Functional Translation of a Calculus of Capabilities

Arthur Charguéraud
INRIA
arthur.chargueraud@inria.fr

François Pottier
INRIA
francois.pottier@inria.fr

Abstract

Reasoning about imperative programs requires the ability to track aliasing and ownership properties. We present a type system that provides this ability, by using regions, capabilities, and singleton types. It is designed for a high-level calculus with higher-order functions, algebraic data structures, and references (mutable memory cells). The type system has polymorphism, yet does not require a value restriction, because capabilities act as explicit store typings.

We exhibit a type-directed, type-preserving, and meaning-preserving translation of this imperative calculus into a pure calculus. Like the monadic translation, this is a store-passing translation. Here, however, the store is partitioned into multiple fragments, which are threaded through a computation only if they are relevant to it. Furthermore, the decomposition of the store into fragments can evolve dynamically to reflect ownership transfers.

The translation offers deep insight about the inner workings and soundness of the type system. If coupled with a semantic model of its target calculus, it leads to a semantic model of its imperative source calculus. Furthermore, it provides a foundation for our long-term objective of designing a system for specifying and certifying imperative programs with dynamic memory allocation.

1. Introduction

Reasoning about imperative programs in the presence of dynamic memory allocation is a challenging task. The existence of aliasing means that an update to a memory block by one principal can affect other principals with which the address of the block is shared, possibly violating their invariants, if aliasing was unintended. Thus, a correctness argument for an imperative program must deal, in one way or another, with aliasing (which object, or group of objects, might a certain pointer denote?) and with ownership (who holds the right to access a certain object, or group of objects?).

There are many ways of attacking this problem (see our discussion of related work in §8). We are interested in a line of work that uses type-theoretic machinery, including regions, capabilities, and singleton types, in order to control aliasing and ownership. A few landmark papers in this area include Crary, Walker, and Morrisett's Calculus of Capabilities [7], Smith, Walker, and Morrisett's Alias Types [19, 22], and Fähndrich and DeLine's Adoption and Focus [12].

Our long-term research project is to design a system (say, a Hoare logic) for proving properties of pointer programs on top of such a capability-based type system. The idea is to express and check assertions about aliasing and ownership at the level of the type system, so that, at the level of the Hoare logic, a view of the store as a collection of separate regions becomes available at no cost. Our approach is closely related to Separation Logic [18]. However, instead of expressing assertions about aliasing and ownership within the logic, we are interested in using more basic machinery—a type system—for this purpose.

In this paper, we present a type system that is designed for a standard, high-level programming language and that combines the key features of the systems cited above. It extends F_μ (that is, System F with recursive types), and permits the co-existence of non-linear values and linear capabilities. Note that, for the moment, we are not interested in type inference or in surface syntax.

On top of this type system, we define a type-directed translation that transforms an imperative program into a purely functional one. In the translation, a capability becomes either a finite map (which encodes a region of the store) or an individual value (which encodes a single, unaliased object). Thus, capabilities, which in the source program are purely type-theoretic entities, are translated to runtime values. The translation is semantics-preserving, and produces well-typed programs in F_μ.

Just like the standard monadic translation [13, 21], this translation is store-passing. It is, however, much more fine-grained than the monadic translation: instead of a single, monolithic store, it exploits multiple store fragments. This has a double benefit: first, separation between regions is made syntactically explicit; second, homogeneous store fragments can be type-checked in F_μ, whereas a monolithic, heterogeneous store would require a more complex type system, equipped perhaps with dependent types.

The value of such a translation is two-fold.

First, we claim that it provides a deep justification and intuition for the soundness of the type system. The design of the system is subtle: it can be hard, even for an expert, to grasp why it all makes sense. We find that, by explaining the type system in terms of a pure λ-calculus, the translation helps expose the intuition behind it. In fact, the soundness of the type system and the soundness of the translation are proved together, in a single statement (§7). When coupled with a semantic model of its target calculus, the translation leads to a semantic model of its imperative source calculus. Thus, the translation is, in a sense, a semantic interpretation. In this sense, our translation serves the same purpose as O'Hearn and Reynolds's translation of Idealized Algol into the polymorphic linear λ-calculus [15].

Second, such a translation provides a foundation for our long-term objective of designing a system for specifying and certifying imperative programs with dynamic memory allocation. The techniques available today for reasoning about purely functional programs can, in principle, be applied to the translated programs. The details of this process are left to future work.

The paper begins with an informal overview of the type system and of the translation (§2), followed with a couple of examples (§3).

Definitions of the untyped source and target calculi follow (§4). The type system and type-directed translation are defined (§5, §6) and proven sound (§7). (A complete proof of soundness appears in the online addendum [1].) The paper ends with discussions of related work (§8) and future work (§9).

2. Overview

2.1 Regions and capabilities

A *region* denotes a set of values. Unlike in previous work, these values are not necessarily memory locations. We distinguish between *singleton regions* σ, which have exactly one inhabitant, and *group regions* ρ, which have an arbitrary number of inhabitants. A value that inhabits region α has type $[\alpha]$ (pronounced: "at α"). A type of the form $[\sigma]$ has exactly one inhabitant: it is a singleton type. A type of the form $[\rho]$ can have zero, one, or more inhabitants.

A *capability* over a region α is a static token of the form $\{\alpha : \theta\}$. Such a capability serves two roles. First, it witnesses the ownership of region α, that is, it represents an exclusive right to access and update the inhabitants of this region. Second, it carries a *memory type* θ, which describes the actual structure of the inhabitants. Indeed, the type $[\alpha]$, alone, does not contain this information. For instance, the capability $\{\rho : \mathrm{ref\,int}\}$ describes and controls a group region whose inhabitants are pointers to integer cells.

Because capabilities represent exclusive ownership, they are linear: they are never duplicated. (By "linear", we mean "non-duplicable", that is, "affine". Discarding a capability is permitted.) Memory types θ, which occur within capabilities and also represent ownership, are linear as well. *Value types* τ, on the other hand, are non-linear. Because the system imposes no restriction on the number of uses of a value, values can be duplicated by β-reduction, and must receive non-linear types. In our system, the value types form a subset of the memory types.

A type of the form $[\alpha]$ is a value type. As a result, the values that inhabit region α can be duplicated at will, while the linear capability $\{\alpha : \theta\}$, which controls access to this region, remains unique. Thanks to this distinction between non-linear values and linear capabilities, the system provides the same degree of control as offered by traditional linear type systems, yet provides greater flexibility, by allowing sharing and multiple uses of values. This distinction appears in earlier work [7, 19, 2].

Compound capabilities are built out of atomic capabilities via conjunction: the composite capability $C_1 * C_2$ controls two store fragments, respectively described by the capabilities C_1 and C_2. Because capabilities are linear, this is naturally a separating conjunction, in the terminology of Separation Logic [18].

Regions and capabilities encode definite non-aliasing information. For instance, if the capability $\{\rho_1 : \mathrm{ref\,int}\} * \{\rho_2 : \mathrm{ref\,int}\}$ is available, then the regions ρ_1 and ρ_2 must be distinct. This implies that the sets of their inhabitants are disjoint: no memory location can simultaneously have type $[\rho_1]$ and type $[\rho_2]$. As another example, consider the capability $\{\sigma : \mathrm{ref\,int} \times \mathrm{ref\,int}\}$. Because the memory type "ref int × ref int" is interpreted linearly, this capability asserts that the unique inhabitant of region σ is a pair of pointers to two distinct integer cells. In contrast, a pair of possibly aliased integer references would be described by a value type $[\rho] \times [\rho]$, together with the group capability $\{\rho : \mathrm{ref\,int}\}$.

2.2 Functions and references

A function may accept not only an argument (a value), but also a (possibly compound) capability. Similarly, a function returns not only a value, but also a capability. For this reason, function types take the form $\chi_1 \to \chi_2$, where *computation types* χ include the productions: $\chi ::= \tau \mid \chi * C$. (The $*$ connective is overloaded.) Furthermore, if a function creates fresh regions, then the names of these regions must be quantified in the return type of the function: for this reason, computation types also include the production $\chi ::= \exists\alpha.\chi$.

An example of a function whose return type is a non-trivial computation type is the primitive operation "ref". The semantics of "$(\mathrm{ref}\, v)$" is to allocate a memory cell at a fresh location l, initialize it with v, and return l. The axiom schema for "ref" is:

$$\mathrm{ref} \quad : \quad \tau \to \exists\sigma.[\sigma] * \{\sigma : \mathrm{ref}\,\tau\}$$

This means that "ref" accepts a value of type τ and returns: (1) a singleton region σ, (2) the inhabitant l of region σ, and (3) the capability $\{\sigma : \mathrm{ref}\,\tau\}$, which controls this region, and indicates that its inhabitant l is the location of a cell that currently holds a value of type τ.

The primitive operations "get" and "set", which read and write a reference cell, have the following axiom schemata:

$$\begin{array}{lcl} \mathrm{get} & : & [\sigma] * \{\sigma : \mathrm{ref}\,\tau\} \to \tau * \{\sigma : \mathrm{ref}\,\tau\} \\ \mathrm{set} & : & ([\sigma] \times \tau_2) * \{\sigma : \mathrm{ref}\,\tau_1\} \to \mathrm{unit} * \{\sigma : \mathrm{ref}\,\tau_2\} \end{array}$$

The "get" operation accepts a memory location l, of type $[\sigma]$, and a capability $\{\sigma : \mathrm{ref}\,\tau\}$. This indicates that, in order to dereference a pointer to a cell, one must provide a capability over that cell. The "get" operation returns the contents of the cell, which has type τ, together with an unchanged capability. The "set" operation supports *strong update:* the type of the contents of the cell is allowed to change from τ_1 to τ_2. In an ordinary typed programming language, such as ML, the need for strong update is seldom perceived. Here, strong update is not just useful (e.g., to enable delayed initialization): it is made necessary by the fact that types are so fine-grained. For instance, without strong update, a reference of type ref $[\sigma]$ would effectively be immutable. Indeed, $[\sigma]$ is a singleton type, so a type-preserving update cannot change the content of such a reference.

