

Effect Types and Region-based Memory Management

Chapter in *Advanced Topics in Types and Programming Languages*
(Benjamin C. Pierce, editor)

Fritz Henglein, Henning Makholm, and Henning Niss

January 26, 2005

1 *Effect Types and Region-based Memory Management*

By Fritz Henglein, Henning Makholm, and Henning Niss

Type-based program analysis is program analysis based on the concepts, theories and technologies developed for type systems and employed in the definition of programming languages. It is a vast field of research with numerous applications and considerable practical impact. Applications include strictness analysis, data representation analysis, binding-time analysis, soft typing (also called dynamic typing inference), boxing analysis, pointer aliasing, value flow analysis (and all its applications), region-based memory management, communication topology analysis, Year 2000 type analysis, cryptographic protocol verification, locking, race detection, and others; see Palsberg (2001) for an overview of additional applications.

This chapter presents type-based program analysis based on *type and effect systems* (or *effect type systems*) and illustrates their application in *region-based memory management*, which is the chapter's ultimate focus. Classical type systems express properties of values, not the computations leading to those values. *Effect types* describe all important effects of computation, not just their results. Region-based memory management refers to programming where heap data is allocated in individually managed regions that are explicitly allocated and deallocated. As we shall see, state-of-the-art region-based memory management employs effect type systems to ensure *region safety*, which guarantees that no accesses to unallocated or deallocated regions occur at run time.

1.1 Introduction and overview

Region-based memory management has a well-developed theory, has been subject to practice-oriented engineering and is deployed in industrial-quality

language implementations and prototype systems. We provide a consolidated review of its state of the art and use it as an application domain to develop fundamental concepts of effect type systems step by step.

Value flow and simple effect analyses

In §1.2, we introduce BL, a standard typed higher-order functional language. Then we present an TL, an extension of BL, with atomic labels p (tags, names) and corresponding *tagging* and *untagging* operations $t \text{ at } p$ and $t ! p$. Evaluation of $t \text{ at } p$ equips the value v of t with label p resulting in a *tagged value* $\langle v \rangle_p$; correspondingly, evaluation of $t ! p'$ simply removes checks that p in the value $\langle v \rangle_p$ of t equals label p' before returning v . We present a type system which ensures that the check in $t ! p'$ always succeeds. Thus, intuitively, the labels and their operations can be thought of as *annotations* that let us trace where values are created and used at run-time; they express and make explicit *value flow information* of the underlying BL-program. The connection of TL and the value flow information expressed in it to region-based memory management is a reinterpretation of labels as regions and tagging/untagging operations as region access operations. An expression $t \text{ at } p$ is then reinterpreted as “evaluate t , allocate it in the region bound to p and return the corresponding pointer” and $t ! p$ is reinterpreted as “evaluate t to a pointer into the region bound to p and load its contents from there.” This leaves the problem of figuring out when to allocate and deallocate a region. The basic idea is extracting lifetime information about values living in a region ρ from typing derivations: If a (sub)term t that contains uses of a region ρ can be typed such that ρ neither occurs in the typing assumptions nor in the result we take this to mean that ρ does not need to exist before evaluating t nor after. So we can evaluate t by first allocating a new region ρ , then evaluating t , and finally deallocating ρ . To express this we extend TL with a construct $\text{new } \rho.t$ and add straightforward evaluation and typing rules, which yields language STL. §1.2 concludes with the observation that STL is *unsound* because the typing judgements do not capture accesses to regions from the environment part in lexical closures; that is, important properties of the computation (evaluation) itself are not reflected in (the types of) the values produced by those computations.

The unsoundness motivates the use of effect type systems to capture accesses to regions during an evaluation. In §1.3, we introduce *effect types* (types and effects), which represent relevant effects (accesses to regions) of an evaluation together with the type of its result. The basic lifetime interpretation of typing judgements for region allocation and deallocation with explicit effects is then sound since all accesses to regions are represented in the effect of an

expression, also those from the environment part of a lexical closure.

