Typing Safe Deallocation

Gérard Boudol

INRIA, 06902 Sophia Antipolis, France

Abstract. In this work we address the problem of proving, by static analysis means, that allocating and deallocating regions in the store provides a safe way to achieve memory management. That is, the goal is to provably ensure that a program does not use pointers into a deallocated region. A well-known approach to this problem is the one of Tofte and Talpin. Our first contribution is to provide a simple proof, by means of a subject reduction property, of type safety for their region calculus. Our second, main contribution is that we actually do this for an extension of Tofte-Talpin's calculus, featuring a primitive construct for deallocating regions, similar to C's `free`, that allows one to circumvent the strict stack-of-regions discipline enforced in Tofte-Talpin's calculus. Our static analysis consists in a novel type and effect system, extending the one of Tofte and Talpin, where we record deallocation effects.

1 Introduction

Some years ago, Tofte and Talpin [15,16] proposed a new memory model for higher-order, typed languages, as an alternative to explicit allocation/deallocation of memory (e.g. `malloc/free` in C) and garbage collection. The main idea is to introduce, in an intermediate language for the compilation process, a block-structured (letregion ρ in e) construct, allocating a new region in the memory for the evaluation of e, and deallocating it upon termination.[1] Experience has shown that, as reported in [17], introducing region management in this way allows the compiler to produce code for ML programs that executes quite efficiently, even without the support of a garbage collector. Moreover, memory management with regions is *provably safe*, and this is one of the most remarkable achievements of Tofte and Talpin's approach.

The safety proof is not so easy, however. The original proof in [16] used a quite elaborate coinductive technique. A number of research works have been done since then on this topic [2,4,5,6,8,11,18], in order to better understand why the apparently simple typing of the letregion construct is actually safe, and to get a simpler proof. Indeed, this construct allows for reusing a dead region, even though there are still pointers in that region in the code, thus creating dangling pointers. Moreover, the same pointers can also be subsequently reused, holding

[1] This idea was previously mentioned in [14]: "*Since the locations belonging to a private region cannot be accessed after the expression returns, they can be safely deallocated when the expression returns.*" However, the semantics given in this paper did not involve deallocation, and this statement was not proved.

S. Drossopoulou (Ed.): ESOP 2008, LNCS 4960, pp. 116–130, 2008.

values of a different type. The difficulty is then to prove that the pointers in the code that actually refer to defunct regions are guaranteed by typing to never be used. The syntactic proof of [5], following the standard steps of a type safety proof (see [20]), seems to be adopted as the classical one by now (see [13]). It formalizes deallocation as the substitution of a dead region $\bullet$ for the deallocated region everywhere in the current configuration, including in the values stored in the memory. In [5] it is shown that this semantics is bisimilar to the original one of [16]. However, using an explicit notion ($\bullet$) of a deallocated region modifies (and overloads) the semantics of the letregion construct, with a rewriting phase that is not present in the original formulation, and therefore a truly direct, syntactic proof of type safety for Tofte-Talpin's region calculus is yet to be done. Our first contribution is to give a fresh look at this problem.

To prove that region deallocation is safe, we have to ensure that when a region is deallocated, it should not be used by the rest of the computation (i.e. the evaluation context). This is done in [16] by means of a sophisticated *consistency predicate*, supporting a coinductive proof technique. We shall do this here by means of a type and effect system. The idea is very simple. First, we decompose the (letregion ρ in e) construct of Tofte and Talpin as follows:

$$(\mathsf{letregion}\ \rho\ \mathsf{in}\ e) = (\mathsf{new}\ \rho\ \mathsf{in}\ (\mathsf{free\_after}\ \rho\ e))$$

where (new ρ in e') allocates a new region, with scope e', and (free_after ρ e) deallocates the region ρ when the computation of e is terminated. Such a decomposition was introduced in [1].[2] Then we introduce a new kind of effects, which we call negative effects, or *deallocation effects*, associated with the latter construct in the type system, and we check that there is no conflict with the ordinary, "positive," or *usage* effects, in order to ensure that regions required in future computations are not deleted in the current computation. Typically, assuming a left-to-right evaluation order, as in [16], the negative effect of e_0 in $(e_0 e_1)$ should not intersect the positive effect of e_1, in order for the application to be typable. With this refined (flow-sensitive) effect system, which directly extends the one of [16], we are able to show the safety of region deallocation, by means of a Subject Reduction argument, as explained below (Section 4).

The idea of using explicit deallocation effects suggests that we could further decompose the (letregion ρ in e) construct of Tofte and Talpin, introducing an explicit, atomic instruction for region deallocation. We write this as (dispose ρ), and we define

$$(\mathsf{free\_after}\ \rho\ e) = (\mathsf{let}\ x = e\ \mathsf{in}\ (\mathsf{dispose}\ \rho)\,;\, x)$$

(see [19] for a similar decomposition). Quite obviously, the deallocation effects are introduced by the dispose instruction, and our proof of type safety actually deals with the refined region calculus, where the letregion construct is replaced by new and dispose. The latter, which could also be defined in terms of free_after and termination (), was considered, in various concrete syntactic forms, in a

[2] In [1], the allocation operation is also separated from the creation of a new name. We shall comment on this below. The free_after construct is denoted (region ρ in e) in [5]. An analogous expression (∗private∗ ρ e) was used in [14], though with different semantics.

number of papers: it is denoted freerg in [8,18], deleteregion in [9,10] and release in [12] (with different semantics), and free in [19]. One reason for using such an explicit region deallocation construct is that, as noted very early in [1,3], the strict stack discipline enforced in Tofte and Talpin's calculus is too constraining to cope with situations where it would be much more efficient to reclaim a region without waiting for the end of its lexical scope. It is then natural to use such an explicit deallocation construct, for optimization purposes, but the problem of ensuring that this operation is safe comes out again, and it is a difficult one.

A specific difficulty, noted in [1], is that "*for correctness it is important that a region be allocated only once and deallocated only once during its lifetime.*" The "allocated-only-once" is a built-in feature of the (new ρ in e) construct, but regarding deallocation, it seems appropriate to use ideas from linear logic, and this is indeed what is done in a number of works, see for instance [7,8,18,19]. However, as we show below, resorting to linear logic techniques is not a necessity. Indeed, introducing deallocation effects makes it very easy to control the "deallocated-only-once" feature in our flow-sensitive effect system: it is enough to ensure that a region occurring in the deallocation effect of a subexpression is not in the (negative) effect of the rest of the computation. Our static analysis for provably ensuring the safety of explicit region deallocation is then much simpler than previously given ones.

The paper is organized as follows: in a first section, we introduce our extended region calculus, and describe its operational semantics. In a next section, we introduce our type and effect system, featuring the notion of a deallocation effect. We then establish a Subject Reduction property up to region renaming, and derive from it our Type Safety result. A brief conclusion is given. For lack of space, most of the proofs are omitted.

2 The Extended Region Calculus

In this section we introduce our region calculus, extending the one of Tofte and Talpin with an explicit primitive construct for deallocating regions, and we describe its operational semantics. We assume given two disjoint denumerable sets $\mathcal{R}eg\mathcal{V}ar$ and $\mathcal{R}eg\mathcal{C}st$ of *region variables* and *region constants*, respectively ranged over by ρ and r. The set $\mathcal{R}eg = \mathcal{R}eg\mathcal{V}ar \cup \mathcal{R}eg\mathcal{C}st$ of *region names*, is ranged over by ϱ. We also assume given a denumerable set $\mathcal{L}oc$ of *memory locations*, or *pointers*, range over by $p,\ q \ldots$, and a denumerable set $\mathcal{V}ar$ of *variables*, ranged over by $x,\ y, \ldots f,\ g \ldots$. The syntax is as follows:

$$
\begin{array}{lll}
a & := & (r, p) & \textit{addresses} \\
v & ::= & () \mid a & \textit{values} \\
w & ::= & \lambda x e & \textit{storable values} \\
e & ::= & x \mid v \mid (w \mathbin{@} \varrho) \mid (e_0 e_1) \mid (\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1) & \textit{expressions} \\
 & & \mid\ (\mathsf{new}\ \rho\ \mathsf{in}\ e) \mid (\mathsf{dispose}\ \varrho) &
\end{array}
$$

The expression $(w \mathbin{@} \varrho)$, pronounced "w at ϱ" in [16], is meant to create a new pointer in region ϱ with contents w. As usual, the variable x is bound in $\lambda x e$, and ρ is bound in (new ρ in e), whereas it is free in $(w \mathbin{@} \rho)$ and (dispose ρ).