In type and effect systems, function types are annotated with an *effect:* a set of regions that the function potentially accesses, for reading or writing. One can view an effect as a particular pattern of use of a capability: an effect is a capability that is required and returned. To reflect this, we define $\chi_1 \to_C \chi_2$ as sugar for $\chi_1 * C \to \chi_2 * C$. In general, capabilities are more general than effects, as they also allow describing functions that allocate, destroy, or re-organize memory. However, the effect notation, when applicable, is quite convenient. For instance, the types of "get" and "set" (in the restricted case of a weak update, for the latter) can be written as follows:

$$\begin{array}{lcl} \mathrm{get} & : & [\sigma] \to_{\{\sigma:\mathrm{ref}\,\tau\}} \tau \\ \mathrm{set} & : & [\sigma] \times \tau \to_{\{\sigma:\mathrm{ref}\,\tau\}} \mathrm{unit} \end{array}$$

2.3 Ownership transfers

Reading a memory cell duplicates a value, while writing a memory cell discards a value. For this reason, the operations "ref", "get" and "set" are restricted to references whose contents are non-linear. This restriction, which also appears in earlier work [12], is enforced through the use of capabilities of the form $\{\sigma : \mathrm{ref}\,\tau\}$, where τ is a value type, as opposed to $\{\sigma : \mathrm{ref}\,\theta\}$, where θ is a memory type. The latter form is more general. In fact, even though this form is not a suitable argument to "get" or "set", it is legal: for instance, the capability $\{\sigma : \mathrm{ref}\,(\mathrm{ref\,int})\}$ describes and controls a reference to a (unique) reference to an integer. So, how does one construct or exploit a capability of the form $\{\sigma : \mathrm{ref}\,\theta\}$?

Our answer is to offer a mechanism for splitting the capability $\{\sigma : \mathrm{ref}\,\theta\}$, which controls both the reference cell denoted by σ and the contents of that cell, into two separate capabilities. In order to do so, we introduce a fresh singleton region σ_1, which serves as a name for the contents of the cell. The capability $\{\sigma : \mathrm{ref}\,\theta\}$ is then converted into the conjunction $\{\sigma : \mathrm{ref}\,[\sigma_1]\} * \{\sigma_1 : \theta\}$. The

first conjunct controls just the cell denoted by σ, and tells that its contents is the unique inhabitant of region σ_1. The second conjunct controls σ_1, and tells that its inhabitant has type θ. This splitting process is reversible. It is described by a symmetric subtyping axiom (FOCUS-REF, Figure 13):

$$\{\sigma : \mathrm{ref}\,\theta\} \equiv \exists\sigma_1.\{\sigma : \mathrm{ref}\,[\sigma_1]\} * \{\sigma_1 : \theta\}$$

An illustration is found in Figure 14. After splitting, because the type $[\sigma_1]$ is non-linear, the capability $\{\sigma : \mathrm{ref}\,[\sigma_1]\}$ can be used to read and write the reference as many times as desired. In particular, reading the reference duplicates its contents, at type $[\sigma_1]$, but does not duplicate the capability $\{\sigma_1 : \theta\}$, which remains unique.

There is another way in which the operations "ref", "get", and "set" are restricted: they act only on singleton regions σ, as opposed to group regions ρ. So, how does one allocate or access a cell within a group region? To address this question, we rely on the *adoption* and *focus* mechanisms proposed by Fähndrich and DeLine [12].

Empty group regions can be created at any time. *Adoption* is used to populate a group region: adoption dissolves a singleton region σ into an existing group region ρ (ADOPT-GRP, Figure 13).

Once adopted, a value v can never be extracted back out of a group region ρ. Indeed, the information that v has type $[\rho]$, which means that v inhabits ρ, can be duplicated: thus, it must remain true forever. Nevertheless, in order to gain access to v, it may be necessary to temporarily isolate it out of the group region ρ. The *focus* operation permits this by placing v in a fresh singleton region σ (FOCUS-GRP, Figures 13 and 14).

Focus accepts a capability $\{\rho : \theta\}$, together with an element of ρ, that is, a value of type $[\rho]$. It creates a fresh singleton region σ, which holds just that element, now viewed at type $[\sigma]$. Because the capability $\{\sigma : \theta\}$ is now available, and has overlap with $\{\rho : \theta\}$, letting these two capabilities co-exist would be unsound. Instead, the latter is revoked, and replaced with the weaker form $\{\rho : \theta \setminus \sigma\}$. Such a capability can be thought of as "region ρ, in which a hole has been carved out at σ", in Boyland and Retert's terminology [4]. This *disabled* capability does not allow access to ρ, but can be transformed back into $\{\rho : \theta\}$ when $\{\sigma : \theta\}$ is released (UNFOCUS-GRP, Figure 13).

In the particular case of references with non-linear contents, a version of "ref" that allows allocation within a group region can be derived using adoption. Similarly, surrounding "get" and "set" with instances of focus and unfocus allows deriving versions of these operations that are applicable directly to group regions.

2.4 Translation

In a well-typed program, capabilities over regions are threaded through computations. Whereas, in the source program, capabilities are type-theoretic entities, they are turned by the translation into runtime representations of the contents of regions. Thus, in the translated program, representations of fragments of memory are threaded through computations.

A group capability $\{\rho : \theta\}$ is translated as a finite map that associates keys with values. The inhabitants of region ρ, which have type $[\rho]$, are translated as keys. (We assume that the target language has primitive notions of keys and of finite association maps.) The logical operations that involve group regions, such as adoption and focus, are translated as operations over keys and finite maps, such as fresh key generation, and map lookup, update, and extension.

A singleton capability $\{\sigma : \theta\}$ is translated as a map of unit to a value, which, in practice, is represented simply as a value. The "ref" type constructor vanishes: that is, the translation of "ref θ" is just the translation of θ. This may seem surprising, but is explained by the fact that "ref" is a type of *unique* references. The translation introduces keys and maps only where there is aliasing, that is, where group regions are used. For instance, the capability $\{\sigma : \mathrm{ref\,int} \times \mathrm{ref\,int}\}$, which describes a pair of distinct integer references, is translated simply as a pair of integers. The inhabitant of region σ, which has type $[\sigma]$, is translated as the unit value.

2.5 Contribution

Type system Regions [20], group capabilities [7], singleton capabilities and strong update [19, 22], adoption and focus [12] are borrowed from earlier work. Our type system combines these ideas, adapts them to a high-level programming language in the style of ML, streamlines their presentation, and introduces a few new features, which we now summarize.

- *Decoupling logical and physical indirection.* Our type constructors for memory indirection ($\mathrm{ref}\,\cdot$) and for membership in a region ($[\cdot]$) are orthogonal. Our regions denote sets of values that are not necessarily memory locations. In the systems cited above, a single type constructor conflates the properties of being a memory location and of belonging to a certain region.
- *Nested memory types.* Our "ref" types can be nested and combined with other memory type constructors, such as products, sums, and recursive types. For instance, $\mu\beta.\mathrm{ref}\,(\mathrm{unit} + \alpha \times \beta)$ (§3.1) is a valid memory type, which represents the ownership of an entire list, including all of the list items. By contrast, in earlier systems [22, 12], memory types are shallow: a capability for a list structure is encoded as a capability for the root cell, under the convention that each cell of the list stores a capability for its successors. Although the two approaches have equivalent expressive power, viewing nested memory types as primitive seems more suited to a high-level programming language.
- *Sum types.* Our type system offers a direct treatment of the sum type constructor ($\cdot + \cdot$). Earlier type systems have dealt only with lower-level, untagged unions ($\cdot \cup \cdot$) [19, 22].
- *Unrestricted polymorphism.* The type system offers universal and existential quantification, in the style of System F, over value and memory types, capabilities, and regions. This is done in the presence of references, yet without a value restriction [16, p. 336], thanks to capabilities, which act as explicit store typings. Although this idea seems natural, we believe that it has not appeared in the literature.

Translation Our translation of well-typed imperative programs into semantically equivalent, purely functional programs is perhaps the most visible contribution of this paper. This translation subsumes the standard monadic translation. By using fine-grained store fragments, it produces programs in which a great amount of separation information is syntactically apparent. By using homogeneous store fragments, it produces programs that are well-typed in F_μ. This is not true of the monadic translation, due to its use of a monolithic, heterogeneous store.

3. Examples

In order to better explain the type system and the translation, we present two examples of imperative data structures: mutable linked lists and union-find. We show the types of each operation in the source and target calculi, separated with a $\triangleright$ symbol. Source code and translated code is shown only for list reversal. For more details and additional examples, see the online addendum [1].

The translation produces redundant occurrences of the unit type, which correspond, in the source program, to occurrences of singleton types of the form $[\sigma]$. In principle, these redundant units could be eliminated in a separate pass. For clarity, we have kept them.

3.1 Mutable Lists

Consider a linked list where every cell is mutable, owns an item of type α, and owns the next cell (so that, indirectly, it owns the

tail of the list). Such a data structure is described by a recursive memory type: a list is a reference to either unit (if the list is empty) or a pair of an item (the head) and a list (the tail). We introduce an abbreviation for this recursive type:

$$\text{mlist}\,\alpha \quad := \quad \mu\beta.\text{ref}\,(\text{unit} + \alpha \times \beta)$$

In the target calculus, ordinary, immutable lists are described by the following recursive type:

$$\text{list}\,\alpha \quad := \quad \mu\beta.(\text{unit} + \alpha \times \beta)$$

Because the "ref" type constructor vanishes through the translation, the image of "mlist α" through the translation is "list α". That is, a mutable list is translated simply to an immutable list. (Of course, the type of the list items is translated too. Here, this is not visible, because the type variable α is translated to itself.) This is possible only because, in this simple example, there is no aliasing: the definition of "mlist" forbids two mutable lists from sharing a common tail. When there is aliasing in the source program, that is, when group regions are used, then the translation involves finite maps and keys.

Nil and cons Two functions help construct mutable lists. The function "nil" creates a fresh empty list in a fresh singleton region. The function "cons" accepts an item, a mutable list, as well as capabilities for each of these values, and returns a mutable list, together with a capability for this list. Through the translation, these become the standard "nil" and "cons" constructors for immutable lists (up to redundant units).

$$\begin{array}{rcl}
\text{nil} & : & \forall\alpha.\ \text{unit} \to \exists\sigma.[\sigma] * \{\sigma : \text{mlist}\,\alpha\} \\
& \triangleright & \forall\alpha.\ \text{unit} \to \text{unit} \times \text{list}\,\alpha \\
\text{cons} & : & \forall\alpha\,\sigma_1\,\sigma_2.\ ([\sigma_1] \times [\sigma_2]) * \{\sigma_1 : \alpha\} * \{\sigma_2 : \text{mlist}\,\alpha\} \\
& & \qquad \to \exists\sigma.[\sigma] * \{\sigma : \text{mlist}\,\alpha\} \\
& \triangleright & \forall\alpha.\ \text{unit} \times \text{unit} \times \alpha \times \text{list}\,\alpha \to \text{unit} \times \text{list}\,\alpha
\end{array}$$

The idiom of packaging together a value and a capability for this value is common enough that it could warrant an abbreviation. Let $\theta^\bullet$ stand for $\exists\sigma.[\sigma] * \{\sigma : \theta\}$. (A similar notation appears in earlier work [12, 2].) This corresponds to a traditional linear type, in the sense that a value and a permission are packaged together. With this notation, the types of "nil" and "cons" in the source calculus could be written:

$$\begin{array}{rcl}
\text{nil} & : & \forall\alpha.\ \text{unit} \to (\text{mlist}\,\alpha)^\bullet \\
\text{cons} & : & \forall\alpha.\ \alpha^\bullet \times (\text{mlist}\,\alpha)^\bullet \to (\text{mlist}\,\alpha)^\bullet
\end{array}$$

These are the types ascribed to "nil" and "cons" in a traditional linear type system. Although this abbreviated notation is convenient, it is not always applicable: there are many situations where separating values and capabilities offers extra expressive power (see, e.g., the type of "iter" below).