Region-based effect analyses

The development in §1.2 and §1.3 is focussed on the conceptual roles of types, effects, value flow information and lifetime interpretation of typing judgements. In particular, the type systems are monomorphic, and regions cannot be passed as parameters. Turning our attention to realistic region-based memory management, §1.4 extends the region annotations by adding region abstraction and region application. This provides the basis for region polymorphism, which is crucial for practicality. The key result of this section is *conditional correctness*: If a region annotated program does not run into an error (in particular, does not access an unallocated or deallocated region) then it has the same result as the underlying program without region annotations. This result by itself expresses that region annotations may introduce errors during evaluation, but do not otherwise change the semantics of the underlying program. It is noteworthy that conditional correctness holds for all region-annotated terms independent of any type system.

§1.5 presents TT, the Tofte–Talpin region type system, simplified and adapted to our setting. The main result in this section is *type soundness*: no TT-annotated term can go wrong. Combining soundness with conditional correctness we obtain *correctness*: A TT-annotated program produces the same result as the underlying unannotated program. This section highlights the role of the type system: its job is to ensure soundness; conditional correctness is already taken care of.

Region-based systems: inference and systems

There are usually many different well-annotated versions of a given underlying program, all of which are correct. They do not have the same efficiency characteristics: Some retain regions substantially longer during execution than others. In §1.6 we turn to the question of how to automatically infer “good” region annotations. Region inference is technically complex. The section discusses the algorithmic techniques that have been used for TT inference and a number of restricted cases, providing pointers to the relevant literature for detailed descriptions.

The Tofte–Talpin type system enforces a stack discipline on region allocation and deallocation driven by a lexically scoped region-creation expression. §1.7 presents refinements of its standard implementation to accomplish better region performance for lexically scoped regions: region resetting and delayed allocation/early deallocation. Furthermore, it discusses region life-

time subtyping and systems where region allocation and deallocation are decoupled altogether: calculus of capabilities for continuation passing style programs and imperative regions.

Finally, §1.8 surveys implementations with statically checked region-based memory management: ML Kit with Regions, a Standard ML compiler; Cyclone and Vault, which are type-safe C-like languages with explicit region management and other novel extensions; and prototype systems for Java and Prolog. It briefly reviews systems and libraries for region-based programming with no static region safety guarantees, but dynamic or no region fault detection.

1.2 Value flow by typing with labels

The language BL, also called *Finitary PCF* (Jung and Stoughton, 1993; Loader, 2001), is a simply typed lambda-calculus with general recursion (fix), Boolean values and call-by-value semantics. It is the *underlying language*, for which we shall develop region-based memory management based on effect type systems in this section. Its syntax and small-step operational semantics are given in Figure 1-1.

Tagged language

In this subsection we introduce TL, which is BL extended with explicit tagging and untagging operations. Syntax and operational semantics are defined by the definitions of Figure 1-2 extending the definitions for BL as given in Figure 1-1.

The category of label variables ρ designates a denumerable set of label variables ρ_0, ρ_1, \dots . Like ordinary program variables label, variables are atomic and have no internal structure. For convenience we may abbreviate *label expression* to *label*. Labels p can only consist of label variables for now. We shall extend p later. Anticipating their reinterpretation later we shall also call label variables *region variables* and labels *regions* or *places*.

Operationally, evaluation of t at p consists of *tagging* the value of t with label p . We write the result of tagging value v with p as $\langle v \rangle_p$. The untagging operation t at p' evaluates t to a tagged value $\langle v \rangle_p$, checks that its label p matches p' and, if so, returns the underlying value v . If the label does not match the evaluation gets stuck—it *goes wrong*. As we shall see, the typing rules of TL guarantee that evaluation never goes wrong in this way for well-typed terms.