We denote by $\mathsf{reg}(e)$ the set of region constants and region variables that occur (free) in e, and by $\mathsf{ref}(e)$ the set of addresses occurring in e. The expression e is said to be *closed* if no variable or region variable occurs free in it. We denote by $\{x \mapsto v\}e$ and $\{\rho \mapsto \varrho\}e$ the capture-avoiding substitutions of values and region names in e. We shall consider expressions up to α-conversion, that is up to the renaming of bound variables and regions. We use the notation $(\lambda x e_1 e_0)$, that is (we_0) where $w = \lambda x e_1$, as a synonym of $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$, and we also write this as $e_0\,;e_1$ whenever x is not free in e_1.

In a more realistic language, there would be more (storable) values, like booleans, integers, pairs, and so on, with appropriate constructs to use these values, like conditional branching, etc. We regard the region calculus of Tofte and Talpin [16] as a sub-language, where $(\mathsf{new}\ \rho\ \mathsf{in}\ e)$ and $(\mathsf{dispose}\ \varrho)$ are replaced by $(\mathsf{letregion}\ \rho\ \mathsf{in}\ e)$, with

$$(\mathsf{letregion}\ \rho\ \mathsf{in}\ e) =_{\mathrm{def}} (\mathsf{new}\ \rho\ \mathsf{in}\ (\mathsf{free\_after}\ \rho\ e)) \quad \text{where}$$
$$(\mathsf{free\_after}\ \varrho\ e) =_{\mathrm{def}} (\mathsf{let}\ x = e\ \mathsf{in}\ (\mathsf{dispose}\ \varrho)\,;x)$$

as explained in the Introduction.

In order to show our safety result, we use a small-step semantics for the language. The evaluation of an expression consists, as usual, in reducing a *redex* inside an *evaluation context*, in the context of a *store*. The redexes and evaluation contexts are as follows:

$$\begin{array}{lll} u ::= (w \mathbin{@} r) \mid (av) \mid (\lambda x e v) & & \textit{redexes} \\ \quad \mid (\mathsf{new}\ \rho\ \mathsf{in}\ e) \mid (\mathsf{dispose}\ r) & & \\ \mathbf{E} ::= [\,] \mid \mathbf{E}[\mathbf{F}] & & \textit{evaluation contexts} \\ \mathbf{F} := ([\,]\,e) \mid (a\,[\,]) \mid (\lambda x e\,[\,]) & & \textit{frames} \end{array}$$

Definition (Stuck Expressions) 2.1. *An expression e is* stuck *if and only if $e = \mathbf{E}[e']$ where e' is either a variable, or $(w \mathbin{@} \rho)$, or $(()e')$, or $(\mathsf{dispose}\ \rho)$.*

Notice that a closed stuck expression has the form $\mathbf{E}[(()e)]$. The following is a standard fact:

Lemma 2.2. *For any expression e, either*

(i) *e is a value, or*

(ii) *e is a stuck expression, or*

(iii) *there exist an evaluation context $\mathbf{E}$ and a redex u such that $e = \mathbf{E}[u]$.*

As in [16], a *store* s is a mapping from a finite set $\mathsf{dom}(s)$ of region constants to *regions*, where a region is a mapping from a finite set of locations to storable values. We denote by $\mathsf{Dom}(s)$ the set $\{\,(r,p) \mid r \in \mathsf{dom}(s)\ \&\ p \in \mathsf{dom}(s(r))\,\}$, and we write $s(r,p)$ for $s(r)(p)$ where $(r,p) \in \mathsf{Dom}(s)$. We define, for $R \subseteq \mathcal{R}eg$ and $r \in \mathcal{R}eg\mathcal{C}st$:

$$\mathsf{dom}(s \restriction R) = R \cap \mathsf{dom}(s)$$
$$r \in \mathsf{dom}(s \restriction R) \ \Rightarrow\ (s \restriction R)(r) = s(r)$$
$$s \backslash r = s \restriction (\mathsf{dom}(s) - \{r\})$$

We shall in fact use the notations $f \restriction X$ and $f \backslash x$ for any partial function $f : A \rightharpoonup B$, with $x \in A$ and $X \subseteq A$.

$$
\begin{array}{rll}
(s, \mathbf{E}[(w \mathbin{@} r)]) \to (s + \{(r,p) \mapsto w\}, \mathbf{E}[(r,p)]) & & r \in \mathsf{dom}(s), \\
& & p \notin \mathsf{dom}(s(r)) \\
(s, \mathbf{E}[((r,p)v)]) \to (s, \mathbf{E}[(wv)]) & & r \in \mathsf{dom}(s), \\
& & p \in \mathsf{dom}(s(r)), \\
& & s(r,p) = w \\
(s, \mathbf{E}[(\lambda x e v)]) \to (s, \mathbf{E}[\{x \mapsto v\}e]) & & \\
(s, \mathbf{E}[(\mathsf{new}\ \rho\ \mathsf{in}\ e)]) \to (s + \{r \mapsto \emptyset\}, \mathbf{E}[\{\rho \mapsto r\}e]) & & r \notin \mathsf{dom}(s) \\
(s, \mathbf{E}[(\mathsf{dispose}\ r)]) \to (s \backslash r, \mathbf{E}[()]) & &
\end{array}
$$

Fig. 1. Reduction

In the operational semantics, we use the notations of [16] for extending or updating the store with new regions, namely $s + \{r \mapsto \emptyset\}$ and $s + \{(r,p) \mapsto w\}$. The reduction relation consists in a transition relation between *configurations*, that are pairs (s, e) of a store and an expression to evaluate. This is defined in Figure 1. The evaluation of an application $(e_0 e_1)$ is standard. We nevertheless examine the various steps in details, since our typing will rely on these: first, one computes the function e_0 until an address a is obtained, possibly by reducing an expression $(w \mathbin{@} r)$. Next, the argument e_1 is computed, producing a value v. Then, to evaluate the resulting expression (av), a read operation occurs, returning the value contained in the store at address a. This value should be a function $\lambda x e$, and we now have to evaluate $(\mathsf{let}\ x = v\ \mathsf{in}\ e)$, as usual, that is: the value v is bound to x, and finally one proceeds evaluating $\{x \mapsto v\}e$. Regarding the construct $(\mathsf{new}\ \rho\ \mathsf{in}\ e)$, evaluating it consists in allocating a new region constant r in the store, which is bound to ρ in e for the rest of the computation,[3] while evaluating $(\mathsf{dispose}\ r)$ deallocates the region named r from the store and terminates. Then one can check that the $(\mathsf{letregion}\ \rho\ \mathsf{in}\ e)$ construct has the same semantics as in [16]. Notice in particular that in allocating a new region, reducing $(\mathsf{new}\ \rho\ \mathsf{in}\ e)$, we do not require that the new name does not occur in e, nor in the evaluation context $\mathbf{E}$, nor in some value currently recorded in the store. Then one can reuse a region name that still occurs in the configuration, with the only proviso that the name is not in the domain of the current store.