Reverse Let us now consider a function that performs in-place reversal of a mutable list. The function "reverse" accepts a mutable list, together with a capability, and produces a mutable list, together with a capability. The capability for the argument list is not returned, which indicates that this list is no longer valid—its cells have been re-used to construct the new list. The translation of "reverse" simply maps an immutable list to an immutable list (up to redundant units).

$$\begin{array}{rcl}
\text{reverse} & : & \forall\alpha.\ (\text{mlist}\,\alpha)^\bullet \to (\text{mlist}\,\alpha)^\bullet \\
& \triangleright & \forall\alpha.\ \text{unit} \times \text{list}\,\alpha \to \text{unit} \times \text{list}\,\alpha
\end{array}$$

In the source program, "reverse" traverses the list, flipping pointers as it goes. The auxiliary function "aux", which implements a loop, expects a pointer l to the current list cell, a pointer p to the previous list cell, as well as capabilities over the two lists that these cells represent. In the translated version of "aux", the pointers l and p become unit, while the capabilities over l and p become immutable lists, which respectively represent a list to be reversed and a list that is already reversed. In summary, up to redundant units, the translation of "reverse" is a standard version of reversal for immutable lists: a tail-recursive function that uses an accumulator. The source and translated code for "reverse" is shown below:

$$\begin{array}{rl}
\text{reverse} = & \text{let aux} = \mu\text{aux}.\lambda(l,p).\text{match}\,(\text{get}\,l)\ \text{with} \\
& \quad |\,\text{inj}^1\,()\ \Rightarrow\ p \\
& \quad |\,\text{inj}^2\,(h,t)\ \Rightarrow\ \text{set}\,(l, \text{inj}^2\,(h,p))\,;\ \text{aux}\,(t,l) \\
& \text{in}\ \lambda l.\ (\text{aux}\,(l, \text{nil}\,())) \\
\triangleright & \text{let aux} = \mu\text{aux}.\lambda((),(),l,p).\text{match}\,l\ \text{with} \\
& \quad |\,\text{inj}^1\,()\ \Rightarrow\ ((),p) \\
& \quad |\,\text{inj}^2\,(h,t)\ \Rightarrow \\
& \qquad \text{let}\,l' = \text{inj}^2\,(h,p)\,\text{in aux}\,((),(),t,l') \\
& \text{in}\ \lambda((),l).\ (\text{aux}\,((),(),l,\text{nil}\,()))
\end{array}$$

Iter Last, let us consider a higher-order iterator over mutable lists. The client function f, which is applied to each item of the list in succession, has effect $\{\sigma : \alpha\} * \beta$, where β is a capability variable. The first conjunct allows f to access and modify the current list item. The second conjunct allows f to perform side-effects on some piece of the outside world, described by β. The application of "iter" to f has effect $\{\sigma : \text{mlist}\,\alpha\} * \beta$, which means that it affects both the list and the store fragment controlled by β. Note that, because of the separating conjunction, the capabilities $\{\sigma : \text{mlist}\,\alpha\}$ and β represent disjoint store fragments. This implies that the client function f is not allowed to modify the structure of the list while it is being traversed.

$$\begin{array}{rcl}
\text{iter} & : & \forall\alpha\beta.\ (\forall\sigma.\ [\sigma] \to_{\{\sigma:\alpha\}*\beta} \text{unit}) \\
& & \qquad \to (\forall\sigma.\ [\sigma] \to_{\{\sigma:\text{mlist}\,\alpha\}*\beta} \text{unit}) \\
& \triangleright & \forall\alpha\beta.\ (\text{unit} \times \alpha \times \beta \to \text{unit} \times \alpha \times \beta) \\
& & \qquad \to (\text{unit} \times \text{list}\,\alpha \times \beta \to \text{unit} \times \text{list}\,\alpha \times \beta)
\end{array}$$

In the target calculus, "iter" is translated to a combination of the standard "map" and "fold" combinators over immutable lists.

3.2 Union-find

Our next example illustrates how group regions are used to describe data structures with sharing. It is a version of Tarjan's union-find data structure. This data structure consists of a set of nodes, organized as a forest: each node either is a root or points to a parent node. Nodes are mutable: the forest evolves over time. The nodes of a single instance of the union-find algorithm are placed in a group region ρ. Their structure is given by the following memory type:

$$\text{node}\,\rho \quad := \quad \text{ref}\,(\text{unit} + [\rho]) \quad \triangleright \quad \text{unit} + \text{key}$$

A node is a reference to either unit (if this a root node) or a node (if this is an internal node).

The group capability $\{\rho : \text{node}\,\rho\}$ describes the entire data structure: it represents the ownership of all nodes. Its translation, a finite map, maps a key (which represents a node) to either unit (if this is a root node) or a key (if this is an internal node):

$$\text{forest} \quad := \quad \text{map}\,(\text{unit} + \text{key})$$

The following functions, which manipulate the union-find data structure, require and return the capability $\{\rho : \text{node}\,\rho\}$. Through the translation, they become functions that require and return an explicit representation of the forest.

$$\begin{array}{rcl}
\text{new} & : & \forall\rho.\ \text{unit} \to_{\{\rho:\text{node}\,\rho\}} [\rho] \\
& \triangleright & \text{unit} \times \text{forest} \to \text{key} \times \text{forest} \\
\text{find} & : & \forall\rho.\ [\rho] \to_{\{\rho:\text{node}\,\rho\}} [\rho] \\
& \triangleright & \text{key} \times \text{forest} \to \text{key} \times \text{forest} \\
\text{union} & : & \forall\rho.\ [\rho] \times [\rho] \to_{\{\rho:\text{node}\,\rho\}} \text{unit} \\
& \triangleright & \text{key} \times \text{key} \times \text{forest} \to \text{unit} \times \text{forest}
\end{array}$$

$$
\begin{array}{lll}
\textit{Values} & v & ::= x \mid () \mid \mathrm{inj}^i v \mid (v_1, v_2) \mid \mu f.\lambda x.t \mid p \mid l \\
\textit{Prim. ops.} & p & ::= \mathrm{case} \mid \mathrm{proj}^i \mid \mathrm{ref} \mid \mathrm{get} \mid \mathrm{set} \\
\textit{Terms} & t & ::= v \mid (v\, t)
\end{array}
$$

Figure 1. Source language syntax

$$
\begin{array}{ll}
((\mu f.\lambda x.t)\, v) \;/\; s & \longrightarrow ([f \to \mu f.\lambda x.t]\,[x \to v]\, t) \;/\; s \\
(\mathrm{case}\,((\mathrm{inj}^i\, v), v_1, v_2)) \;/\; s & \longrightarrow (v_i\, v) \;/\; s \\
(\mathrm{proj}^i\,(v_1, v_2)) \;/\; s & \longrightarrow v_i \;/\; s \\
(\mathrm{ref}\, v) \;/\; s & \longrightarrow l \;/\; s \uplus [l \mapsto v] \\
(\mathrm{get}\, l) \;/\; s & \longrightarrow s[l] \;/\; s \\
(\mathrm{set}\,(l, v)) \;/\; s & \longrightarrow () \;/\; s[l \mapsto v] \\
(v\, t) \;/\; s & \longrightarrow (v\, t') \;/\; s' \quad \text{if } (t \;/\; s \longrightarrow t' \;/\; s')
\end{array}
$$

Figure 2. Source language semantics

$$
\begin{array}{lll}
\textit{Values} & w & ::= x \mid () \mid \mathrm{inj}^i\, w \mid (w_1, w_2) \mid \mu f.\lambda x.u \mid q \mid k \mid m \\
\textit{Prim. ops.} & q & ::= \mathrm{case} \mid \mathrm{proj}^i \mid \\
 & & \quad \mathrm{map\_fresh} \mid \mathrm{map\_add} \mid \mathrm{map\_get} \mid \mathrm{map\_set} \\
\textit{Terms} & u & ::= w \mid (w\, u)
\end{array}
$$

Figure 3. Target language syntax

The function "new" creates a new node; its translation extends the map that represents the region ρ at a fresh key and returns that key. "find" follows parent pointers out of a given node until it reaches a root node, and performs path compression; its translation involves map lookup and update operations. "union" merges two components, via a side effect; its translation involves calls to "find", as well as direct map lookup and update operations.

The capability $\{\rho : \mathrm{node}\,\rho\}$ does not encode the fact that the nodes form a forest, that is, the fact that there are no cycles. The type system is not capable of expressing this property. In a program logic defined on top of the type system, this property would be expressed as an invariant (a pre- and post-condition) of the above functions.

4. Source and target languages

The source language of our translation is a λ-calculus equipped with imperative features. The target language is a pure λ-calculus equipped with a primitive form of finite association maps. Figures 1 through 4 give the syntax and small-step operational semantics of the two languages.

