagno, C. (2001) Stratified operational semantics for safety and correctness of the region calculus. *Proc. POPL'01*, pp. 155–165. ACM Press.

Cardelli, L., Ghelli, G. & Gordon, A. D. (2000a) Ambient groups and mobility types. In *Proc. IFIP TCS 2000: Lecture Notes in Computer Science 1872*, pp. 333–347. Springer-Verlag.

Cardelli, L., Ghelli, G. & Gordon, A. D. (2000b) Group creation and secrecy. In *Proc. CONCUR 2000: Lecture Notes in Computer Science 1877*, pp. 365–379. Springer-Verlag.

Crary, K., Walker, D. & Morrisett, G. (1999) Typed memory management in a calculus of capabilities. *Proc. POPL'99*, pp. 262–275. ACM Press.

Dal Zilio, S. (1999) *A bisimulation for the blue calculus*. Research Report 3664. INRIA.

Dal Zilio, S. & Gordon, A. D. (2000a) Region analysis and a π-calculus with groups. *Proc. MFCS 2000: Lecture Notes in Computer Science 1893*, pp. 1–20. Springer-Verlag.

Dal Zilio, S. & Gordon, A. D. (2000b) *Region analysis and a π-calculus with groups*. Technical Report MSR-TR-2000-57, Microsoft Research.

Flanagan, C. & Abadi, M. (1999) Object types against races. In *Proc. CONCUR'99: Lecture Notes in Computer Science 1664*, pp. 288–303. Springer-Verlag.

Fournet, C. & Gonthier, G. (1996) The reflexive CHAM and the Join-calculus. *Proc. POPL'96*, pp. 372–385. ACM Press.

Gifford, D. K. & Lucassen, J. M. (1986) Integrating functional and imperative programming. *Proc. L&FP'86*, pp. 28–38.

Helsen, S. & Thiemann, P. (2000) Syntactic type soundness for the region calculus. *Proc. HOOTS 2000*, Electronic Notes in Theoretical Computer Science, **41**(3), pp. 1–19. Elsevier.

Honda, K. (1992) *Two bisimilarities for the ν-calculus*. Technical Report 92-002, Department of Computer Science, Keio University.

Hughes, J. & Pareto, L. (1999) Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. *Proc. ICFP'99*, pp. 70–81. ACM Press.

Kobayashi, N. (1998) A partially deadlock-free typed process calculus. *ACM Trans. Programming Languages and Systems*, **20**, 436–482.

Launchbury, J. & Peyton Jones, S. (1995) State in Haskell. *Lisp & Symbolic Computation*, **8**(4), 293–341.

Merro, M. & Sangiorgi, D. (1998) On asynchrony in name-passing calculi. *Proc. ICALP'98: Lecture Notes in Computer Science 1443*, pp. 856–867. Springer-Verlag.

Milner, R. (1999) *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press.

Milner, R. & Sangiorgi, D. (1992) Barbed bisimulation. *Proc. ICALP'92: Lecture Notes in Computer Science 623*, pp. 685–695. Springer-Verlag.

Milner, R., Parrow, J. & Walker, D. (1992) A calculus of mobile processes, parts I and II. *Information & Computation*, **100**, 1–77.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (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. & Palumbo, F. (1999) Monadic encapsulation of effects: a revised approach. *Proc. HOOTS'99*, pp. 119–136. Electronic Notes in Theoretical Computer Science, **26**. Elsevier.

Nielson, F. & Nielson, H. R. (1993) From CML to process algebras. *Proc. CONCUR'93: Lecture Notes in Computer Science 715*, pp. 493–508. Springer-Verlag.

Nielson, H. R. & Nielson, F. (1994) Higher-order concurrent programs with finite communication topology. *Proc. POPL'94*, pp. 84–97. ACM Press.

Pierce, B. C. & Sangiorgi, D. (1996) Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, **6**(5), 409–454.

Pierce, B. C. & Sangiorgi, D. (1997) Behavioral equivalence in the polymorphic pi-calculus. *Proc. POPL'97*, pp. 242–255. ACM Press. (Also appears as INRIA-Sophia Antipolis Rapport de Recherche 3042 and as Indiana University Computer Science Technical Report 468.)

Pierce, B. C. & Turner, D. N. (2000) Pict: A programming language based on the pi-calculus. In: G. Plotkin, C. Stirling and M. Tofte (editors), *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 455–494. MIT Press.

Plotkin, G. D. (1975) Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, **1**, 125–159.

Sangiorgi, D. & Walker, D. (2001) *The pi-calculus: A theory of mobile processes*. Cambridge University Press.

Semmelroth, M. & Sabry, A. (1999) Monadic encapsulation in ML. *Proc. ICFP'99*, pp. 8–17. ACM Press.

Talpin, J.-P. (1993) *Aspects théoretiques et pratiques de l'inférence de types et d'effets*. Thèse de doctorat, Université Paris VI and Ecole des Mines de Paris.

Talpin, J.-P. (2001) A simplified account of region inference. Research Report 4104, INRIA, Rennes.

Talpin, J.-P. & Jouvelot, P. (1992) Polymorphic type, region and effect inference. *J. Functional Programming*, **2**(3), 245–271.

Talpin, J.-P. (1993) *Aspects théoriques et pratiques de l'inférence de types et d'effets*. Thèse de Doctorat, Université Paris VI and Ecole des Mines de Paris, May 1993.

Tofte, M. & Talpin, J.-P. (1997) Region-based memory management. *Information & Computation*, **132**(2), 109–176.

Turner, D. N. (1995) *The polymorphic pi-calculus: theory and implementation*. PhD thesis, University of Edinburgh.

Wadler, P. (1998) The marriage of effects and monads. *Proc. ICFP'98*, pp. 63–74. ACM Press.

Walker, D. (1995) Objects in the pi-calculus. *Information & Computation*, **116**(2), 253–271.

Yoshida, N. (1996) Graph types for monadic mobile processes. *Foundations of Software Technology and Theoretical Computer Science (TSTTCS'96): Lecture Notes in Computer Science 1180*, pp. 371–386. Springer-Verlag.