3 The Type and Effect System

3.1 Effects, Types, Judgements and Rules

Our main technical novelty in this work consists, as explained in the Introduction, in refining the notion of an effect, introducing negative, *deallocation effects* that are distinct from the usual "positive" effects of creating, reading or updating a region. In this work, it will be unnecessary to distinguish various kinds of positive effects. Then an *effect* here is a pair $\varphi = (\varphi^+, \varphi^-)$ of a positive effect φ^+ and a negative effect φ^-, which both are finite sets of region names. The standard

[3] For simplicity, we use region substitution $\{\rho \mapsto r\}e$ instead of a region environment.

$$\frac{}{\Sigma;\Gamma,x:\tau\vdash x:\emptyset,\tau} \qquad \frac{}{\Sigma;\Gamma\vdash():\emptyset,\mathbb{1}} \qquad \frac{}{\Sigma,(r,p):\zeta;\Gamma\vdash(r,p):\emptyset,\zeta_r}$$

$$\frac{\Sigma;\Gamma,x:\tau\vdash e:\varphi,\sigma}{\Sigma;\Gamma\vdash\lambda xe:(\tau\xrightarrow{\varphi}\sigma)} \qquad \frac{\Sigma;\Gamma\vdash w:\zeta}{\Sigma;\Gamma\vdash(w@\varrho):(\{\varrho\},\emptyset),\zeta_\varrho}$$

$$\frac{\Sigma;\Gamma\vdash e_0:\varphi_0,(\tau\xrightarrow{\varphi_2}\sigma)_\varrho \quad \Sigma;\Gamma\vdash e_1:\varphi_1,\tau}{\Sigma;\Gamma\vdash(e_0e_1):(\varphi_0+\varrho)\cup\varphi_1\cup\varphi_2,\sigma} \quad \begin{cases}\varphi_0^-\cap(\{\varrho\}\cup\varphi_1^\pm\cup\varphi_2^\pm)=\emptyset\\ \varphi_1^-\cap(\{\varrho\}\cup\varphi_2^\pm)=\emptyset\end{cases}$$

$$\frac{\Sigma;\Gamma\vdash e_0:\varphi_0,\tau \quad \Sigma;\Gamma,x:\tau\vdash e_1:\varphi_1,\sigma}{\Sigma;\Gamma\vdash(\mathsf{let}\ x=e_0\ \mathsf{in}\ e_1):\varphi_0\cup\varphi_1,\sigma}\ \varphi_0^-\cap\varphi_1^\pm=\emptyset$$

$$\frac{\Sigma;\Gamma\vdash e:\varphi,\tau}{\Sigma;\Gamma\vdash(\mathsf{new}\ \rho\ \mathsf{in}\ e):\varphi\backslash\rho,\tau}\ \rho\notin\Sigma,\Gamma,\tau \qquad \frac{}{\Sigma;\Gamma\vdash(\mathsf{dispose}\ \varrho):(\emptyset,\{\varrho\}),\mathbb{1}}$$

Fig. 2. The type and effect system: expressions

set-theoretic notions, like inclusion, union, etc. are extended componentwise to effects. In the following we write $\varphi^\pm$ for $\varphi^+\cup\varphi^-$, $\varphi+\varrho$ for $\varphi\cup(\{\varrho\},\emptyset)$, and $\varphi\backslash\varrho$ for $\varphi-(\{\varrho\},\{\varrho\})$. The types are standard:

$$\begin{array}{rll}\tau,\ \sigma\ldots ::= t \mid \mathbb{1} \mid \zeta_\varrho & & \textit{types}\\ \zeta ::= (\tau\xrightarrow{\varphi}\sigma) & & \textit{storable value types}\end{array}$$

The type $\mathbb{1}$ is also often denoted unit. As in [16], $(\tau\xrightarrow{\varphi}\sigma)_\varrho$ is the type of addresses in region ϱ of the store where one finds a functional value of type $(\tau\xrightarrow{\varphi}\sigma)$. As usual, a functional type records the latent effect φ a function of this type may have when applied to an argument.

There are two kinds of judgments in our type and effect system. A judgment $\Sigma;\Gamma\vdash e:\varphi,\tau$ means that under the assumptions Σ and Γ, the expression e is anticipated to have an effect φ, and has type τ. Similarly, a judgment $\Sigma;\Gamma\vdash w:\zeta$ means that, under the assumptions Σ and Γ, the storable value w has type ζ (and no effect, since this is a value). The component Σ in these judgments is the region typing context, which maps a finite set $\mathsf{dom}(\Sigma)$ of region constants to region typings, where a region typing is a map from a finite set of locations to types of storable values. The set $\{\,(r,p)\mid r\in\mathsf{dom}(\Sigma)\ \&\ p\in\mathsf{dom}(\Sigma(r))\,\}$ is denoted $\mathsf{Dom}(\Sigma)$, and a region typing context is written $(r_1,p_1):\zeta_1,\ldots,(r_m,p_m):\zeta_m$. The Γ component is, as usual, a typing context, mapping a finite set of variables to types. In the typing rule for $(\mathsf{new}\ \rho\ \mathsf{in}\ e)$, we write $\rho\notin\Sigma,\Gamma,\tau$ to mean that the variable ρ does not occur in Σ, Γ (that is, in the types assigned by these typing contexts) and τ.

The rules of the type and effect system are given in Figure 2, which we now comment. First we point out that the negative effects are, as expected, introduced when typing an expression $(\mathsf{dispose}\ \varrho)$, while a positive effect results from a storing operation $(w@\varrho)$ and reading a (functional) value from the store, in an application. Our effect system then checks that a subexpression does not deallocate a region in which some future effect is anticipated. In our core language,

where we adopt a left-to-right evaluation order, the only subexpressions that have a "future" are e_0 in $(e_0 e_1)$ and $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$, and e_1 in $(e_0 e_1)$, where in the latter case, the effects that may arise after evaluating e_1 are the effect of reading the function from the store (at address e_0), and the latent effect of that function. Then in typing the application $(e_0 e_1)$, we have the constraint that the region in which the value resulting from evaluating e_0 is stored should not be disposed of before the actual reading operation occurs, that is $\varrho \notin (\varphi_0^- \cup \varphi_1^-)$. Similarly, e_0 should not have the effect of removing regions that may be used in the rest of the computation, that is $\varphi_0^- \cap \varphi_1^{\pm} = \emptyset = \varphi_0^- \cap \varphi_2^{\pm}$, and finally, e_1 should not delete regions occurring in the latent effect of the function, that is $\varphi_1^- \cap \varphi_2^{\pm} = \emptyset$. The constraint in typing a $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$ is similar. These constraints mean in particular that one cannot deallocate twice the same region. Indeed, in our calculus where allocating (via new) and deallocating (via dispose) a region are not restricted to follow the strict block-structured discipline of [16], it would be generally unsafe (and not very useful) to deallocate several times the same region. For instance, evaluating an expression of the form

$$(\mathsf{new}\,\rho_0 \cdots \mathsf{dispose}\,\rho_0 \cdots (\mathsf{new}\,\rho_1 \cdots \mathsf{dispose}\,\rho_0 \cdots (w \mathbin{@} \rho_1) \cdots))$$

could result in assigning to ρ_1 the same region r that has been assigned to ρ_0, since r has been disposed of, but then the second instruction $(\mathsf{dispose}\ \rho_0)$ has the effect of deleting the region associated with ρ_1, and the evaluation of $(w \mathbin{@} \rho_1)$ then fails in this case. In the rule for $(\mathsf{new}\ \rho\ \mathsf{in}\ e)$, we could require $\rho \in \varphi^-$, in order to ensure that the region assigned to ρ has been disposed of when the evaluation exits its scope, but this would be just an indication, because the effects anticipated by typing are not guaranteed to occur (though it is guaranteed that no other effect can possibly occur).