A source language configuration is a pair of a term t and of a store s. A store is a finite map of locations l to values v. Values are built out of variables, unit, injections, pairs, recursive λ-abstractions, unary primitive operations p, and locations l. (The variables x and f range over a single class. The binder μf is omitted when unnecessary.) The primitive operations include eliminators for sums and products as well as the standard operations for allocating, reading, and writing a reference cell. Terms include values v and function applications $(v\, t)$. The asymmetric character of applications prevents any ambiguity about evaluation order and simplifies the reduction and typing rules. Sequencing $(\mathrm{let}\, x = t_1\, \mathrm{in}\, t_2)$ is encoded as $((\lambda x.\, t_2)\, t_1)$, and $(t_1\,;\; t_2)$ is encoded as $(\mathrm{let}\, () = t_1\, \mathrm{in}\, t_2)$.

$$
\begin{array}{ll}
(\mu f.\lambda x.u)\, w & \longrightarrow [f \to \mu f.\lambda x.u][x \to w]\, u \\
\mathrm{proj}^i\,(w_1, w_2) & \longrightarrow w_i \\
\mathrm{case}\,(\mathrm{inj}^i\, w, w_1, w_2) & \longrightarrow (w_i\, w) \\
\mathrm{map\_fresh}\, m & \longrightarrow \min\{k \mid k \notin \mathrm{dom}(m)\} \\
\mathrm{map\_add}\,(m, k, w) & \longrightarrow m \uplus [k \mapsto w] \\
\mathrm{map\_get}\,(m, k) & \longrightarrow m[k] \\
\mathrm{map\_set}\,(m, k, w) & \longrightarrow m[k \mapsto w] \\
(w\, u) & \longrightarrow (w\, u') \quad \text{if } (u \longrightarrow u')
\end{array}
$$

Figure 4. Target language semantics

$$
\begin{array}{lll}
o & ::= & \alpha \mid \bot \mid \top \mid \mathrm{unit} \mid [o] \mid o + o \mid o \times o \mid o \to o \mid \mathrm{ref}\, o \mid \\
 & & o * o \mid \forall \alpha.o \mid \exists \alpha.o \mid \mu \alpha.o \mid \emptyset \mid \{o : o\} \mid \{o : o \setminus o\} \mid \\
 & & \varnothing \mid o,\, \alpha \mid o,\, x : o \\
\kappa & ::= & \textsc{val} \mid \textsc{mem} \mid \textsc{cmp} \mid \textsc{cap} \mid \textsc{sng} \mid \textsc{grp} \mid \textsc{dnv} \mid \textsc{lnv}
\end{array}
$$

Figure 5. Syntax of types, capabilities, environments and kinds

$$
\begin{array}{l}
(\tau : \textsc{val})\,(\theta : \textsc{mem})\,(\chi : \textsc{cmp})\,(C : \textsc{cap}) \\
(\sigma : \textsc{sng})\,(\rho : \textsc{grp})\,(\Delta : \textsc{dnv})\,(\Gamma : \textsc{lnv})
\end{array}
$$

Figure 7. Conventional metavariables

The target language is purely functional. Its values are written w and its terms are written u. In lieu of memory locations l and memory stores s, it has keys k (isomorphic to the natural numbers) and finite association maps m from keys to values. The primitive operations q include operations on finite maps instead of operations on references. In short, "map_fresh" deterministically returns a key that does not appear in the domain of its argument; "map_get" looks up a key in a map; "map_set" updates a map at an existing key, producing a new map; and "map_add" extends a map at a previously undefined key, also producing a new map. We let "map_empty" stand for the empty map; it is a value.

5. Capabilities and types

The definition of the type system involves various kinds of entities, and several connectives are shared between multiple kinds. In order to avoid a combinatorial explosion in the syntax, we find it necessary to define a single syntactic category of *objects* o, and to then classify objects using *kinds* κ (Figure 5). The well-kindedness rules (Figure 6) are unfortunately quite technical; we suggest skipping them upon first reading. Throughout the paper, kinds remain implicit, thanks to the use of conventional metavariables (Figure 7). Type variables of all kinds are written α. We assume that every type variable α intrinsically belongs to some fixed kind κ, in which case we write $\alpha : \kappa$. We now briefly and informally review each of the kinds.

Value types τ Value types include the standard constructors bottom, top, unit, sum, product, and arrow. The type $[\alpha]$ ("at α") represents membership in the region α. The bottom type $\bot$ is useful in combination with the type constructor for sums: for instance, a value of type $(\tau + \bot)$ must be a left injection.

Memory types θ Memory types describe the structure of the elements of a region. They appear in capabilities, such as $\{\sigma : \theta\}$, and describe the extent of the piece of memory that is controlled by this capability. The grammar of memory types extends that of value types with references ($\theta ::= \mathrm{ref}\,\theta$) and with a separating con-

$$\frac{o \,:\, \text{VAL}}{o \,:\, \text{CMP}} \qquad \frac{o \,:\, \text{VAL}}{o \,:\, \text{MEM}} \qquad \frac{}{\bot \,:\, \text{VAL}} \qquad \frac{}{\top \,:\, \text{VAL}} \qquad \frac{}{\text{unit} \,:\, \text{VAL}} \qquad \frac{o \,:\, \kappa}{[o] \,:\, \text{VAL}}\ \kappa\in\{\text{SNG},\text{GRP}\}$$

$$\frac{o_1 \,:\, \kappa \qquad o_2 \,:\, \kappa}{(o_1 + o_2) \,:\, \kappa}\ \kappa\in\{\text{VAL},\text{MEM}\} \qquad \frac{o_1 \,:\, \kappa \qquad o_2 \,:\, \kappa}{(o_1 \times o_2) \,:\, \kappa}\ \kappa\in\{\text{VAL},\text{MEM}\} \qquad \frac{o_1 \,:\, \text{CMP} \qquad o_2 \,:\, \text{CMP}}{(o_1 \to o_2) \,:\, \text{VAL}} \qquad \frac{o \,:\, \text{MEM}}{(\text{ref}\, o) \,:\, \text{MEM}}$$

$$\frac{o_1 \,:\, \kappa \qquad o_2 \,:\, \text{CAP}}{(o_1 * o_2) \,:\, \kappa}\ \kappa\in\{\text{MEM},\text{CAP},\text{CMP}\} \qquad \frac{\alpha \,:\, \kappa_1 \qquad o \,:\, \kappa_2}{(\forall\alpha.o) \,:\, \kappa_2}\ \begin{array}{l}\kappa_1\in\{\text{VAL},\text{MEM},\text{CAP},\text{SNG},\text{GRP}\}\\ \kappa_2\in\{\text{VAL}\}\end{array}$$

$$\frac{\alpha \,:\, \kappa_1 \qquad o \,:\, \kappa_2}{(\exists\alpha.o) \,:\, \kappa_2}\ \begin{array}{l}\kappa_1\in\{\text{VAL},\text{MEM},\text{CAP},\text{SNG},\text{GRP}\}\\ \kappa_2\in\{\text{VAL},\text{MEM},\text{CAP},\text{CMP}\}\end{array} \qquad \frac{\alpha \,:\, \kappa \qquad o \,:\, \kappa}{(\mu\alpha.o) \,:\, \kappa}\ \begin{array}{l}\kappa\in\{\text{VAL},\text{MEM},\text{CAP}\}\\ o \text{ not a variable or a } \mu \text{ form}\end{array}$$

$$\frac{}{\emptyset \,:\, \text{CAP}} \qquad \frac{o_1 \,:\, \kappa \qquad o_2 \,:\, \text{MEM}}{\{o_1 : o_2\} \,:\, \text{CAP}}\ \kappa\in\{\text{SNG},\text{GRP}\} \qquad \frac{o_1 \,:\, \text{GRP} \qquad o_2 \,:\, \text{MEM} \qquad o_3 \,:\, \text{SNG}}{\{o_1 : o_2 \setminus o_3\} \,:\, \text{CAP}}$$

$$\frac{}{\varnothing \,:\, \kappa}\ \kappa\in\{\text{DNV},\text{LNV}\} \qquad \frac{o \,:\, \kappa_1 \qquad \alpha \,:\, \kappa_2}{(o,\, \alpha) \,:\, \kappa_1}\ \begin{array}{l}\kappa_1\in\{\text{DNV},\text{LNV}\}\\ \kappa_2\in\{\text{VAL},\text{MEM},\text{CAP},\text{SNG},\text{GRP}\}\\ \alpha\#o\end{array} \qquad \frac{o_1 \,:\, \kappa_1 \qquad o_2 \,:\, \kappa_2}{(o_1,\, x : o_2) \,:\, \kappa_1}\ \begin{array}{l}(\kappa_1\in\{\text{DNV}\} \wedge \kappa_2\in\{\text{VAL}\})\\ \text{or } (\kappa_1\in\{\text{LNV}\} \wedge \kappa_2\in\{\text{VAL},\text{CMP},\text{CAP}\})\\ x\#o_1 \wedge \text{fv}(o_2)\subseteq\text{dom}(o_1)\end{array}$$

Figure 6. Well-kindedness

$$\begin{array}{lcl} [\![\bot]\!] & = & \bot \\ [\![\top]\!] & = & \top \\ [\![\text{unit}]\!] & = & \text{unit} \\ [\![\,[\sigma]\,]\!] & = & \text{unit} \\ [\![\,[\rho]\,]\!] & = & \text{key} \end{array} \qquad \begin{array}{lcl} [\![o_1 + o_2]\!] & = & [\![o_1]\!] + [\![o_2]\!] \\ [\![o_1 \times o_2]\!] & = & [\![o_1]\!] \times [\![o_2]\!] \\ [\![o_1 \to o_2]\!] & = & [\![o_1]\!] \to [\![o_2]\!] \\ [\![\text{ref}\, o]\!] & = & [\![o]\!] \\ [\![o_1 * o_2]\!] & = & [\![o_1]\!] \times [\![o_2]\!] \end{array} \qquad \begin{array}{lcl} [\![\emptyset]\!] & = & \text{unit} \\ [\![\{\sigma : o\}]\!] & = & [\![o]\!] \\ [\![\{\rho : o\}]\!] & = & \text{map}\ [\![o]\!] \\ [\![\{\rho : o \setminus \sigma\}]\!] & = & \text{map}\ [\![o]\!] \times \text{key} \\ [\![\varnothing]\!] & = & \varnothing \\ [\![o_1,\, x : o_2]\!] & = & [\![o_1]\!],\, x : [\![o_2]\!] \end{array}$$

$$\begin{array}{l} \text{If } (\alpha \,:\, \text{SNG}) \\ \text{or } (\alpha \,:\, \text{GRP}): \\ [\![\forall\alpha.o]\!] = [\![o]\!] \\ [\![\exists\alpha.o]\!] = [\![o]\!] \\ [\![\mu\alpha.o]\!] = [\![o]\!] \\ [\![o,\, \alpha]\!] = [\![o]\!] \end{array} \qquad \begin{array}{l} \text{Otherwise:} \\ [\![\alpha]\!] = \alpha \\ [\![\forall\alpha.o]\!] = \forall\alpha.[\![o]\!] \\ [\![\exists\alpha.o]\!] = \exists\alpha.[\![o]\!] \\ [\![\mu\alpha.o]\!] = \mu\alpha.[\![o]\!] \\ [\![o,\, \alpha]\!] = [\![o]\!],\, \alpha \end{array}$$

Figure 8. Translation of types, capabilities and environments

junction ($\theta ::= \theta * C$), which allows capabilities to be embedded within memory types. Notice that a value type for a memory location l must be of the form $[\alpha]$. The type of the contents of the location appears in the capability $\{\alpha : \text{ref}\,\theta\}$ that controls l.

Computation types χ As explained earlier (§2.2), computation types admit the productions $\chi ::= \tau \mid \chi * C \mid \exists\alpha.\chi$. They are used in function types, which take the form $\chi_1 \to \chi_2$.