The tagging and untagging operations serve to *name* certain sets of values

Terms	Types
$t ::=$ v x $t t$ $\text{if } t \text{ then } t \text{ else } t$ $\text{fix } x.t$ $v ::=$ $\lambda x.t$ bv $bv ::=$ tt ff	$T ::=$ bool $T \rightarrow T$ <i>terms:</i> <i>value expression</i> <i>variable</i> <i>application</i> <i>conditional</i> <i>recursion</i> <i>value expressions:</i> <i>abstraction</i> <i>truth value</i> <i>truth values:</i> <i>true</i> <i>false</i> <i>types:</i> <i>Boolean type</i> <i>function type</i> <i>Typing rules</i>
<i>Evaluation rules</i> $(\lambda x.t_{12}) v_2 \rightarrow [x \mapsto v_2]t_{12}$ (E-BETA) $\text{fix } x.t \rightarrow [x \mapsto \text{fix } x.t]t$ (E-FIXBETA) $\text{if } tt \text{ then } t_2 \text{ else } t_3 \rightarrow t_2$ (E-IFTRUE) $\text{if } ff \text{ then } t_2 \text{ else } t_3 \rightarrow t_3$ (E-IFFALSE) $\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$ (E-APP1) $\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2}$ (E-APP2) $\frac{t_1 \rightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$ (E-IF)	$\boxed{\Gamma \vdash t : T}$ $\frac{x \notin \Gamma'}{\Gamma, x : T, \Gamma' \vdash x : T}$ (T-VAR) $\Gamma \vdash bv : \text{bool}$ (T-BOOL) $\frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$ (T-IF) $\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \rightarrow T_2}$ (T-ABS) $\frac{\Gamma \vdash t_0 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_1 : T_1}{\Gamma \vdash t_0 t_1 : T_2}$ (T-APP) $\frac{\Gamma, x : T \vdash t : T}{\Gamma \vdash \text{fix } x.t : T}$ (T-FIX) <i>Derived form</i> $\text{let } x = t_1 \text{ in } t_2 \stackrel{\text{def}}{=} (\lambda x.t_2) t_1$

Figure 1-1: Base language BL.

and to mark where such values are constructed and used. Note the following:

- Multiple subterms of a term may have the same label.
- Even though a label may occur only once in a program, it may tag multiple different values at run time; e.g., in $\lambda x_0.(x_0 \text{ at } \rho_0)$ the same label ρ_0 will tag multiple different values if the function is called multiple times with different argument values.
- Labels let us distinguish between values produced in different places even though they are extensionally equal; e.g., in a call $f (tt \text{ at } \rho_0) (tt \text{ at } \rho_1)$

<p><i>New terms</i></p> $t ::= \dots$ $t \text{ at } p$ $t ! p$ $v ::= \dots$ $\langle v \rangle_p$ $p ::= \dots$ ρ <p><i>New evaluation rules</i></p> $\frac{t \longrightarrow t'}{t \text{ at } \rho \longrightarrow t' \text{ at } \rho} \quad (\text{E-TAG})$ $v \text{ at } \rho \longrightarrow \langle v \rangle_\rho \quad (\text{E-TAGBETA})$ $\frac{t \longrightarrow t'}{t ! \rho \longrightarrow t' ! \rho} \quad (\text{E-UNTAG})$ $\langle v \rangle_\rho ! \rho \longrightarrow v \quad (\text{E-UNTAGBETA})$	<p><i>terms:</i></p> <p style="padding-left: 20px;"><i>tagging</i></p> <p style="padding-left: 20px;"><i>untagging</i></p> <p><i>value expressions:</i></p> <p style="padding-left: 20px;"><i>tagged value</i></p> <p><i>label expressions:</i></p> <p style="padding-left: 20px;"><i>label variable</i></p> <div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 5px auto;">$t \longrightarrow t'$</div>	<p><i>New types</i></p> $T ::= \dots$ $T \text{ at } p$ <p><i>New typing rules</i></p> <div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 5px auto;">$\Gamma \vdash t : T$</div> $\frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ at } p : T \text{ at } p} \quad (\text{T-TAG})$ $\frac{\Gamma \vdash t : T \text{ at } p}{\Gamma \vdash t ! p : T} \quad (\text{T-UNTAG})$ $\frac{\Gamma \vdash v : T}{\Gamma \vdash \langle v \rangle_p : T \text{ at } p} \quad (\text{T-TAGVALUE})$ <p><i>types:</i></p> <p style="padding-left: 20px;"><i>tagged value type</i></p>
---	--	--

Figure 1-2: Tagged Language, TL (Extension of BL)

we can keep track of the uses of both arguments separately even though they are the same values.

- We distinguish between $v \text{ at } p$ and $\langle v \rangle_p$. The former denotes an unevaluated expression, where v has not been tagged with p yet, and the latter denotes the result of performing the tagging. This distinction will be important when interpreting labels as regions later on: evaluation of the former has the effect of accessing the region p whereas the latter does not.

A term is *closed* if it has no free occurrences of variables. (Closed terms may have free occurrences of label variables.) A closed value expression is a *value*.

