To appear in the *Proceedings of MFCS 2000: the 25th International Symposium on Mathematical Foundations of Computer Science, August 28–September 1, 2000, Bratislava, Slovak Republic*, to be published by Springer in the series *Lecture Notes in Computer Science*.

Region Analysis and a π-Calculus with Groups

Silvano Dal Zilio and Andrew D. Gordon

Microsoft Research

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.

1 Motivation

This paper reports a new proof of correctness of region-based memory management [26], based on a new garbage collection principle for the π-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 [4]. The basic idea of the region calculus is to partition heap memory into a stack of regions. Each boxed value (that is, a heap-allocated value such as a closure or a cons cell) is annotated with the particular region into which it is stored. The construct *letregion* ρ *in* b manages the allocation and de-allocation of regions. It means: "Allocate a fresh, empty region, denoted by the region variable ρ; evaluate the expression b; de-allocate ρ." A type and effect system for the region calculus guarantees the safety of de-allocating the defunct region as the last step of *letregion*. The allocation and de-allocation of regions obeys a stack discipline determined by the nesting of the *letregion* constructs. A region inference algorithm compiles ML to the region calculus by computing suitable region annotations for boxed values, and inserting *letregion* constructs as necessary. In practice, space leaks, where a particular region grows without bound, are a problem. Still, they can practically always be detected by profiling and eliminated by simple modifications. The ML Kit efficiently executes an impressive range of benchmarks without a garbage collector and without space leaks. Region-based memory management facilitates

interoperability with languages like C that have no garbage collector and helps enable realtime applications of functional programming.

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

The region calculus is a strikingly simple example of a language with type generativity. A language has type generativity when type equivalence is by name (that is, when types with different names but the same structure are not equivalent), and when type names can be generated at run-time. A prominent example is the core of Standard ML [17], whose *datatype* construct effectively generates a fresh algebraic type each time it is evaluated. (The ML module system also admits type generativity, but at link-time rather than run-time.) The region calculus has type generativity because the type of a boxed value includes the name of the region where it lives, and region names are dynamically generated by *letregion*. The semantics of Standard ML accounts operationally for type generativity by inventing a fresh type name on each elaboration of *datatype*. Various researchers have sought more abstract accounts of type generativity [13, 21].

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 [15] 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 [22, 25, 28], for the design of concurrent programming languages [9, 20], and for the study of security protocols [1], 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 [19], names may not occur in types.

We solve the problem of modelling region names by defining a typed π-calculus equipped with name groups and a new-group construct [5]. 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 [22]. 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 $[\![\mathit{letregion}\ \rho\ \mathit{in}\ b]\!]k = (\nu\rho)[\![b]\!]k$. Then the rather subtle correctness property of de-allocation of defunct regions turns out to be a simple instance of a new abstract principle expressed in the π-calculus. Hence, an advantage of our π-calculus proof is that it is conceptually simpler than a direct proof.
- Second, the semantics provides a more abstract, equational account of type generativity in the region calculus than the standard operational semantics.

The specific technical results of the paper are:

- A simple proof of type soundness of the region calculus (Theorem 1).
- A new semantics of the region calculus in terms of the π-calculus with groups. The translation preserves types and effects (Theorem 2) and operational behaviour (Theorem 3).
- A new garbage collection principle for the π-calculus (Theorem 4) whose corollary (Theorem 5) justifies de-allocation of defunct regions in the region calculus.

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 concludes. Omitted proofs may be found in a long version of this paper [8].

2 A λ-Calculus with Regions

To focus on the encoding of *letregion* with the new-group construct, we work with a simplified version of the region calculus of Tofte and Talpin [26]. Our calculus omits the recursive functions, type polymorphism, and region polymorphism present in Tofte and Talpin's calculus. The long version of this paper includes an extension of our results to a region calculus with recursive functions, finite lists, and region polymorphism. To encode these features, we need to extend our π-calculus with recursive types and group 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
x	name
$v\ at\ \rho$	allocation of v at ρ
$x(y)$	application
$let\ x = a\ in\ b$	sequencing
$letregion\ \rho\ in\ b$	region allocation, de-allocation
$u, v ::=$	boxed value
$\lambda(x{:}A)b$	function