Regarding the derived constructs that are involved in the Tofte and Talpin's sub-calculus, one can see that a derived typing rule is

$$\frac{\Sigma; \Gamma \vdash e : \varphi, \tau}{\Sigma; \Gamma \vdash (\mathsf{free\_after}\ \varrho\ e) : \varphi \cup (\emptyset, \{\varrho\}), \tau}\ \varrho \notin \varphi^-$$

and consequently

$$\frac{\Sigma; \Gamma \vdash e : \varphi, \tau}{\Sigma; \Gamma \vdash (\mathsf{letregion}\ \rho\ \mathsf{in}\ e) : \varphi \backslash \rho, \tau}\ \rho \notin \Sigma, \Gamma, \tau, \varphi^-$$

One may then observe that in typing expressions of the derived sub-calculus the negative effect is always empty, and conclude that, up to the identification of $(\varphi^+, \emptyset)$ with the single effect φ^+, what we get is exactly the usual typing for Tofte and Talpin's region calculus, without any constraint on the effect.

Now let us see an example of a typable expression, inspired from examples in [1,18]. Let w be a typable storable value, e a typable expression using (via the variable x) this value from region ρ (and possibly having other positive effects in this region), and let e' be a typable expression that has no effect in region ρ. Then the following is typable:

$$\begin{array}{l}\mathsf{new}\ \rho\ \mathsf{in}\ \mathsf{let}\ x = (w \mathbin{@} \rho)\ \mathsf{in}\\ \mathsf{new}\ \rho'\ \mathsf{in}\ \mathsf{let}\ f = (\lambda x(\mathsf{let}\ y = (\mathsf{free\_after}\ \rho\ e)\ \mathsf{in}\ e') \mathbin{@} \rho')\ \mathsf{in}\\ (\mathsf{free\_after}\ \rho'\ (f x))\end{array}$$

This example shows, first, that regions may have arbitrarily overlapping extent [1]: here the evaluation will execute the sequence

$$\mathsf{new}\ \rho \cdots \mathsf{new}\ \rho' \cdots \mathsf{dispose}\ \rho \cdots \mathsf{dispose}\ \rho'$$

Second, in the code for the function f, the region constant assigned to ρ can be disposed of without waiting for the call (fx) to end, since this region is only used in a first part of the computation of (fx). As another example, one can see that with a conditional branching construct, typed as follows:

$$\frac{\Sigma; \Gamma \vdash e : \varphi, \mathsf{bool} \quad \Sigma; \Gamma \vdash e_i : \varphi_i, \tau}{\Sigma; \Gamma \vdash (\mathsf{if}\ e\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) : \varphi \cup \varphi_0 \cup \varphi_1, \tau} \ \varphi^- \cap (\varphi_0^\pm \cup \varphi_1^\pm) = \emptyset$$

then if a branch does not use region ϱ, one can immediately dispose of it, while in the other branch this action is deferred after the use of values in that region. (As above with the new construct, we could additionally require $\varphi_0^- = \varphi_1^-$ in this rule.)

To show the type safety result, we have to extend the typing to configurations. In order to type the store, one should have enough assumptions in the region typing context: any address in the store should be the subject of a typing assumption. Moreover, the value stored at some address should have type as prescribed by the region typing context. Finally, for a configuration to be typable, we shall require a "well-formedness" property, asserting that any region in which the computation may have an effect should be allocated in the store. Indeed, it is essential for safety to preserve the property that accesses to the memory never fail. Our definition is therefore as follows:

Definition (Typing Configurations) 3.1.

$$\text{(i)} \quad \Sigma; \Gamma \vdash s \quad \Leftrightarrow_{\text{def}} \quad \begin{cases} \mathsf{Dom}(s) \subseteq \mathsf{Dom}(\Sigma) \\ (r,p) \in \mathsf{Dom}(s) \ \Rightarrow \ \Sigma; \Gamma \vdash s(r,p) : \Sigma(r,p) \end{cases}$$

$$\text{(ii)} \quad \Sigma; \Gamma \vdash (s,e) : \varphi, \tau \quad \Leftrightarrow_{\text{def}} \quad \begin{cases} \Sigma; \Gamma \vdash s \ \&\ \Sigma; \Gamma \vdash e : \varphi, \tau \\ \forall r.\ r \in \varphi^\pm \ \Rightarrow \ \begin{cases} r \in \mathsf{dom}(s)\ \& \\ \mathsf{dom}(\Sigma(r)) \subseteq \mathsf{dom}(s(r)) \end{cases} \end{cases}$$

3.2 Some Properties

We notice a few facts that will be used in our proof of type safety. First, the type and effect system reflects the fact that a value has no effect:

Remark 3.2. $\Sigma; \Gamma \vdash v : \varphi, \tau \ \Rightarrow \ \varphi = \emptyset$

Second, some errors are, as usual, statically precluded by typing:

Remark 3.3. *A closed stuck expression is not typable.*

Finally, one can show some standard properties relating typing and substitution:

Lemma (Substitution) 3.4.

(i) $\Sigma; \Gamma \vdash v : \psi, \tau \ \&\ \Sigma; \Gamma, x : \tau \vdash e : \varphi, \sigma \ \Rightarrow \ \Sigma; \Gamma \vdash \{x \mapsto v\}e : \varphi, \sigma$

(ii) *If* $\Sigma; \Gamma \vdash e : \varphi, \tau$ *and* r *does not occur in* $\Sigma; \Gamma \vdash e : \varphi, \tau$ *then* $\{\rho \mapsto r\}(\Sigma; \Gamma \vdash e : \varphi, \tau)$.

$$\frac{}{\Sigma;\Gamma\vdash[]:(\tau\xrightarrow{\emptyset}\tau)} \qquad \frac{\Sigma;\Gamma\vdash\mathbf{E}:(\theta\xrightarrow{\varphi_1}\sigma)\quad\Sigma;\Gamma\vdash\mathbf{F}:(\tau\xrightarrow{\varphi_0}\theta)}{\Sigma;\Gamma\vdash\mathbf{E}[\mathbf{F}]:(\tau\xrightarrow{\varphi_0\cup\varphi_1}\sigma)}\ \varphi_0^-\cap\varphi_1^\pm=\emptyset$$

$$\frac{\Sigma;\Gamma\vdash e:\varphi_0,\tau}{\Sigma;\Gamma\vdash([]e):((\tau\xrightarrow{\varphi_1}\sigma)_\varrho\xrightarrow{(\varphi_0+\varrho)\cup\varphi_1}\sigma)}\ \varphi_0^-\cap(\{\varrho\}\cup\varphi_1^\pm)=\emptyset$$

$$\frac{}{\Sigma,(r,p):(\tau\xrightarrow{\varphi}\sigma);\Gamma\vdash((r,p)\,[]):(\tau\xrightarrow{\varphi+r}\sigma)} \qquad \frac{\Sigma;\Gamma,x:\tau\vdash e:\varphi,\sigma}{\Sigma;\Gamma\vdash(\lambda xe\,[]):(\tau\xrightarrow{\varphi}\sigma)}$$

Fig. 3. The type and effect system: evaluation contexts

For the proof of our main result, it will be convenient to decompose the typing of an expression of the form $\mathbf{E}[e]$ into a typing of e and a typing of the evaluation context. (An alternative is to use a "Replacement Lemma", see [20] for instance.) The type system for evaluation contexts allows one to infer judgments of the form $\Sigma;\Gamma\vdash\mathbf{E}:(\tau\xrightarrow{\varphi}\sigma)$, meaning that if the context is filled with an expression of type τ, then it will return a result of type σ, while producing effects as indicated by φ. There are constraints regarding the effects similar to the ones that hold for expressions. The rules are given in Figure 3. Then we introduce an alternative way to type expressions, by means of judgments of the form $\Sigma;\Gamma\Vdash e:\varphi,\tau$, established as follows:

$$\frac{\Sigma;\Gamma\vdash e:\varphi_0,\tau\quad\Sigma;\Gamma\vdash\mathbf{E}:(\tau\xrightarrow{\varphi_1}\sigma)}{\Sigma;\Gamma\Vdash\mathbf{E}[e]:\varphi_0\cup\varphi_1,\sigma}\ \varphi_0^-\cap\varphi_1^\pm=\emptyset$$

We can prove that this provides us with just an equivalent way of typing:

Lemma 3.5.