We write $t \xrightarrow{T} t'$ if $t \longrightarrow t'$ can be derived from the TL evaluation rules; that is, the evaluation rules of both Figures 1-1 and 1-2. A term t is *final* (or a *final state*) if there is no term t' such that $t \xrightarrow{T} t'$. Each value expression is final. We call all final states that are not value expressions *stuck* (or *stuck states*).

We write $t \downarrow t'$ if $t \xrightarrow{T}^* t'$ and t' is final. We write $t \downarrow$ if there exists t' such that $t \downarrow t'$. If t has no final state and thus does not terminate we write $t \uparrow$.

For simplicity we shall think of BL as a subset of TL. This is justified since all BL-terms are also TL-terms and both evaluation and typing relations for TL are conservative over BL. For emphasis we may write $t \xrightarrow{\text{BL}} t'$ if $t \xrightarrow{\text{T}} t'$ and t, t' are BL-terms.

Labels as value flow information

The label erasure (or simply erasure) of a TL-term is the BL-term we obtain by erasing all occurrences of $\text{at } p$, $! p$ and $\langle \cdot \rangle_p$ in it. More precisely, we define erasure and its inverse, completion, as follows:

1.2.1 DEFINITION [ERASURE, COMPLETION]: Let $t \in \text{TL}$. Then the *erasure* $\|t\|$ of term t is defined as follows:

$$\begin{aligned} \|x\| &= x \\ \|t_1 t_2\| &= \|t_1\| \|t_2\| \\ \|\text{if } t_1 \text{ then } t_2 \text{ else } t_3\| &= \text{if } \|t_1\| \text{ then } \|t_2\| \text{ else } \|t_3\| \\ \|\text{fix } x.t\| &= \text{fix } x.\|t\| \\ \|\lambda x.t\| &= \lambda x.\|t\| \\ \|tt\| &= tt \\ \|ff\| &= ff \\ \|t \text{ at } p\| &= \|t\| \\ \|t ! p\| &= \|t\| \\ \|\langle v \rangle_p\| &= \|v\| \end{aligned}$$

Conversely, we call a TL-term t' a *completion* of BL-term t if $\|t'\| = t$. \square

Note that erasures are BL-terms. Note also that erasures are closed under substitution:

1.2.2 PROPOSITION: $\|[x \mapsto t_2]t_1\| = [x \mapsto \|t_2\|]\|t_1\|$ \square

A *constructor/deconstructor completion* (or *con/decon completion*) is a completion where each value expression is tagged and untagging takes place in each destructive context; labels must not occur anywhere else. Formally, *con/decon completions* are generated from t in Figure 1-3.

In a con/decon completion each value gets tagged when it is created and every such tag is checked and removed immediately before the underlying untagged value is destructed; that is, needed as the function in a function application or as the Boolean test in a conditional. In this fashion the label p in $t ! p$ tells us which value expressions could have constructed the value of t .

<i>Con/decon completion templates</i>	
$t ::=$ v x $(t ! p) t$ $\text{if } (t ! p) \text{ then } t \text{ else } t$ $\text{fix } x.t$	$:$ $v ::=$ $(\lambda x.t) \text{ at } p$ $bv \text{ at } p$ $bv ::=$ tt ff
	$:$ <i>abstraction</i> <i>truth value</i> <i>truth values:</i> <i>true</i> <i>false</i>

Figure 1-3: Con/decon completions

For this to be true, however, evaluations of TL-terms must not get stuck due to label mismatch in a redex $\langle v \rangle_{\rho} ! \rho'$. Intuitively, the reason for this is as follows. It would be clearly wrong to conclude that evaluation of $t ! \rho_1$ uses a value constructed by a value expression labeled ρ_1 in the original source program if t evaluates to $\langle tt \rangle_{\rho_0}$. Note, however, that $\langle tt \rangle_{\rho_0} ! \rho_1$ is stuck, which means $t ! \rho_1$ gets stuck. Conversely, if an evaluation does not get stuck, all its computation steps of the form $\langle tt \rangle_{\rho} ! \rho'$ succeed, which is only possible if $\rho = \rho'$. In that case a subterm $t ! \rho$ expresses that the value of t is constructed from one of the value expressions labeled ρ .