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

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

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

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

2.2 Dynamic Semantics

Like Tofte and Talpin, we formalize the intuitive semantics via a conventional structural operational semantics. A heap, h, is a map from region names to regions, and a region, r, is a map from pointers (names) to boxed values (function closures). In Tofte and Talpin's semantics, defunct regions are erased from the heap when they are de-allocated. In our semantics, the heap consists of both live regions and defunct regions. Our semantics maintains a set S containing the region names for the live regions. This is the main difference between the two semantics. Side-conditions on the evaluation rules guarantee that only the live regions in S are accessed during evaluation. Retaining the defunct regions simplifies the proof of subject reduction. Semmelroth and Sabry [23] 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 $r - p$ to be $p_i \mapsto v_i{}^{\,i\in(1..n)-\{j\}}$ if $p = p_j$ for some $j \in 1..n$, and otherwise to be simply r. Then we define the notation $r + (p \mapsto v)$ to mean $(r - p), p \mapsto v$.

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

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

Judgments:

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

Evaluation Rules:

(Eval Var)

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

(Eval Alloc)

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

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

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

Recall the example expression ex_1 from the previous section. Consider an initial heap $h = \rho \mapsto \varnothing$ 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 $h' = \rho \mapsto (g \mapsto \lambda(y{:}\mathit{Lit})f(y)), \rho' \mapsto (f \mapsto \lambda(x{:}\mathit{Lit})x)$. Since $\rho \in S$ but $\rho' \notin S$, ρ is live but ρ' is defunct.

2.3 Static Semantics

The static semantics of the region calculus is a simple type and effect system [10, 24, 27]. 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, $\mathit{put}(\rho)$, and invocations, $\mathit{get}(\rho)$, 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 $(A \xrightarrow{e} B)\ \mathit{at}\ \rho$, 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
$\quad \mathit{Lit}$	$\quad$ type of literals
$\quad (A \xrightarrow{e} B)\ \mathit{at}\ \rho$	$\quad$ type of functions stored in ρ
$E ::=$	environment
$\quad \varnothing$	$\quad$ empty environment
$\quad E, \rho$	$\quad$ entry for a region ρ
$\quad E, x{:}A$	$\quad$ entry for a name x

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

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

Type and Effect Judgments:

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

Type and Effect Rules:

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

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

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

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

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

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

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

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

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

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

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

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:

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

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, 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:

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

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

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, $dom_2(H) \cap dom_2(H') = \varnothing$. 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 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 run-time errors of allocation or invocation of a closure in a defunct region.

Proposition 1.

(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 1 and Proposition 1 we may conclude that such run-time errors never arise in any intermediate configuration reachable from an initial well-typed configuration. Implicitly, this amounts to asserting the safety of

region-based memory management, that defunct regions make no difference to the behaviour of a well-typed configuration. Our π-calculus semantics of regions makes this explicit: we show equationally that direct deletion of defunct regions makes no difference to the semantics of a configuration.

3 A π-Calculus with Groups

In this section, we define a typed π-calculus with groups. In the next, we explain a semantics of our region calculus in this π-calculus. Exactly as in the ambient calculus with groups [5], 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 [6], not concerned with the region calculus, we consider a simpler version of this π-calculus, with groups but without an effect system, and show that new-group helps keep names secret, in a certain formal sense.

3.1 Syntax

The following table gives the syntax of processes, P. The syntax depends on a set of atomic names, x, y, z, p, q, and a set of groups, G, H. For convenience, we assume that the sets of names and groups are identical to the sets of names and region names, respectively, of the region calculus. We impose a standard constraint [9, 14], usually known as locality, that received names may be used for output but not for input. This constraint is actually unnecessary for any of the results of this paper, but is needed for proofs of additional results in the long version [8]. 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 [15].

Expressions and Processes:

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

The set $inp(P)$ contains each name x such that an input process $x(y_1{:}T_1, \ldots, y_n{:}T_n).P'$ occurs as a subprocess of P, with x not bound. We explain the types T below. 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. 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.

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

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

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

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

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

The following is a more accurate encoding:

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

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

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

3.2 Dynamic Semantics

We formalize the semantics of our π-calculus using standard techniques. A reduction relation, $P \to Q$, 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)
$P \equiv Q \Rightarrow (\nu x{:}T)P \equiv (\nu x{:}T)Q$	(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.