(i) $\Sigma;\Gamma\Vdash e:\varphi,\tau\ \Rightarrow\ \Sigma;\Gamma\vdash e:\varphi,\tau$

(ii) *If* $\Sigma;\Gamma\vdash\mathbf{E}[e]:\varphi,\tau$ *then there exist* φ_0, φ_1 *and* σ *such that* $\Sigma;\Gamma\vdash e:\varphi_0,\sigma$ *and* $\Sigma;\Gamma\vdash\mathbf{E}:(\sigma\xrightarrow{\varphi_1}\tau)$ *with* $\varphi=\varphi_0\cup\varphi_1$ *and* $\varphi_0^-\cap\varphi_1^\pm=\emptyset$.

4 Type Safety

A technical difficulty in showing the soundness of typing deallocation is that there is a discrepancy between the operational semantics and typing as regards the generation of new regions. More specifically, to establish Subject Reduction would require that the fresh name generated when reducing (new ρ in e) be as fresh as possible, and in particular, that it does not occur in the expression e, in order for the substitution of the new name to yield a valid typing judgment. On the opposite side, from the operational point of view, it could be beneficial, and therefore allowed (as it is), to reuse a name that has been disposed of, even though it could still occur in the expression e, in a dead pointer to a deallocated region for instance. Our way to reconcile the typing with the operational

semantics is to establish a Subject Reduction property *up to simulation* (where a "simulation" is half a bisimulation – but we do not need any coinductive machinery). The idea is actually very simple: it is to show that, to preserve the typing along a given computation, one may have to, not exactly follow, but simulate the actual computation by just making "better" (from the typing point of view) choices of new region names and pointers, while still maintaining a tight correspondence with the given computation, by means of a region and pointer renaming. Safety will then result from the fact that the use of dangling pointers is precluded by typing, *cf.* Definition 3.1(ii).

4.1 The Simulation Relation

We introduce a relation over configurations (s, e), that will be proved to be a simulation. More precisely our simulation relates (s, e) to (s', e') by means of a *translation* $\mathbf{t}$, in such a way that, if (s, e) is typable and (s', e') performs a transition, then one can choose regions and pointers so that (s, e) performs a similar transition, resulting in similar configurations, while preserving typability. The translation, relating region constants to regions constants, and pointers to pointers, may evolve along the transitions, either because a new pointer is created, or because a region constant is created, or reused.

Definition (Translations) 4.1. *A* translation $\mathbf{t}$ *is a pair* $(\mathbf{r}, \mathbf{p})$ *where*

(i) $\mathbf{r}$ *is a mapping from a finite subset* $\mathsf{dom}(\mathbf{r})$ *of* $\mathcal{R}eg\mathcal{C}st$ *to* $\mathcal{R}eg\mathcal{C}st$,

(ii) $\mathbf{p}$ *is a function with the same domain as* $\mathbf{r}$, *such that, for any* $r \in \mathsf{dom}(\mathbf{r})$, $\mathbf{p}(r)$ *is an injective mapping from a finite subset* $\mathsf{dom}(\mathbf{p}(r))$ *of* $\mathcal{L}oc$ *to* $\mathcal{L}oc$.

We denote by $\mathcal{T}$ the set of translations. We also write $\mathbf{p}(r)$ as $\mathbf{p}_r$. We extend the inclusion relation to translations, as follows:

$$(\mathbf{r}, \mathbf{p}) \subseteq (\mathbf{r}', \mathbf{p}') \quad \Leftrightarrow_{\text{def}} \quad \begin{cases} \mathbf{r} \subseteq \mathbf{r}' \ \& \\ r \in \mathsf{dom}(\mathbf{r}) \ \Rightarrow \ \mathbf{p}_r \subseteq \mathbf{p}'_r \end{cases}$$

For each translation $\mathbf{t} = (\mathbf{r}, \mathbf{p})$, we define the partial mapping $\langle \mathbf{t} \rangle$ on expressions and storable values, by induction on the structure, as follows – omitting the cases where the translation just goes through the structure of the expression:

$$\begin{aligned} \langle \mathbf{t} \rangle(r, p) &= (\mathbf{r}(r), \mathbf{p}_r(p)) && \text{if } r \in \mathsf{dom}(\mathbf{r}) \ \& \ p \in \mathsf{dom}(\mathbf{p}_r) \\ \langle \mathbf{t} \rangle(w \mathbin{@} r) &= (\langle \mathbf{t} \rangle w \mathbin{@} \mathbf{r}(r)) && \text{if } r \in \mathsf{dom}(\mathbf{r}) \\ \langle \mathbf{t} \rangle(\mathsf{dispose}\ r) &= (\mathsf{dispose}\ \mathbf{r}(r)) && \text{if } r \in \mathsf{dom}(\mathbf{r}) \end{aligned}$$

We write:

$$e \rhd_{\mathbf{t}} e' \quad \Leftrightarrow_{\text{def}} \quad e \in \mathsf{dom}(\langle \mathbf{t} \rangle) \ \& \ e' = \langle \mathbf{t} \rangle(e)$$

The syntactic structure of e' is identical to the one of e whenever $e \rhd_{\mathbf{t}} e'$: the expression e' is obtained from e by renaming region constants and pointers. The following should be obvious:

Remarks 4.2.

(i) *For any expression* e, *if we let* $\mathbf{t}_e = (\mathbf{r}, \mathbf{p})$ *where* $\mathbf{r} = \{ (r, r) \mid r \in \mathsf{reg}(e) \}$ *and* $\mathbf{p}_r = \{ (p, p) \mid (r, p) \in \mathsf{ref}(e) \}$ *for* $r \in \mathsf{reg}(e)$, *then* $\mathbf{t}_e \in \mathcal{T}$ *and* $e \rhd_{\mathbf{t}_e} e$.

(ii) *If* $e \in \mathsf{dom}(\langle\mathbf{t}\rangle)$ *then* $\langle\mathbf{t}\rangle e$ *is a value (resp. a redex, resp. a stuck expression) if and only if* e *is a value (resp. a redex, resp. a stuck expression).*
(iii) *If* $e \rhd_{\mathbf{t}} e'$ *and* $\mathbf{t} \subseteq \mathbf{t}'$ *then* $e \rhd_{\mathbf{t}'} e'$.

The relation $\rhd_{\mathbf{t}}$ is compatible with substitution:

Lemma 4.3.
(i) $v \rhd_{\mathbf{t}} v' \ \&\ e \rhd_{\mathbf{t}} e' \ \Rightarrow\ \{x \mapsto v\}e \rhd_{\mathbf{t}} \{x \mapsto v'\}e'$
(ii) $r \in \mathsf{dom}(\mathbf{r}) \ \&\ e \rhd_{\mathbf{r},\mathbf{p}} e' \ \Rightarrow\ \{\rho \mapsto r\}e \rhd_{\mathbf{r},\mathbf{p}} \{\rho \mapsto \mathbf{r}(r)\}e'$.