As we shall see, the type system of TL guarantees that no stuck states can occur during evaluation of (well-typed) TL-terms. So the label information in TL-terms can be soundly interpreted as value flow information.

1.2.3 EXAMPLE: Consider the BL-program t_0 :

```
let fst =  $\lambda u.\lambda v.u$  in
  (let x =  $\lambda p.p \ tt \ ff$  in  $\lambda y.\lambda q.q (x \ fst) \ y$ )
  tt
```

Value flow analysis should tell us that x may be applied to fst (which is rather easy to see), fst may be applied to tt (which is not immediately obvious from the source code), and the λ -abstraction $\lambda y.\lambda p.p (x \ \text{fst}) \ y$ may be applied to tt , but $\lambda p.p (x \ \text{fst}) \ y$ is not applied anywhere.

The following con/decon completion t_1 of t_0 captures this:

```
let fst =  $\lambda_{l_k} u.\lambda_{l_b} v.u$  in
  (let x =  $\lambda_{l_x} p.((p^{l_k} \ tt_{l_t})^{l_b} \ ff_{l_f})$  in
     $\lambda_{l_f} y.\lambda_{l_c} q.((q^{l_q} (x^{l_x} \ \text{fst}))^{l_d} \ y)$ 
     $tt_{l_t}$ 
```

To make the completion more readable, we have written $\lambda_p x.t$ for $(\lambda x.t) \text{ at } p$, bv_p for $bv \text{ at } p$, and $(t^p t')$ for $(t ! p) t'$. \square

- 1.2.4 EXERCISE [$\star \rightarrow$]: Show that t_1 is a TL-term by giving a TL-typing derivation for it. \square
- 1.2.5 EXERCISE [$\star \rightarrow$]: Give a reduction sequence $t_1 \xrightarrow{T} \dots \xrightarrow{T} t_k$ such that t_k is final. Which (E-UNTAGBETA) reduction steps occur in it? Which labels occur in those steps? \square
- 1.2.6 EXERCISE [$\star \rightarrow$]: Note that t_0 is the erasure of t_1 : $\|t_1\| = t_0$. Give a reduction sequence $\|t_1\| \xrightarrow{BL} \dots \xrightarrow{BL} t'_m$ such that t'_m is final. How are t_k from Exercise 1.2.5 and t'_m related to each other? How long is the reduction sequence for t_0 to t_k in comparison to the reduction sequence for t_1 ? (Generalize to arbitrary TL-terms and their erasures.) \square
- 1.2.7 EXERCISE [$\star\star$]: Let S be a substitution mapping the labels occurring in t_1 to (not necessarily different) labels. Consider the term $S(t_1)$, which is t_1 with its labels substituted according to S . Is $S(t_1)$ TL-typable? If so, does closure under all substitutions hold for all closed TL-terms? If not, for which subset of the closed TL-terms does it hold? \square
- 1.2.7 SOLUTION: All TL-typing rules are closed under arbitrary substitutions. Consequently all substitutions of TL-typable closed terms yield typable terms. In particular, $S(t_1)$ is TL-typable.

Correctness

A TL-term can be thought of as an *instrumented* version of the underlying BL-term. Intuitively, this is because an evaluation of a TL-term performs the same proper computation steps as its erasure (the underlying BL-term), with interspersed auxiliary label reduction steps (E-TAGBETA) and (E-UNTAGBETA).

Correctness means that evaluation of TL-terms gives the “same” results as evaluation of their underlying BL-terms. It is factored into two orthogonal parts:

1. Conditional correctness, which states that TL-terms produce the same results as their underlying BL-terms unless they get stuck. Conditional correctness is a property of the evaluation rules for TL and BL alone; it is *independent* of their typing rules.
2. Soundness, which states that TL-terms do not get stuck.

It is instructive to see how this method works in a technically very simple setting such as TL. For this reason we shall introduce it below. The same results for more expressive languages with effect typing, region scoping and polymorphism will be proved later on in §1.4 and §1.5.