Erasing type annotations and group restrictions:

$erase((\nu G)P) \triangleq erase(P)$
$erase((\nu x{:}T)P) \triangleq (\nu x)erase(P)$
$erase(\mathbf{0}) \triangleq \mathbf{0}$
$erase(P \mid Q) \triangleq erase(P) \mid erase(Q)$
$erase(!P) \triangleq !erase(P)$

$erase(x(y_1{:}T_1, \ldots, y_n{:}T_n).P) \triangleq x(y_1, \ldots, y_n).erase(P)$
$erase(\overline{x}\langle y_1, \ldots, y_n\rangle) \triangleq \overline{x}\langle y_1, \ldots, y_n\rangle$

Proposition 2 (Erasure). *For all typed processes P and Q, if $P \to Q$ then $erase(P) \to erase(Q)$ and if $erase(P) \to R$ then there is a typed process Q such that $P \to Q$ and $R \equiv erase(Q)$.*

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 $\mathbf{H} = \varnothing$, 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 5.

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 G channels. 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 $fg(G[T_1,\ldots,T_n]\backslash\mathbf{H}) \triangleq \{G\} \cup fg(T_1) \cup \cdots \cup fg(T_n) \cup \mathbf{H}$.

Syntax of Types and Environments, Typing Judgments:

$\mathbf{G},\mathbf{H} ::= \{G_1,\ldots,G_k\}$	finite set of name groups
$T ::= G[T_1,\ldots,T_n]\backslash\mathbf{H}$	type of channel in group G with hidden effect $\mathbf{H}$
$E ::= \varnothing \mid E,G \mid E,x{:}T$	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 $\varnothing$) $\dfrac{}{\varnothing \vdash \diamond}$ (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) $\dfrac{E \vdash \diamond}{E \vdash \mathbf{0} : \varnothing}$ (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.

In the special case when the hidden effect $\mathbf{H}$ is $\varnothing$, (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 $\mathbf{G}$ of the prefixed process P to the input process $x(y_1{:}T_1, \ldots, y_n{:}T_n).P$. The effect $\mathbf{G}$ 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 $\mathbf{H}$. 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; 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. *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 $\overline{x}$.

Exhibition of a barb: $P \downarrow \beta$

$$\frac{}{x(y_1{:}T_1, \ldots, y_n{:}T_n).P \downarrow x} \qquad \frac{}{\overline{x}\langle y_1, \ldots, y_n\rangle \downarrow \overline{x}}$$

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

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

Proposition 4. *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* $G \in \mathbf{H}$.

4 Encoding Regions as Groups

This section interprets the region calculus in terms of our π-calculus. 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 $(\nu k')(\llbracket a \rrbracket k' \mid k'(x).\llbracket b \rrbracket k)$. This process invents a fresh, intermediate continuation k'. The process $\llbracket a \rrbracket k'$ evaluates a returning a result on k'. The process $k'(x).\llbracket b \rrbracket k$ blocks until the result x is returned on k', then evaluates b, returning its result on k.

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

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

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

Translation rules:

$\llbracket Lit \rrbracket \triangleq Lit[]$

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

$\llbracket \varnothing \rrbracket \triangleq K, Lit, \ell_1{:}Lit[], \ldots, \ell_n{:}Lit[]$

$\llbracket E, \rho \rrbracket \triangleq \llbracket E \rrbracket, \rho$

$\llbracket E, x{:}A \rrbracket \triangleq \llbracket E \rrbracket, x{:}\llbracket A \rrbracket$

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

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

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

Theorem 2 (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. First, we take our process equivalence to be barbed congruence [16], a standard operational equivalence for the π-calculus. We use a typed version of (weak) barbed congruence, as defined by Pierce and Sangiorgi [19]; the long version of this paper contains the detailed definition. Then, our theorem states that if one region calculus configuration evaluates to another, their π-calculus interpretations are equivalent. In the following, let $E \vdash P$ mean there is an effect $\mathbf{G}$ such that $E \vdash P : \mathbf{G}$.

Typed process equivalence: $E \vdash P \approx Q$

For all typed processes P and Q, let $E \vdash P \approx Q$ mean that $E \vdash P$ and $E \vdash Q$ and that P and Q are barbed congruent.