We define what it means for a translation $\mathbf{t}$ to *comply* with an effect, which intuitively means that the translation does not confuse the region constants involved in the effect:

Definition 4.4. *A translation* $\mathbf{t} = (\mathbf{r}, \mathbf{p})$ complies *with the effect* φ*, in notation* $\mathbf{t} \propto \varphi$ *if and only if* $\varphi^{\pm} \cap \mathcal{R}eg\mathcal{C}st \subseteq \mathsf{dom}(\mathbf{r})$ *and* $\mathbf{r} \restriction \varphi^{\pm}$ *is injective.*

Clearly

$$\mathbf{t} \propto \varphi \ \&\ \psi \subseteq \varphi \ \Rightarrow\ \mathbf{t} \propto \psi \tag{1}$$

Our simulation on configurations is indexed by a translation $\mathbf{t}$ and an effect φ. We first define the relation $\rhd^{\varphi}_{\mathbf{t}}$ on stores, as follows:

Definition 4.5. *Let* $\mathbf{t} = (\mathbf{r}, \mathbf{p})$ *be a translation and* φ *an effect such that* $\mathbf{t} \propto \varphi$. *Then* $s \rhd^{\varphi}_{\mathbf{t}} s'$*, read "*$s$* simulates* s' up to φ modulo $\mathbf{t}$*," if and only if*
(i) $\mathbf{r}(\varphi^{\pm}) \subseteq \mathsf{dom}(s')$
(ii) $r \in \varphi^{+} \ \&\ (r,p) \rhd_{\mathbf{t}} (r',p') \ \Rightarrow\ \begin{cases} (r,p) \in \mathsf{Dom}(s) \Leftrightarrow (r',p') \in \mathsf{Dom}(s') \\ (r,p) \in \mathsf{Dom}(s) \ \Rightarrow\ s(r,p) \rhd_{\mathbf{t}} s'(r',p') \end{cases}$

It should be obvious that

$$s \rhd^{\varphi}_{\mathbf{t}} s' \ \&\ \psi \subseteq \varphi \ \Rightarrow\ s \rhd^{\psi}_{\mathbf{t}} s' \tag{2}$$

Then we define

$$(s,e) \rhd^{\varphi}_{\mathbf{t}} (s',e') \ \Leftrightarrow_{\mathrm{def}}\ s \rhd^{\varphi}_{\mathbf{t}} s' \ \&\ e \rhd_{\mathbf{t}} e'$$

4.2 Main Result

Now we show the Subject Reduction property suggested above: if (s_0, e_0) is typable, and simulates (s_1, e_1), and if the latter performs a transition to (s'_1, e'_1) then there is a choice of regions and a typable configuration (s'_0, e'_0) which simulates (modulo the updated region translation) (s'_1, e'_1), and is the result of the corresponding transition from (s_0, e_0). This property can be drawn:

$$\begin{array}{ccc} (s_1,e_1) & \longrightarrow & (s'_1,e'_1) \\ \Delta & & \Delta \\ \Sigma;\Gamma \vdash (s_0,e_0) : \varphi,\tau & \dashrightarrow & \Sigma';\Gamma \vdash (s'_0,e'_0) : \psi,\tau \end{array}$$

Lemma (Subject Reduction up to Simulation) 4.6.
If $\Sigma; \Gamma \vdash (s_0, e_0) : \varphi, \tau$ *and* $(s_1, e_1) \to (s_1', e_1')$ *with* $(s_0, e_0) \rhd_{\mathbf{t}}^{\varphi} (s_1, e_1)$ *then there exist* s_0', e_0', Σ' *and* ψ *such that* $(s_0, e_0) \to (s_0', e_0')$ *and* $\Sigma'; \Gamma \vdash (s_0', e_0') : \psi, \tau$ *with* $(s_0', e_0') \rhd_{\mathbf{t}'}^{\psi} (s_1', e_1')$ *for some* $\mathbf{t}' \in \mathcal{T}$.
Proof: by case on the transition $(s_1, e_1) \to (s_1', e_1')$.

• $(s_1, \mathbf{E}_1[(w' @ r')]) \to (s_1 + \{(r', p') \mapsto w'\}, \mathbf{E}_1[(r', p')])$ with $r' \in \mathsf{dom}(s_1)$ and $p' \notin \mathsf{dom}(s_1(r'))$. We have $e_0 = \mathbf{E}_0[(w @ r)]$ with $r \in \mathsf{dom}(\mathbf{r})$, $r' = \mathbf{r}(r)$ and $w \rhd_{\mathbf{t}} w'$. By Lemma 3.5(ii), there exist φ_0, φ_1 and σ such that $\Sigma; \Gamma \vdash (w @ r) : \varphi_0, \sigma$ and $\Sigma; \Gamma \vdash \mathbf{E}_0 : (\sigma \xrightarrow{\varphi_1} \tau)$ with $\varphi = \varphi_0 \cup \varphi_1$ and $\varphi_0^- \cap \varphi_1^{\pm} = \emptyset$. Then $\sigma = \zeta_r$ with $\Sigma; \Gamma \vdash w : \zeta$, and $\varphi_0 = (\{r\}, \emptyset)$. We have $r \in \varphi^+$, and therefore $r \in \mathsf{dom}(s_0)$ by Definition 3.1(ii). We distinguish two cases.
(a) If there exists p such that $\mathbf{p}_r(p) = p'$, that is $(r, p) \rhd_{\mathbf{t}} (r', p')$ then $p \notin \mathsf{dom}(s_0(r))$ by Definition 4.5(ii), and therefore

$$(s_0, e_0) \to (s_0', e_0') \quad \text{where} \quad \begin{cases} s_0' = s_0 + \{(r, p) \mapsto \lambda x e\} \\ e_0' = \mathbf{E}_0[(r, p)] \end{cases}$$

Since $r \in \varphi^+$ and $(r, p) \notin \mathsf{Dom}(s_0)$ we have, by Definition 3.1(ii), $(r, p) \notin \mathsf{Dom}(\Sigma)$, and $\Sigma, (r, p) : \zeta; \Gamma \vdash (r, p) : \emptyset, \zeta_r$. Then $\Sigma, (r, p) : \zeta; \Gamma \vdash (s_0', e_0') : \varphi_1, \tau$ by Lemma 3.5(i) and $\varphi_1 \subseteq \varphi$. Then obviously $\mathbf{t} \propto \varphi_1$ (see (1) above), and

$$\mathbf{r}(\varphi_1^{\pm}) \subseteq \mathbf{r}(\varphi^{\pm}) \subseteq \mathsf{dom}(s_1) = \mathsf{dom}(s_1')$$

by Definition 4.5(i). If $r'' \in \varphi_1^+$ and $(r'', p'') \rhd_{\mathbf{t}} (r', p')$ then $r'' = r$ since $\mathbf{r} \restriction \varphi^{\pm}$ is injective, and $p'' = p$ since $\mathbf{p}_r$ is injective, hence $(r'', p'') \in \mathsf{Dom}(s_0')$. From this we easily conclude $(s_0', e_0') \rhd_{\mathbf{t}}^{\varphi_1} (s_1', e_1')$.
(b) Otherwise, that is if there is no $p \in \mathsf{dom}(\mathbf{p}_r)$ such that $\mathbf{p}_r(p) = p'$, let p be such that $(r, p) \notin \mathsf{Dom}(\Sigma)$. Then $(r, p) \notin \mathsf{Dom}(s_0)$ by Definition 3.1(i), and therefore

$$(s_0, e_0) \to (s_0', e_0') \quad \text{where} \quad \begin{cases} s_0' = s_0 + \{(r, p) \mapsto \lambda x e\} \\ e_0' = \mathbf{E}_0[(r, p)] \end{cases}$$

Since $\Sigma, (r, p) : \zeta; \Gamma \vdash (r, p) : \emptyset, \zeta_r$ we have $\Sigma, (r, p) : \zeta; \Gamma \vdash (s_0', e_0') : \varphi_1, \tau$ by Lemma 3.5(i). Let $\mathbf{t}' = (\mathbf{r}, \mathbf{p}')$ where $\mathbf{p}' = \mathbf{p} + \{(r, p) \mapsto (r', p')\}$. Then $\mathbf{p}_r'$ is injective, and since $\varphi_1 \subseteq \varphi$ we have $\mathbf{t}' \propto \varphi_1$ (see (1) above) and $\mathbf{r}(\varphi_1^{\pm}) \subseteq \mathbf{r}(\varphi^{\pm}) \subseteq \mathsf{dom}(s_1) = \mathsf{dom}(s_1')$. There is no (r'', p'') such that $r'' \in \varphi_1^+$ and $(r'', p'') \rhd_{\mathbf{t}} (r', p')$, since otherwise we would have $r'' = r$, for $\mathbf{r} \restriction \varphi^{\pm}$ is injective, by Definition 4.4, and this would contradict our assumption (b). From this it is easy to conclude $(s_0', e_0') \rhd_{\mathbf{t}'}^{\varphi_1} (s_1', e_1')$, using Remark 4.2(iii).