Capabilities C Atomic capabilities include the null capability $\emptyset$, singleton capabilities $\{\sigma : \theta\}$, group capabilities $\{\rho : \theta\}$, and disabled capabilities $\{\rho : \theta \setminus \sigma\}$. The latter represents ownership of all elements of a group region ρ, except the unique element of a singleton region σ. Compound capabilities are built via the separating conjunction $C_1 * C_2$.

Regions σ, ρ The kinds of singleton regions σ and of group regions ρ contain only variables. We take the liberty of using σ and ρ to denote region variables, so we write $\forall\sigma$, $\exists\rho$, etc.

Environments Δ, Γ A *duplicable*, or non-linear, *environment* Δ binds type variables α, and binds variables to value types $(x : \tau)$. A *linear environment* Γ can additionally bind variables to computation types $(x : \chi)$ or to capabilities $(x : C)$. Bindings commute, provided dependencies are respected. When a variable x is bound to a capability C, it cannot occur in the program: in the source calculus, capabilities are not values, and do not exist at runtime. We assign names to capabilities not only for sake of uniformity, but also because these names are naturally used in the translation.

Quantification Universal quantification ($\forall$) is present in the syntax of value types ($\tau ::= \forall\alpha.\tau$). It is not present in the syntax of memory types or capabilities, as it would not make sense to quantify over a type variable that occurs in a store typing. Existential quantification ($\exists$) and recursive definition (μ) are available not only within value types, but also within memory types and capabilities.

Translation of types The target language is typed in F_μ with type constructors "key" (of arity 0) and "map" (of arity 1). Our translation of programs preserves types. Figure 8 shows how a source object o is translated to an F_μ object $[\![o]\!]$. The translation concerns objects of all kinds except SNG and GRP, as regions vanish in the translation. The translation of region membership types $[\alpha]$, of references, and of atomic capabilities has been described earlier (§2.4). The translation of disabled capabilities is explained later on (§6.2). A separating conjunction is translated as a product. The rest of the translation is structural.

6. Type system and translation

6.1 Structural rules

Judgements Values admit a non-linear value type in a non-linear environment, while terms admit a linear computation type in a linear environment. The corresponding judgements are

$$\Delta \vdash v : \tau \qquad \text{and} \qquad \Gamma \Vdash t : \chi$$

In other words, the construction of a value v does not consume or produce any capability. The evaluation of a term t, on the other hand, consumes the capabilities contained in the linear context Γ and produces the capabilities contained in the computation type χ.

The type-directed translation of values and terms into the target language is defined via translation judgements that extend the

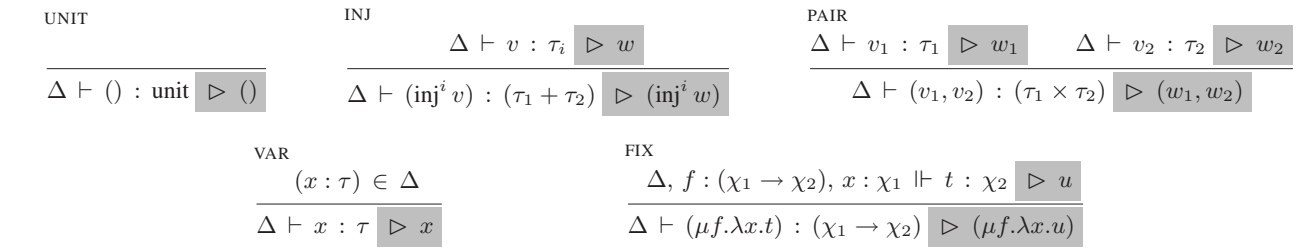

Figure 9. Type-checking and type-directed translation: values

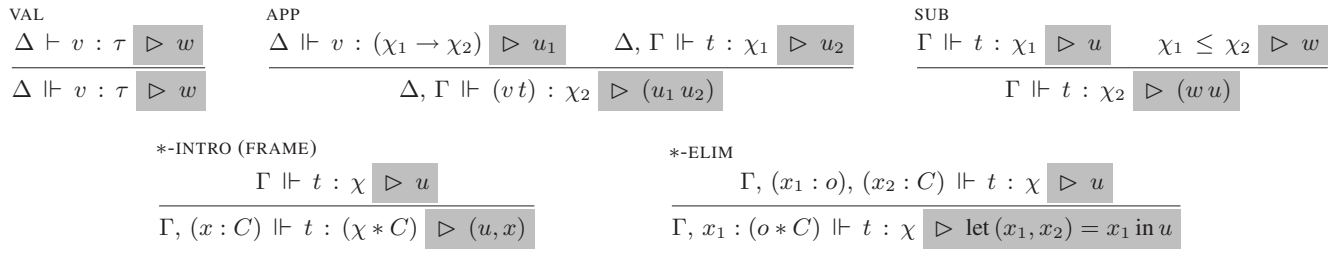

Figure 10. Type-checking and type-directed translation: terms

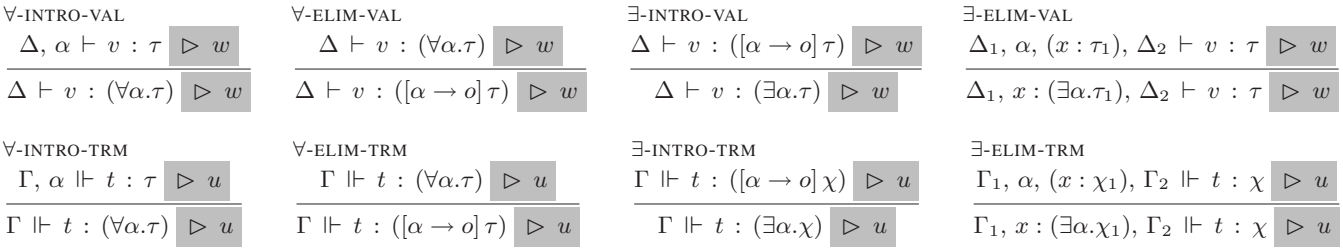

Figure 11. Additional rules for values and terms: quantifier introduction and elimination

typing judgements:

$$\Delta \vdash v : \tau \rhd w \qquad \text{and} \qquad \Gamma \Vdash t : \chi \rhd u$$

(The symbol $\rhd$ should be read: "is translated to".) The left-hand judgement states that the source value v is translated to the target value w. The right-hand judgement states that the source term t is translated to the target term u.

The translation preserves well-typedness, compositionally. This fact admits the following succinct statement (in which the symbol $\vdash_{F_\mu}$ indicates well-typedness in F_μ):

Lemma (Type preservation)

$$\begin{array}{lll} \Delta \vdash v : \tau \rhd w & \textit{implies} & [\![\Delta]\!] \vdash_{F_\mu} w : [\![\tau]\!] \\ \Gamma \Vdash t : \chi \rhd u & \textit{implies} & [\![\Gamma]\!] \vdash_{F_\mu} u : [\![\chi]\!] \end{array} \quad \diamond$$

The translation judgements extend the typing judgements in the following sense. First, every valid translation judgement contains a valid typing judgement, which can be recovered simply by erasing the translation-specific annotations. Conversely, every valid typing judgement is the erasure of some valid translation judgement.

In order to save space and avoid redundancy, only the translation rules are presented in this paper (Figures 9, 10 and 11). The translation-specific parts are printed on a gray background, so that, by ignoring them, one recovers the typing rules.

Values (Figure 9) Values are type-checked in a standard way. Their translation is structural.

The rule that type-checks recursive functions does hide a couple of subtleties. First, it uses computation types χ, which describe the transfer not only of a value, but also of capabilities. Second, it uses a duplicable environment Δ, which means that a closure cannot capture a capability that happens to be available at its allocation site. This is required for type soundness [12].

Terms (Figure 10) VAL states that a value of type τ can be viewed as a term of type τ. This statement is well-formed, because every duplicable environment is also a linear environment, and every value type is also a computation type.

APP states that the application $(v\, t)$ has type χ_2 if the function v has type $\chi_1 \to \chi_2$ and the argument t has type χ_1. Because the left-hand side is a value, which consumes no capability, all of the available capabilities, represented by Γ, are transmitted to the right-hand side t. The environment fragment Δ, which is duplicated, does not contain any capability. An application is translated to an application.

The subtyping rule, SUB, weakens the type of a term. The subtyping relation is defined later on (§6.2). For the moment, note that a subtyping judgement translates to a coercion, that is, a closed λ-term, and that an instance of SUB gives rise, in the translated term, to an application of a coercion.

$*$-INTRO, also known as FRAME, states that if a term is fed with a capability C that it does not need, then its evaluation preserves that capability. This is the first-order frame rule of Separation Logic [18]. An instance of FRAME is translated to a pair, whose first component is the translation of the term t, and whose second component is the translation of the capability C. (In order to satisfy our syntactic restrictions, the notation (u, x) is defined as syntactic sugar for the β-expanded form "let $x' = u$ in (x', x)".)

$*$-ELIM is a left elimination form for separating conjunctions, which may be of the form $\chi * C$ or $C_1 * C_2$. Because conjunctions are translated as pairs, the deconstruction of a conjunction is translated as pair decomposition.

Quantifiers (Figure 11) All quantifier introduction and elimination rules are standard, except $\forall$-INTRO-TRM, which states that it is permitted to generalize the type of a term. The very existence of this rule means that there is no value restriction [16, p. 336]. Yet, the system is sound. This is guaranteed by a restriction on the syntax of types: while $(\forall\alpha.\tau)$ is a valid type, $(\forall\alpha.\chi)$ is not. Indeed, a computation type χ may contain capabilities, which describe the structure of a piece of store; the type variables that occur within these capabilities must not be generalized.

***Deriving* LET** Because the "let" construct is sugar for a β-redex, its typing rule follows from APP, FIX, VAL, and FRAME:

$$\text{LET}\quad \frac{\Delta, \Gamma_1 \Vdash t_1 : \chi_1 \rhd u_1 \qquad \Delta, (x : \chi_1), \Gamma_2 \Vdash t_2 : \chi_2 \rhd u_2}{\Delta, \Gamma_1, \Gamma_2 \Vdash (\text{let } x = t_1 \text{ in } t_2) : \chi_2 \rhd (\text{let } x = u_1 \text{ in } u_2)}$$

6.2 Primitive operations and subtyping

The subtyping judgement takes the form:

$$o_1 \leq o_2 \rhd w$$

where o_1 and o_2 have kind $\kappa \in \{\text{VAL}, \text{MEM}, \text{CMP}, \text{CAP}\}$. The coercion w is a closed value of the target calculus (a coercion), whose type is $[\![o_1]\!] \to [\![o_2]\!]$. It is used to translate the subtyping operation.

In the following, the types of the primitive operations (Figure 12) and the subtyping rules (Figure 13) are explained together.