Theorem 3 (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 3 we obtain the following equations (in which we abbreviate environments and types for the sake of clarity):

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

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

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

Suppose there are processes P and R such 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. 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. *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. It asserts that deleting defunct regions makes no difference to the behaviour of a configuration:

Theorem 5. *Suppose* $H \models S \cdot (a,h) : A$ *and* $k \notin dom_2(H) \cup L$. *Let* $\{\boldsymbol{\rho}_{\mathrm{defunct}}\} = dom(H) - S$. *Then we can derive the equation* $[\![\varnothing]\!], S, k{:}K[\![A]\!] \vdash [\![S \cdot (a,h)]\!]k \approx (\nu\boldsymbol{\rho}_{\mathrm{defunct}})(\nu[\![ptr(H)]\!])([\![a]\!]k \mid \prod_{\rho\in S}[\![H(\rho)]\!])$.

5 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 in the setting of the ambient calculus [5]. We showed that the rather subtle correctness of memory de-allocation in the region calculus is an instance of Theorem 4, a new garbage collection principle for the π-calculus. The translation is an example of how the new-group construct accounts for the type generativity introduced by *letregion*, just as the standard new-name construct of the π-calculus accounts for dynamic generation of values.

Banerjee, Heintze, and Riecke [3] give an alternative proof of the soundness of region-based memory management. Theirs is obtained by interpreting the region calculus in a polymorphic λ-calculus equipped with a new binary type constructor # that behaves like a union or intersection type. Their techniques are those of denotational semantics, completely different from the operational techniques of this paper. The formal connections between the two approaches are not obvious but would be intriguing to investigate. A possible advantage of our semantics in the π-calculus is that it could easily be extended to interpret a region calculus with concurrency, but that remains future work. Another line of future work is to consider the semantics of other region calculi [2, 7, 11] in terms of the π-calculus. Finally, various researchers [18, 23] have noted a connection between the monadic encapsulation of state in Haskell [12] 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 Cédric Fournet, Giorgio Ghelli and Mads Tofte. Luca Cardelli, Tony Hoare, and Andy Moran commented on a draft.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999. An extended version appears as Research Report 149, Digital Equipment Corporation Systems Research Center, January 1998.
2. A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings PLDI'95*, pages 174–185, 1995.
3. A. Banerjee, N. Heintze, and J. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings LICS'99*, 1999.
4. L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings POPL'96*, pages 171–183. 1996.
5. L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *Proceedings TCS2000*, Lecture Notes in Computer Science. Springer, 2000. To appear.
6. L. Cardelli, G. Ghelli, and A. D. Gordon. Group creation and secrecy. In *Proceedings Concur'00*, Lecture Notes in Computer Science. Springer, 2000. To appear.
7. K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings POPL'99*, pages 262–275, 1999.

8. S. Dal Zilio and A. D. Gordon. Region analysis and a π-calculus with groups. Technical Report MSR–TR–2000–57, Microsoft Research, 2000.
9. C. Fournet and G. Gonthier. The reflexive CHAM and the Join-calculus. In *Proceedings POPL'96*, pages 372–385, 1996.
10. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings L&FP'86*, pages 28–38, 1986.
11. J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings ICFP'99*, pages 70–81, 1999.
12. J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.
13. X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.
14. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer, 1998.
15. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
16. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings ICALP'92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992.
17. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
18. E. Moggi and F. Palumbo. Monadic encapsulation of effects: a revised approach. In *Proceedings HOOTS99*, volume 26 of *Electronic Notes in Theoretical Computer Science*, pages 119–136. Elsevier, 1999.
19. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
20. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, editors, MIT Press, 2000.
21. C. V. Russo. Standard ML type generativity as existential quantification. Technical Report ECS–LFCS–96–344, LFCS, University of Edinburgh, 1996.
22. D. Sangiorgi. Interpreting functions as π-calculus processes: a tutorial. Technical Report 3470, INRIA, 1998. Draft chapter to appear in *The pi-calculus: a theory of mobile processes*, D. Sangiorgi and W. Walker, Cambridge University Press, 2000.
23. M. Semmelroth and A. Sabry. Monadic encapsulation in ML. In *Proceedings ICFP'99*, pages 8–17, 1999.
24. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
25. C. J. Taylor. *Formalising and Reasoning about Fudgets.* PhD thesis, University of Nottingham, 1998. Available as Technical Report NOTTCS–TR–98–4.
26. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
27. P. Wadler. The marriage of effects and monads. In *Proceedings ICFP'98*, pages 63–74, 1998.
28. D. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.