• $(s_1, \mathbf{E}_1[((r', p')v')]) \to (s_1, \mathbf{E}_1[(w'v')])$ with $r' \in \mathsf{dom}(s_1)$, $p' \in \mathsf{dom}(s_1(r'))$ and $s_1(r', p') = w'$. We have $e_0 = \mathbf{E}_0[((r, p)e_2)]$ with $(r, p) \rhd_{\mathbf{t}} (r', p')$ and $e_2 \rhd_{\mathbf{t}} v'$, hence e_2 is a value v, by Remark 4.2(ii), and therefore $((r, p)e_2)$ is a redex. By Lemma 3.5(ii) there exist φ_0, φ_1 and σ such that $\Sigma; \Gamma \vdash ((r, p)v) : \varphi_0, \sigma$ and $\Sigma; \Gamma \vdash \mathbf{E}_0 : (\sigma \xrightarrow{\varphi_1} \tau)$ with $\varphi_0^- \cap \varphi_1^{\pm} = \emptyset$ and $\varphi = \varphi_0 \cup \varphi_1$. Then $(r, p) \in \mathsf{Dom}(\Sigma)$ with $\Sigma(r, p) = (\theta \xrightarrow{\psi_0} \sigma)_r$ and $\Sigma; \Gamma \vdash v : \psi_1, \theta$ with $\varphi_0 = (\psi_0 + r) \cup \psi_1$. Since $r \in \varphi^+$, we have $r \in \mathsf{dom}(s_0)$ by Definition 3.1(ii), hence $p \in \mathsf{dom}(s_0(r))$ and $s_0(r, p) \rhd_{\mathbf{t}} s_1(r', p')$ by Definition 4.5(ii), that is $s_0(r, p) = w$ with $w \rhd_{\mathbf{t}} w'$. Then $\Sigma; \Gamma \vdash w : (\theta \xrightarrow{\psi_0} \sigma)$ by Definition 3.1(i), and we have

$$(s_0, e_0) \to (s_0, e_0') \quad \text{where} \quad e_0' = \mathbf{E}_0[(wv)]$$

By Lemma 3.5(i) we have $\Sigma; \Gamma \vdash (s_0, e_o') : \psi, \tau$ where $\psi = \psi_0 \cup \psi_1 \cup \varphi_1 \subseteq \varphi$, and it is obvious that $\mathbf{t} \propto \psi$ and $(s_0, e_0') \rhd_{\mathbf{t}}^{\psi} (s_1, e_1')$ (see the remarks (1) and (2) above).

• $(s_1, \mathbf{E}_1[(\lambda x e_2' v')]) \to (s_1, \mathbf{E}_1[\{x \mapsto v'\}e_2'])$. In this case we use the Substitution Lemma 3.4(i). The details are left to the reader.

• $(s_1, \mathbf{E}_1[(\mathsf{new}\ \rho\ \mathsf{in}\ e_2')]) \to (s_1 + \{r' \mapsto \emptyset\}, \mathbf{E}_1[\{\rho \mapsto r'\}e_2'])$ with $r' \notin \mathsf{dom}(s_1)$. We have $e_0 = \mathbf{E}_0[(\mathsf{new}\ \rho\ \mathsf{in}\ e_2)]$ with $e_2 \rhd_{\mathbf{t}} e_2'$, and there exist φ_0, φ_1 and σ such that $\Sigma; \Gamma \vdash (\mathsf{new}\ \rho\ \mathsf{in}\ e_2) : \varphi_0, \sigma$ and $\Sigma; \Gamma \vdash \mathbf{E}_0 : (\sigma \xrightarrow{\varphi_1} \tau)$ with $\varphi_0^- \cap \varphi_1^{\pm} = \emptyset$ and $\varphi = \varphi_0 \cup \varphi_1$ by Lemma 3.5(ii). Let r be a fresh region constant, that does not occur in the statement $\Sigma; \Gamma \vdash (s_0, e_0) : \varphi, \tau$, nor in $\mathsf{dom}(\mathbf{r})$. In particular, $r \notin \mathsf{dom}(s_0)$, and therefore

$$(s_0, e_0) \to (s_0', e_0') \quad \text{where} \quad \begin{cases} s_0' = s_0 + \{r \mapsto \emptyset\} \\ e_0' = \mathbf{E}_0[\{\rho \mapsto r\}e_2] \end{cases}$$

We have $\Sigma; \Gamma \vdash e_2 : \varphi_0', \sigma$ with $\rho \notin \Sigma, \Gamma, \tau$ and $\varphi_0 = \varphi_0' \backslash \rho$. Then by Lemma 3.4(ii) we have $\Sigma; \Gamma \vdash \{\rho \mapsto r\}e_2 : \{\rho \mapsto r\}\varphi_0', \sigma$, hence $\Sigma; \Gamma \vdash (s_0', e_0') : \psi, \tau$ by Lemma 3.5(i), where $\varphi^{\pm} \subseteq \psi^{\pm} \subseteq \varphi^{\pm} \cup \{r\}$. Let $\mathbf{t}' = (\mathbf{r}', \mathbf{p}')$ where $\mathbf{r}' = \mathbf{r} + \{r \mapsto r'\}$ and $\mathbf{p}' = \mathbf{p} + \{r \mapsto \emptyset\}$. If $r'' \in \mathsf{dom}(\mathbf{r})$ is such that $\mathbf{r}(r'') = r'$ then $r'' \notin \psi^{\pm}$ by Definition 4.5(i) since $r' \notin \mathsf{dom}(s_1)$, and therefore $\mathbf{r}' \restriction \psi^{\pm}$ is injective, hence $\mathbf{t}' \propto \psi$. Clearly $\mathbf{r}'(\psi^{\pm}) \subseteq \mathsf{dom}(s_1')$, since $\mathbf{r}(\varphi^{\pm}) \subseteq \mathsf{dom}(s_1)$ by Definition 4.5(i). It is then easy to conclude $(s_0', e_0') \rhd_{\mathbf{t}'}^{\psi} (s_1', e_1')$, using Lemma 4.3(ii).

• $(s_1, \mathbf{E}_1[(\mathsf{dispose}\ r')]) \to (s_1 \backslash r', \mathbf{E}_1[()])$. We have $e_0 = \mathbf{E}_0[(\mathsf{dispose}\ r)]$ with $r \in \mathsf{dom}(\mathbf{r})$ and $\mathbf{r}(r) = r'$. Then

$$(s_0, e_0) \to (s_0', e_0') \quad \text{where} \quad \begin{cases} s_0' = s_0 \backslash r \\ e_0' = \mathbf{E}_0[()] \end{cases}$$

and by Lemma 3.5(ii) there exist ψ and σ such that $\Sigma; \Gamma \vdash \mathbf{E}_0 : (\mathbb{1} \xrightarrow{\psi} \tau)$ with $r \notin \psi^{\pm}$ and $\varphi = \psi \cup (\emptyset, \{r\})$. Then we have $\Sigma; \Gamma \vdash (s_0', e_0') : \psi, \tau$, thanks to Lemma 3.5(i). We obviously have $\mathbf{t} \propto \psi$ (see the remark (1) above). Assume that $r'' \in \psi^{\pm}$ is such that $\mathbf{r}(r'') = r'$. Since $\mathbf{r} \restriction \varphi^{\pm}$ is injective, we should then have $r'' = r$, but this is impossible since $r \notin \psi^{\pm}$. This shows $\mathbf{r}(\psi^{\pm}) \subseteq \mathsf{dom}(s_1')$. Now assume that $r'' \in \psi^{+}$ is such that $(r'', p'') \rhd_{\mathbf{t}} (r', p')$. Then we should have $r'' = r$ since $\mathbf{r} \restriction \varphi^{\pm}$ is injective, but this is impossible since $r \notin \psi^{\pm}$. From this we easily conclude that $s_0' \rhd_{\mathbf{t}}^{\psi} s_1'$, and therefore $(s_0', e_0') \rhd_{\mathbf{t}}^{\psi} (s_1', e_1')$. ❏

We can now use this lemma to show that a typable closed expression does not run into an error. We first define the erroneous configurations.