Notation In Figure 13, we use λ-abstractions of the form $(\lambda\pi.\, u)$, where π is a pattern, that is, a value composed of unit, pairs, injections, and distinct variables. We write $(\pi \rightsquigarrow u)$ for $(\lambda\pi.\, u)$, and let the symmetric subtyping axiom: $o_1 \equiv o_2 \rhd (\pi_1 \leftrightsquigarrow \pi_2)$ stand for the conjunction of axioms: $o_1 \leq o_2 \rhd (\pi_1 \rightsquigarrow \pi_2)$ and $o_2 \leq o_1 \rhd (\pi_2 \rightsquigarrow \pi_1)$.

Illustration Among the subtyping rules, four FOCUS rules, which reorganize regions, are informally illustrated in Figure 14. There, a solid arrow points to an object that is owned by the origin region of the arrow, so it never crosses a region boundary. On the contrary, a dashed arrow points into some distinct region, so it always crosses a region boundary. The object at the end of such an arrow has a type of the form $[\alpha]$, for some region α, and it is controlled by the capability associated with region α.

General SNG-CREATE states that every value can be viewed as a member of a singleton type. This helps derive variants of several other rules. Conversely, SNG-EXTRACT turns a singleton type $[\sigma]$ back into a value type τ, provided the capability $\{\sigma : \tau\}$ is present. (SNG-EXTRACT can be used to show that SNG-CREATE is in fact an equivalence.) These rules reflect the fact that ownership of a non-linear value is never exclusive. This is acceptable, because values are immutable.

FREE discards a capability C. Technically, its presence means that capabilities are affine, rather than linear. Its translation, $(\lambda x.\, ())$, discards x, which represents the memory controlled by C.

References The types of the primitive operations "ref", "get", and "set", as well as the subtyping rule FOCUS-REF (illustrated in Figure 14), have been explained earlier (§2.2). The translation of "ref", "get" and "set" is exactly the standard monadic translation [13, 21], in the particular case where the state consists of a single memory cell. In the translated λ-terms, x stands for the translation of $\{\sigma : \text{ref}\,\tau\}$, that is, the contents of the cell. The location l at type $[\sigma]$ is translated to unit.

Pairs Just like the primitive operations on references, the pair projections extract a non-linear component out of a linear container. That is, "proj^1" requires a capability of the form $\{\sigma : \tau_1 \times \theta_2\}$, and returns a value of type τ_1.

In order to extract a linear component out of a pair, one relies on the subtyping rule FOCUS-PAIR^1 (from left to right) to isolate the first component of a pair in a fresh region (See Figure 14). This mechanism is identical to that provided by FOCUS-REF for accessing references with linear contents.

Conversely, applying rule FOCUS-PAIR^1 twice, from right to left, allows constructing a linear pair $\{\sigma : \theta_1 \times \theta_2\}$ out of a value of type $([\sigma_1] \times [\sigma_2])$ and out of the two capabilities $\{\sigma_1 : \theta_1\}$ and $\{\sigma_2 : \theta_2\}$.

The traditional type $(\tau_1 \times \tau_2 \to \tau_i)$ for projection out of a non-linear pair can be derived, via SNG-CREATE and FREE. Moreover, up to $\beta\eta-$equivalence, its translation is just projection.

Sums The type of "case" is somewhat complex. The reason is that "case" performs two tasks: on the one hand, it branches on a tag; on the other hand, it deconstructs a sum by stripping off its tag and returning the underlying value. In the following, we explain how these tasks could be assigned separate types. The type of "case" is then derived as a combination of these types and of a focusing step.

First, a hypothetical primitive operation, whose dynamic semantics is to examine the tag of a sum and transfer control, accordingly, to one of two branches, could be assigned the type:

$$\begin{array}{l}\big(\ (\text{unit} * \{\sigma : \theta_1 + \bot\} * C \ \to \chi) \\ \times\ (\text{unit} * \{\sigma : \bot + \theta_2\} * C \ \to \chi) \\ \times\ [\sigma]\big) * \{\sigma : \theta_1 + \theta_2\} * C \ \to \chi\end{array}$$

Here, each branch consumes a capability C and produces a result of type χ; the same is true of the entire "case" construct. The capability $\{\sigma : \theta_1 + \theta_2\}$ is transformed to $\{\sigma : \theta_1 + \bot\}$ in the first branch and to $\{\sigma : \bot + \theta_2\}$ in the second branch, reflecting the knowledge acquired by examining the tag. This knowledge can be discarded, so as to recover the original capability, via the subtyping axiom $\bot \leq \theta$.

Second, a hypothetical primitive operation, whose dynamic semantics is to deconstruct a left injection could be assigned the type:

$$(\tau + \bot) \to \tau$$

The application of this function is well-typed only if its argument is statically known to be a left injection.

Third and last, it is natural to introduce the subtyping rule FOCUS-SUM^i. This rule is analogous to FOCUS-PAIR^i. It allows isolating the contents of an injection in a fresh region, when the tag is statically known (see Figure 14).

Combining the types of the above two hypothetical primitive operations with instances of FOCUS-SUM^i yields the type of "case", shown in Figure 12.

Regions There are four subtyping rules for producing or consuming group regions [12]. These rules are purely logical: they change one's view of memory, but have no runtime effect in the source language. In the translation, these subtyping rules become instructions for rearranging the finite maps that represent regions.

NEW-GRP allows the creation of n fresh, empty group regions. Every θ_i can contain free occurrences of every ρ_j, which is why

$$
\begin{array}{lcll}
\text{ref} & : & \tau \to \exists\sigma.[\sigma] * \{\sigma : \text{ref}\,\tau\} & \triangleright \quad \lambda x.\,((), x) \\
\text{get} & : & [\sigma] * \{\sigma : \text{ref}\,\tau\} \to \tau * \{\sigma : \text{ref}\,\tau\} & \triangleright \quad \lambda((), x).\,(x, x) \\
\text{set} & : & ([\sigma] \times \tau_2) * \{\sigma : \text{ref}\,\tau_1\} \to \text{unit} * \{\sigma : \text{ref}\,\tau_2\} & \triangleright \quad \lambda((), x_2, x_1).\,((), x_2) \\
\text{proj}^1 & : & [\sigma] * \{\sigma : \tau_1 \times \theta_2\} \to \tau_1 * \{\sigma : \tau_1 \times \theta_2\} & \triangleright \quad \lambda((), (x_1, x_2)).\,(x_1, (x_1, x_2)) \\
\text{case} & : & \big(((\exists\sigma_1.([\sigma_1] * \{\sigma : [\sigma_1] + \bot\} * \{\sigma_1 : \theta_1\} * C)) \to \chi) & \triangleright \quad \lambda(f_1, f_2, (), x, c).\,\text{case}\,(\\
 & & \quad \times ((\exists\sigma_2.([\sigma_2] * \{\sigma : \bot + [\sigma_2]\} * \{\sigma_2 : \theta_2\} * C)) \to \chi) & \qquad (\lambda x_1.\,(f_1\,((), \text{inj}^1(), x_1, c))), \\
 & & \quad \times [\sigma]\,\big) * \{\sigma : \theta_1 + \theta_2\} * C \to \chi & \qquad (\lambda x_2.\,(f_2\,((), \text{inj}^2(), x_2, c))), x)
\end{array}
$$

Figure 12. Typing and translation of primitives

$$
\begin{array}{lcll}
\textbf{General} & & & \\
\text{SNG-CREATE} & : & \tau \le \exists\sigma.[\sigma] * \{\sigma : \tau\} & \triangleright \quad x \rightsquigarrow ((), x) \\
\text{SNG-EXTRACT} & : & [\sigma] * \{\sigma : \tau\} \le \tau * \{\sigma : \tau\} & \triangleright \quad ((), x) \rightsquigarrow (x, x) \\
\text{FREE} & : & C \le \emptyset & \triangleright \quad x \rightsquigarrow () \\
 & & & \\
\textbf{Focus-value} & & & \\
\text{FOCUS-REF} & : & \{\sigma : \text{ref}\,\theta_1\} \equiv \exists\sigma_1.\{\sigma : \text{ref}\,[\sigma_1]\} * \{\sigma_1 : \theta_1\} & \triangleright \quad x \leftrightsquigarrow ((), x) \\
\text{FOCUS-PAIR}^1 & : & \{\sigma : \theta_1 \times \theta_2\} \equiv \exists\sigma_1.\{\sigma : [\sigma_1] \times \theta_2\} * \{\sigma_1 : \theta_1\} & \triangleright \quad (x_1, x_2) \leftrightsquigarrow (((), x_2), x_1) \\
\text{FOCUS-SUM}^1 & : & \{\sigma : \theta_1 + \bot\} \equiv \exists\sigma_1.\{\sigma : [\sigma_1] + \bot\} * \{\sigma_1 : \theta_1\} & \triangleright \quad (\text{inj}^1\,x) \leftrightsquigarrow (\text{inj}^1\,(), x) \\
 & & & \\
\textbf{Regions} & & & \\
\text{NEW-GRP} & : & \emptyset \le \exists\rho_1 \cdots \rho_n.\{\rho_1 : \theta_1\} * \ldots * \{\rho_n : \theta_n\} & \triangleright \quad () \rightsquigarrow (\text{map\_empty}, \ldots, \text{map\_empty}) \\
\text{ADOPT-GRP} & : & [\sigma] * \{\sigma : \theta\} * \{\rho : \theta\} \le [\rho] * \{\rho : \theta\} & \triangleright \quad ((), x, m) \rightsquigarrow \text{let}\,k = \text{map\_fresh}\,m\,\text{in} \\
 & & & \qquad\qquad (k, \text{map\_add}\,(m, k, x)) \\
\text{FOCUS-GRP} & : & [\rho] * \{\rho : \theta\} \le \exists\sigma.[\sigma] * \{\sigma : \theta\} * \{\rho : \theta \setminus \sigma\} & \triangleright \quad (k, m) \rightsquigarrow ((), \text{map\_get}\,(m, k), (m, k))) \\
\text{UNFOCUS-GRP} & : & \{\sigma : \theta\} * \{\rho : \theta \setminus \sigma\} \le \{\rho : \theta\} & \triangleright \quad (x, (m, k)) \rightsquigarrow \text{map\_set}\,(m, k, x) \\
 & & & \\
\textbf{Embedding} & & & \\
\exists.\text{EMBED} & : & \{\sigma : (\exists\alpha.\theta)\} \equiv \exists\alpha.\{\sigma : \theta\} & \triangleright \quad x \leftrightsquigarrow x \\
*.\text{EMBED} & : & \{\sigma : (\theta * C)\} \equiv \{\sigma : \theta\} * C & \triangleright \quad x \leftrightsquigarrow x \\
 & & & \\
\textbf{Administrative} & & & \\
*.\text{COMM-CAP} & : & C_1 * C_2 \equiv C_2 * C_1 & \triangleright \quad (x_1, x_2) \leftrightsquigarrow (x_2, x_1) \\
*.\text{ASSOC} & : & (o * C_1) * C_2 \equiv o * (C_1 * C_2) & \triangleright \quad ((x_1, x_2), x_3) \leftrightsquigarrow (x_1, (x_2, x_3)) \\
*.\text{NEUTRAL} & : & o * \emptyset \equiv o & \triangleright \quad (x, ()) \leftrightsquigarrow x \\
\exists.\text{COMM} & : & \exists\alpha_1.\exists\alpha_2.o \equiv \exists\alpha_2.\exists\alpha_1.o & \triangleright \quad x \leftrightsquigarrow x \\
\exists.\text{EXTRUDE-L} & : & o_1 * (\exists\alpha.o_2) \equiv \exists\alpha.(o_1 * o_2) & \triangleright \quad x \leftrightsquigarrow x \\
\exists.\text{EXTRUDE-R} & : & (\exists\alpha.o_1) * o_2 \equiv \exists\alpha.(o_1 * o_2) & \triangleright \quad x \leftrightsquigarrow x
\end{array}
$$

Figure 13. Subtyping rules and their translation

permitting $n > 1$ is useful. The capabilities over these empty regions are translated as empty maps.

ADOPT-GRP dissolves a singleton region σ into an existing group region ρ. The capability over σ is lost. The unique inhabitant of σ henceforth becomes an inhabitant of ρ. Its type is coerced from $[\sigma]$ to $[\rho]$, so as to reflect this change. In the target calculus, the capability $\{\rho : \theta\}$ is translated to an association map m, which represents the initial state of region ρ. The capability $\{\sigma : \theta\}$ is translated to a value x, which represents the state of the object σ. Because σ is a singleton region, the object itself is translated to unit. An application of "map_fresh" produces a key k that does not already appear in the domain of m. Then, an application of "map_add" extends the association map m with x at k, yielding an updated representation of region ρ. Finally, because ρ is a group region, the adopted object, at type $[\rho]$, is translated to the key k.

FOCUS-GRP isolates a particular object out of a group region: this creates a fresh singleton region, and disables the group region. UNFOCUS-GRP undoes this effect. This mechanism was explained earlier (§2.3; Figure 14). In the target language, a disabled capability $\{\rho : \theta \setminus \sigma\}$ is translated to a pair (m, k), where the association map m represents the full capability $\{\rho : \theta\}$, and the key k represents the index of object σ within region ρ. Intuitively, the meaning of the pair (m, k) is that the state of region ρ is m, except at key k, where the value is stale and must not be accessed. FOCUS-GRP creates such a pair (m, k), which forms the translation of $\{\rho : \theta \setminus \sigma\}$, and looks up the map m at index k, so as to form the translation of $\{\sigma : \theta\}$. UNFOCUS-GRP requires a pair (m, k), which represents $\{\rho : \theta \setminus \sigma\}$, and updates m at k with a new value x, which represents $\{\sigma : \theta\}$. The value previously found in m at k, which at this point is stale, is overwritten.

A typical imperative coding pattern consists in focusing on an object σ, updating it via a side effect, then de-focusing. In the translation, this corresponds to looking up a value at some key k in an association map, computing an updated value, then updating the map at key k with that new value. Such a sequence is a typical functional programming idiom.

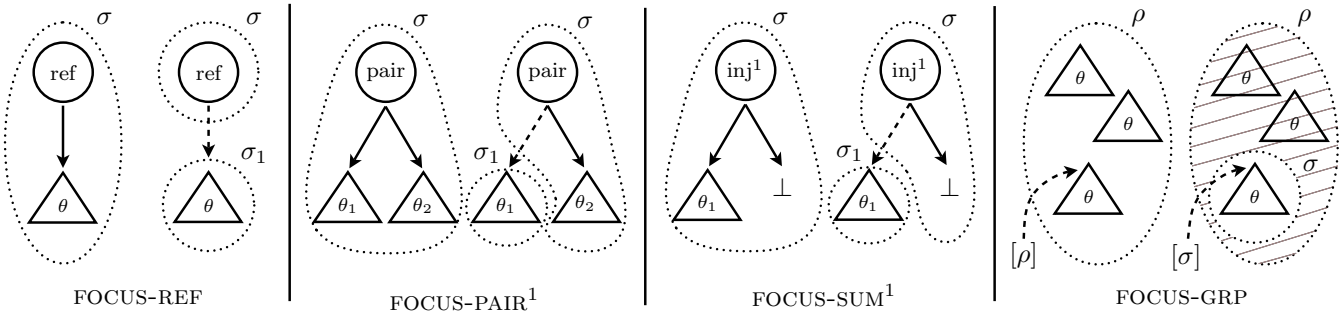

Figure 14. Illustration of the focus operations

Embedding $*$.EMBED and $\exists$.EMBED help attach regions and capabilities to the components of a data structure, and, conversely, extract them back out. For instance, they can convert the conjunction $\exists\rho.(\{\sigma : \theta\} * \{\rho : \theta'\})$ to the atom $\{\sigma : \exists\rho.(\theta * \{\rho : \theta'\})\}$, where region ρ is owned by region σ. This mechanism allows regions to form an ownership hierarchy. Because both inner and outer conjunctions are translated as pairs, the coercions that witness these rules are the identity.

Administrative The $*$ connective is commutative, when applied to two capabilities, associative, and admits the null capability $\emptyset$ as a neutral element. Moreover, existential binders can commute and be extruded out of conjunctions.

6.3 Subtyping under a context

Subtyping is applicable under a context. Most of the corresponding rules are standard, thus not shown (see the online addendum [1]).

Subtyping references The "ref" type constructor is covariant. This might come as a shock, as it is well-known that soundness requires references to be invariant in extensions of simply-typed λ-calculus with references and subtyping [16, p. 198]. Here, however, "ref" is not a value type constructor. Instead, it is a memory type constructor: it is linear. It is safe to weaken the type of the contents of a reference, for the same reason that it is safe to perform a strong update: there exists only one copy of this type.

Subtyping recursive types For comparing recursive types, we use rules that closely resemble those of Brandt and Henglein [5]. The rules involve subtyping contexts Σ, which are sets of subtyping assumptions. Two symmetric rules allow unfolding a recursive type, so as to compare it with some other type. The left rule is:

SUB-REC-LEFT

$$\frac{o_1 = \mu\alpha.o \quad \Sigma,\ (o_1 \le o_2 \ \rhd\ x) \vdash ([\alpha \to o_1]\, o) \le o_2 \ \rhd\ w}{\Sigma \vdash o_1 \le o_2 \ \rhd\ \mu x.w}$$

The conclusion of the rule appears as part of the subtyping context in the premise. This is sound, because all recursive types are contractive. The corresponding coercion is a recursive function. (Because w is necessarily an abstraction, $(\mu x.w)$ is a well-formed recursive λ-abstraction.) A standard "fold/unfold" axiom is provable:

$$\mu\alpha.o \equiv ([\alpha \to \mu\alpha.o]\, o) \quad \rhd \quad \lambda x.\, x$$

7. Soundness of the type system and translation

We use a syntactic approach to soundness, via subject reduction and progress theorems. We prove roughly the following simulation statement, which subsumes subject reduction: if a well-typed term t is translated as u, and if t reduces to t', then t' is well-typed, and its translation u' is a reduct of u. Formalizing this statement requires the introduction of several new definitions.

Region maps We must explain how to type-check and translate source configurations, rather than just source terms. To that end, we introduce an oracle, known as a *region map*, which records the contents of regions. (This is analogous to a standard type soundness proof for ML [16, §13.4], where an oracle, known as a store typing, records the types of the memory locations.) A region map μ maps a singleton region σ to a closed source value $\mu[\sigma]$, and maps a group region ρ to a map $\mu[\rho]$ from keys to closed source values.

The typing and translation judgements are extended so as to carry μ as a parameter. This new parameter is ignored by the typing and translation rules shown earlier. It is, however, exploited in the following two new rules, which assign type $[\alpha]$ to an inhabitant of region α, and translate it appropriately:

SNG

$$\frac{}{\mu;\ \Delta \vdash \mu[\sigma] : [\sigma] \ \rhd\ ()}$$

GRP

$$\frac{}{\mu;\ \Delta \vdash \mu[\rho][k] : [\rho] \ \rhd\ k}$$

Executable terms Moreover, we introduce a new judgement for typing and translating closed source terms. The judgement:

$$s;\ \mu;\ \bar\alpha;\ P \ \Vdash\ t : \chi \ \rhd\ u$$

states that the term t has type χ and is translated as the term u. This judgement is relative to a store s, a region map μ, a set of type variables $\bar\alpha$ (in the absence of a value restriction, a universal quantifier introduction rule is an evaluation context, hence execution takes place in the context of a set of type variables), and a set P of locations and regions that the term t consumes (that is, must initially own). The derivation rules for this judgement are quite similar to those that define the original typing judgement for (potentially opened) terms. An exception is the frame rule, which, for this judgement, takes the form:

EXE-FRAME

$$\frac{s;\ \mu;\ \bar\alpha;\ P_1 \ \Vdash\ t : \chi \ \rhd\ u \qquad s;\ \mu \vdash C \between P_2 \ \rhd\ w}{s;\ \mu;\ \bar\alpha;\ (P_1 \uplus P_2) \ \Vdash\ t : (\chi * C) \ \rhd\ (u, w)}$$

This rule states that, if the term t consumes P_1 to produce a result of type χ, and if the capability C controls exactly the locations and regions in the set P_2, then, when provided with the disjoint union of P_1 and P_2, the term t produces a result of type $(\chi * C)$. The right-hand premise involves another judgement form, which type-checks and translates a capability. By lack of space, this judgement is not described here; see the online addendum [1] for details.

Monotonicity A last ingredient is needed to state soundness. We introduce the notation $(s, \mu)_{\setminus P} \sqsubseteq (s', \mu')_{\setminus P'}$ to indicate that, as

a term is executed, regions can only grow, and that a term cannot affect or acquire a piece of state that it initially does not own. This notation is an abbreviation for a conjunction of three statements:

$$(1)\quad \mu \sqsubseteq \mu' \qquad (2)\quad s_{\setminus P} \subseteq s'_{\setminus P'} \qquad (3)\quad \mu_{\setminus P} \subseteq \mu'_{\setminus P'}$$

Proposition (1) states that the region map grows: singleton regions remain fixed, and group regions grow. It is defined as follows:

$$\mu \sqsubseteq \mu' \quad := \quad \begin{cases} \forall \sigma \in \mathrm{dom}(\mu),\ \mu[\sigma] = \mu'[\sigma] \\ \forall \rho \in \mathrm{dom}(\mu),\ \mu[\rho] \subseteq \mu'[\rho] \end{cases}$$

Propositions (2) and (3) are set-theoretic inclusions between restrictions of finite maps. Proposition (2) means that for every memory location l in $(\mathrm{dom}(s) \setminus P)$, the value $s'[l]$ equals the value $s[l]$ (*locations not owned cannot be affected*) and l is not in P' (*locations not owned cannot be acquired*). Proposition (3) is an analogous statement about regions.