Definition (Faulty Configuration) 4.7. *A configuration* (s, e) *is* faulty *if either*

(i) e *is a stuck expression, that is* $\mathbf{E}[e']$ *where* e' *is either a variable, or* $(\lambda x e' @ \rho)$, *or* $(()e')$, *or* $(\mathsf{dispose}\ \rho)$, *or*

(ii) e *writes in a deallocated region, that is* $e = \mathbf{E}[(w @ r)]$ *with* $r \notin \mathsf{dom}(s)$, *or*

(iii) e *uses a dangling pointer, that is* $e = \mathbf{E}[((r, p)v)]$ *with* $(r, p) \notin \mathsf{Dom}(s)$.

Then our main result is as follows:

THEOREM (TYPE SAFETY) 4.8. *If (s,e) is a closed, typable configuration, and $(s,e) \xrightarrow{*} (s',e')$, then the configuration (s',e') is not faulty.*

PROOF: first we notice that closedness is preserved by reduction. We have $\Sigma;\Gamma \vdash (s,e) : \varphi,\tau$. Let R be a finite subset of $\mathcal{R}eg\mathcal{C}st$ which contains all the region constants involved in the judgment $\Sigma;\Gamma \vdash (s,e) : \varphi,\tau$ (including the name occurring in values stored in s, etc.). Let us define $\mathbf{t} = (\mathbf{r},\mathbf{p})$ as follows: $\mathbf{r} = \{\,(r,r) \mid r \in R\,\}$ and, for $r \in R$, $\mathbf{p}_r$ maps any address (r,p) occurring in the judgment $\Sigma;\Gamma \vdash (s,e) : \varphi,\tau$ onto itself. Clearly $\mathbf{t} \in \mathcal{T}$, $e \rhd_{\mathbf{t}} e$ (see Remarks 4.2(i)-(iii)), $\mathbf{t} \propto \varphi$ and $s \rhd_{\mathbf{t}}^{\varphi} s$ since $\varphi^{\pm} \subseteq \mathsf{dom}(s)$ by Definition 3.1(ii), and therefore $(s,e) \rhd_{\mathbf{t}}^{\varphi} (s,e)$. Then by Lemma 4.6 there exist (s'',e''), Σ', ψ and $\mathbf{t}' = (\mathbf{r}',\mathbf{p}')$ such that $(s,e) \xrightarrow{*} (s'',e'')$ with $\Sigma';\Gamma \vdash (s'',e'') : \psi,\tau$ and $(s'',e'') \rhd_{\mathbf{t}'}^{\psi} (s',e')$, and in particular $e \rhd_{\mathbf{t}'} e'$. Then by Remarks 3.3 and 4.2(ii), e' is not a stuck expression. If $e' = \mathbf{E}'[(w' \mathbin{@} r')]$ then $e'' = \mathbf{E}[(w \mathbin{@} r)]$ with $\mathbf{r}'(r) = r'$ and $r \in \psi^+$ by Lemma 3.5(ii), and therefore $r \in \mathsf{dom}(s'')$ by Definition 3.1(ii), hence $r' \in \mathsf{dom}(s')$ by Definition 4.5. If $e' = \mathbf{E}'[((r',p')e_1)]$ then $e'' = \mathbf{E}[((r,p)e_0)]$ with $(r,p) \rhd_{\mathbf{t}'} (r',p')$, and $r \in \psi^+$ by Lemma 3.5(ii). We conclude as in the previous case. ⬎

Given a closed, typable expression e with effect φ, which does not contain any memory address, this result applies in particular to an initial configuration (s,e) where $s = \{\, r \mapsto \emptyset \mid r \in \mathsf{reg}(e) \cup \varphi^{\pm} \,\}$. (We conjecture that for such an expression, there exists a typing such that $\varphi^{\pm} \subseteq \mathsf{reg}(e)$.)

5 Conclusion

In this work we have presented a new static analysis for a language with explicit manipulations of memory regions. Our type and effect system is a direct generalization of the one of Tofte and Talpin. We also have introduced a new method for proving type safety in such a language, establishing a "subject reduction up-to-simulation" property that makes apparent the fact that, if we choose "properly" the names created along the computation, then the typing is preserved. We believe that our idea of introducing explicit deallocation effects, which are in some sense dual to the capabilities of [18], and to the set of "currently allocated regions" of [4], can be adapted to richer settings. In particular, in an extended version of this work, we shall show how to deal with region polymorphism and aliasing. We also think our technique could be extended to deal with explicit allocation, as proposed in [1] for instance, by introducing anticipated and actual allocation effects. We preferred not to consider such an extended language here, mainly for the purpose of keeping the exposition simple.

It would be interesting to see whether our static analysis could justify the safety of some of the optimizations, as decribed in [1] for instance, to Tofte and Talpin's compilation from the call-by-value λ-calculus into the region calculus. More generally, it would be interesting to see whether one can take some advantage in using the typed language we have presented as an intermediate language in the compilation process of functional languages.

References

1. Aiken, A., Fähndrich, M., Levien, R.: Better static memory management: improving region-based analysis of higher-order languages. In: PLDl 1995, pp. 174–185 (1995)
2. Banerjee, A., Heintze, N., Riecke, J.: Region analysis and the polymorphic lambda-calculus. In: LICS 1999, pp. 88–97 (1999)
3. Birkedal, L., Tofte, M., Vejlstrup, M.: From region inference to von Neumann machines via region representation inference. In: POPL 1996, pp. 171–183 (1996)
4. Calcagno, C.: Stratified operational semantics for safety and correctness of the region calculus. In: POPL 2001, pp. 155–165 (2001)
5. Calcagno, C., Helsen, S., Thiemann, P.: Syntactic type soundness results for the region calculus. Information and Computation 173(2), 199–221 (2002)
6. Dal Zilio, S., Gordon, A.: Region analysis and a π-calculus with groups. J. of Functional Programming 12(3), 229–292 (2002)
7. Fähndrich, M., DeLine, R.: Adoption and focus: practical linear types for imperative programming. In: PLDI 2002, pp. 13–24 (2002)
8. Fluet, M., Morrisett, G., Ahmed, A.: Linear regions are all you need. In: Sestoft, P. (ed.) ESOP 2006 and ETAPS 2006. LNCS, vol. 3924, Springer, Heidelberg (2006)
9. Gay, D., Aiken, A.: Memory management with explicit regions. In: PLDI 1998, pp. 313–323 (1998)
10. Gay, D., Aiken, A.: Language support for regions. In: PLDI 2001, pp. 70–80 (2001)
11. Helsen, S., Thiemann, P.: Syntactic type soundness for the region calculus. HOOTS 2000, ENTCS 41(3), 1–19 (2001)
12. Henglein, F., Makholm, H., Niss, F.: A direct approach to control-flow sensitive region-based memory management. In: PPDP 2001, pp. 175–186 (2001)
13. Henglein, F., Makholm, H., Niss, F.: Effect Types and Region-Based Memory Management. In: Pierce, B.C. (ed.) Chap. 3 of Advanced Topics in Types and Programming Languages, pp. 87–135. MIT Press, Cambridge (2005)
14. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL 1988, pp. 47–57 (1988)
15. Tofte, M., Talpin, J.-P.: Implementation of the typed call-by-value λ-calculus using a stack of regions. In: POPL 1994, pp. 188–201 (1994)
16. Tofte, M., Talpin, J.-P.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)
17. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N.: A retrospective on region-based memory management. HOSC 17(2), 245–265 (2004)
18. Walker, D., Crary, K., Morrisett, G.: Typed memory management via static capabilities. TOPLAS 22(4), 701–771 (2000)
19. Walker, D., Watkins, K.: On regions and linear types. In: ICFP 2001, pp. 181–192 (2001)
20. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38-94 1994