Stating soundness We can now formally state that a reduction step in a well-typed source term is matched by one or more reduction steps in the translation of this term. The facts that regions grow with time, and that inaccessible locations and regions are unaffected and remain inaccessible, are required for the proof by induction to go through.

Theorem (Simulation) *If the following hypotheses hold:*

$$\begin{cases} t \,/\, s \longrightarrow t' \,/\, s' \\ s;\ \mu;\ \bar{\alpha};\ P \models t : \chi \rhd u \end{cases}$$

then there exists μ', P' and u' such that

$$\begin{cases} u \longrightarrow^{+} u' \\ s';\ \mu';\ \bar{\alpha};\ P' \models t' : \chi \rhd u' \\ (s, \mu)_{\setminus P} \sqsubseteq (s', \mu')_{\setminus P'} \end{cases} \qquad \diamond$$

Conclusion Independently, we prove a progress theorem: a well-typed and irreducible source term is necessarily a value. Furthermore, we prove that the translation of a value, considered as a term, must converge to a value. By combining these facts with the simulation theorem, we conclude that the type system is sound (*well-typed programs do not go wrong*), and that the type-directed translation is meaning-preserving (*a program and its translation either both diverge, or converge to related values*).

8. Related work

Capabilities The Calculus of Capabilities [7] introduces a type system with non-linear values and linear capabilities. Regions are sets of memory locations (of possibly heterogeneous type). As in this paper, a capability represents an exclusive right to access and free the contents of a region. The use of capabilities allows arbitrary separation of allocation and deallocation points, a significant gain in expressiveness compared to earlier work by Tofte and Talpin [20], where regions have lexical scope. The Calculus of Capabilities enjoys a complete collection property. Thus, it does not require garbage collection: instead, it has runtime support for regions. Specifically, the calculus has a primitive type of region handles, as well as primitive operations for creating, extending, and freeing regions. In contrast, because we are interested in a high-level language, where ownership and deallocation of immutable data structures are implicit, we omit this machinery and rely on garbage collection.

The calculus of Alias Types [19] uses singleton capabilities to describe the structure of the store at the level of individual objects and support strong update. A later paper [22] adds the ability to embed capabilities within data structures, which effectively gives rise to an ownership hierarchy.

Building upon these works, Fähndrich and DeLine's Vault [12] allows reasoning about both aliased and unique objects. Furthermore, Vault introduces adoption and focus. Together with these mechanisms comes the ability for an aliased object to own a unique object. The soundness of the type system is not argued, but Boyland and Retert later prove the soundness of a similar system, where focus works at the level of object fields [4]. Our presentation of adoption and focus is closely inspired by Fähndrich and DeLine's work, with two simplifications in the presentation, which we feel are important. First, Fähndrich and DeLine conflate regions and objects: every variable ρ serves both as the name of a unique object and as the name of a group region, which holds the object's adoptees. We avoid this unfortunate identification between a static entity (a region) and a dynamic one (an object). Second, Fähndrich and DeLine require every object to keep a list of its adoptees at runtime. This list is part of the runtime machinery that is used to avoid garbage collection. In our case, no such list is needed. Our presentation of adoption as a subtyping rule emphasizes the fact that adoption has no computational content.

L^3, a Linear Language with Locations [2], is a linear λ-calculus extended with support for references and strong updates. Following Alias Types [7], pointers and capabilities are distinguished. Pointers are typically unrestricted, while capabilities are linear. In contrast with the capabilities found in Alias Types and in this paper, which are static entities, capabilities in L^3 are values: they exist at runtime. This makes quite a difference. In our system, the only runtime operations are "ref", "get", and "set"; everything else, including the operations that move capabilities around and re-organize the ownership hierarchy (adoption, the various forms of focus, embedding, etc.), takes the form of subtyping axioms, which in the source calculus have no computational content.

Not every well-typed ML program is well-typed in our system. One tentative way of translating an ML program into our system would be to place all references in a single, global region, and to thread a capability over that region throughout the program. However, that would require a heterogeneous region, while our regions are homogeneous.

Adopting the second author's *higher-order anti-frame* rule [17] does allow encoding every ML program. However, it is not yet clear how to extend our functional translation in order to support this extra rule.

Monads Monads [13, 21] and effects [20] offer a way of statically controlling which regions of memory are read or written by a program term. Monads and effects are closely related. We have explained earlier (§2.2) how an effect is just a capability that is required and returned. A monad is just a universe of computations with a fixed effect: that is, $M\,\alpha$ can be viewed as an abbreviation for $(\mathrm{unit} \to_C \alpha)$, for a fixed capability C. With this in mind, the connection between linear types and monads imagined by Chen and Hudak [6] can be made precise in our system. A monad that encapsulates a mutable data structure, such as a linked list or a binary search tree, can be defined by the programmer (in terms of a concrete capability C for the data structure) and, thanks to existential quantification, given an abstract interface (so, in the end, only an abstract type constructor M is published).

Such a precise connection between effects, monads, and a linear type system is already fleshed out, in two stages, by Ahmed, Fluet, and Morrisett [10, 11]. First, Fluet and Morrisett [10] encode Tofte and Talpin's type and effect system into F^{RGN}, an extension of F with a region-indexed monad. Second, Ahmed, Fluet, and Morrisett [11] encode F^{RGN} into λ^{rgnUL}, a linear λ-calculus equipped with regions and capabilities; their encoding of the monad is the one suggested above. Much of the complexity of the first encoding stage lies in the fact that Tofte and Talpin's effects are sets of regions, while the monad in F^{RGN} is indexed with a sin-

gle region. This difficulty is resolved by exploiting the fact that regions have nested lifetimes, and by introducing region subtyping in F^{RGN}. It seems that it could be avoided entirely by encoding Tofte and Talpin's system directly into λ^{rgnUL}.

Is it possible, analogously, to encode Tofte and Talpin's type and effect system into our type system? Maybe, but there is a snag. Whereas Tofte and Talpin's system, as well as λ^{rgnUL}, have heterogeneous regions, our system has homogeneous regions: a group region stores objects of a single, fixed type. It is an open question whether the former can be encoded into the latter.

Translations into pure calculi The monadic translation [13, 21] is perhaps the most famous translation of imperative programs into purely functional ones. Filliâtre [8] presents a refined version of this translation, where monads are indexed with effects, so that the store consists of multiple, independent fragments. This technique, implemented in the Why tool [9], does not support aliasing. The translations of Java and C into Why implemented in the Krakatoa and Caduceus tools [9] deal with aliasing by introducing arrays that play the same role as our maps. These arrays are global, however: there is one such array per record field in the source program.

O'Hearn and Reynolds translate two variants of Algol into a polymorphic linear λ-calculus [15]. Linearity is used to establish the fact that store fragments are never duplicated, and are created and destroyed in well-identified places. On the one hand, our translation is more ambitious, since our source calculus supports dynamic memory allocation. On the other hand, we have not attempted to exploit linearity in the target calculus. It must be true that our store fragments are linear, but we have not yet proved this fact. A technical difference between the two translations is in the treatment of the FRAME rule. O'Hearn and Reynolds encode it in terms of polymorphism (a full store is passed down into FRAME, at a partially abstract type), whereas our translation is direct (only a fragment of the store is passed down).

Program logics Separation Logic [18] and Stateful Views [23] are related to one another, and to our work. In fact, the syntaxes of separation logic formulae, of stateful views, and of our capabilities, share a basic fragment: all three have constructs for empty heap, singleton heap, separating conjunction, and quantification over (static names for) memory locations. Neither of the two systems cited above has primitive group capabilities or primitive mechanisms for adoption and focus.

The Spec# static program verifier [3] extracts proof obligations out of programs expressed in a variant of the C# programming language. Like our type system, it relies on an ownership hierarchy, which can evolve dynamically. Its "pack" and "unpack" constructs, which attach and detach an owned object and an owner object, serve the same purpose as the subtyping axiom FOCUS-REF in this paper.

We share with the authors of Hoare Type Theory (HTT) [14] a motivation: reasoning about imperative programs with dynamic memory allocation. HTT incorporates Hoare-style specifications into types. In keeping with a tradition of type theory, HTT blurs the distinction between code, types, and specifications, whereas we intend to preserve this distinction: we are not convinced that it is useful to let effectful code appear within specifications.

9. Future work

In this paper, we have focused on just code and types. In the future, we plan to add support for Hoare-style specifications on top of our type system. Our specification language will be a pure type theory, such as the Calculus of Inductive Constructions. This would allow decorating imperative programs with logical assertions and extracting proof obligations, while taking advantage of the separation information provided by the type system.

We also wish to further augment the expressiveness of our system. There are several interesting candidates for new type-theoretic mechanisms. *Multi-focus* permits simultaneously focusing on multiple elements of a group region, as long as they are provably distinct. *Fusion*, a generalization of adoption, dissolves an entire region into another region. These mechanisms are more complex than those presented in this paper. They involve new forms of capabilities, and entail proof obligations: that is, the translated program is equivalent to the source program only up to validation of certain assertions embedded in the translated code.

References

[1] Arthur Charguéraud and François Pottier. Technical appendix. `http://arthur.chargueraud.org/research/2008/icfp/`.

[2] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundamenta Informaticæ*, 77(4), 2007.

[3] Mike Barnett, Rob DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004.

[4] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL*, pages 283–295, January 2005.

[5] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

[6] Chih-Ping Chen and Paul Hudak. Rolling your own mutable ADT—a connection between linear types and monads. In *POPL*, 1997.

[7] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL*, 1999.

[8] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *JFP*, 13(4), 2003.

[9] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, 2007.

[10] Matthew Fluet and Greg Morrisett. Monadic regions. *JFP*, 16(4–5):485–545, 2006.

[11] Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *ESOP*, volume 3924 of *LNCS*, March 2006.

[12] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, 2002.

[13] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, 1989.

[14] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *ESOP*, LNCS, March 2007.

[15] Peter W. O'Hearn and John C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, 2000.

[16] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[17] François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, 2008.

[18] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[19] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *ESOP*, volume 1782 of *LNCS*, 2000.

[20] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[21] Philip Wadler. The essence of functional programming. In *POPL*, 1992.

[22] David Walker and Greg Morrisett. Alias types for recursive data structures. In *TIC*, volume 2071 of *LNCS*, 2000.

[23] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *PADL*, volume 3350 of *LNCS*, 2005.