1.2.8 DEFINITION: Define relations $\cdot \xrightarrow{T_1} \cdot$ and $\cdot \xrightarrow{T_2} \cdot$ on TL-terms as follows:

1. $t_1 \xrightarrow{T_2} t_2$ if $t_1 \longrightarrow t_2$ is derived by application of Axiom (E-TAGBETA) or (E-UNTAGBETA) from Figure 1-2.
2. $t_1 \xrightarrow{T_1} t_2$ if $t_1 \longrightarrow t_2$ is derived from all evaluation rules of Figures 1-1 and 1-2, though without application of Axioms (E-TAGBETA) or (E-UNTAGBETA).

□

Each $\xrightarrow{T_1}$ reduction step corresponds to a reduction step in the underlying BL-term whereas $\xrightarrow{T_2}$ reductions do not change the underlying BL-term at all. This is captured in the following lemma.

1.2.9 LEMMA [SIMULATION]: Let t, t_1, t_2 range over TL-terms.

1. If v is a value expression then so is $\|v\|$.
2. $\xrightarrow{T_2}$ is strongly normalizing.
3. If $t_1 \xrightarrow{T_1} t_2$ then $\|t_1\| \xrightarrow{BL} \|t_2\|$.
4. If $t_1 \xrightarrow{T_2} t_2$ then $\|t_1\| = \|t_2\|$.

□

1.2.10 EXERCISE [★★ \Rightarrow]: Prove Lemma 1.2.9. □

Using Lemma 1.2.9 we can prove the following theorem. It states that evaluation of a TL-term performs basically the same computation steps as the underlying BL-term until it gets stuck or arrives at a value expression.

1.2.11 THEOREM [CONDITIONAL CORRECTNESS]: For TL-terms t, t' we have:

1. If $t \xrightarrow{T}^* t'$ then $\|t\| \xrightarrow{BL}^* \|t'\|$.
2. If $t \uparrow$ then $\|t\| \uparrow$.
3. If $\|t\|$ gets stuck then t gets stuck, too.

□

1.2.12 EXERCISE [★]: Prove Theorem 1.2.11. □

The following lemma expresses that the type of a term is preserved under evaluation.

1.2.13 LEMMA [SUBJECT REDUCTION (PRESERVATION)]: Let t, t' be TL-terms. If $\Gamma \vdash t : T$ and $t \xrightarrow{T} t'$ then $\Gamma \vdash t' : T$. □

1.2.14 EXERCISE [★★→]: Prove Lemma 1.2.13 in standard fashion: by induction on $t \xrightarrow{T} t'$ and formulating the requisite substitution lemma. \square

1.2.15 LEMMA [PROGRESS]: If $\vdash t : T$ then either $t = v$ for some value (closed value expression) v or there exists t' such that $t \xrightarrow{T} t'$. \square

Proof: (Sketch) The lemma follows from the following statement: For all derivable $\Gamma \vdash t : T$, if $\Gamma = \emptyset$ then

1. there exists t' such that $t \xrightarrow{T} t'$, or
2. (a) if T is of the form $T' \rightarrow T''$ then $t = \lambda x.t''$ for some x, t'' , and
(b) if $T = \text{bool}$ then $t \in \{tt, ff\}$.

This statement can be proved by rule induction on $\Gamma \vdash t : T$. \square

The Progress Lemma says that a well-typed closed term is not stuck. Together with the Subject Reduction Lemma it says that, since all its reducts are well-typed, too, it never gets stuck.

1.2.16 THEOREM [SOUNDNESS]: If $\vdash t : T$ then evaluation of t does not get stuck. \square

Putting the Conditional Correctness Theorem and the Soundness Theorem together we obtain as a corollary the correctness of TL relative to BL:

1.2.17 COROLLARY [CORRECTNESS]: Let t be a closed TL-term and v a TL-value.

1. $t \uparrow$ if and only if $\|t\| \uparrow$.
2. $\|t\| \xrightarrow{\text{BL}}^* \|v\|$ if and only if there exists a TL-value v' such that $\|v'\| = \|v\|$ and $t \xrightarrow{T}^* v'$. \square

1.2.18 EXERCISE [★★]: Prove Corollary 1.2.17. \square

1.2.18 SOLUTION:

1. That $t \uparrow$ implies $\|t\| \uparrow$ follows directly from the Conditional Correctness Theorem, part 2. Assume now $t \downarrow$, that is t terminates. By the Soundness Theorem t cannot terminate with a stuck state, so $t \xrightarrow{T}^* v$ for some value v . By Conditional Correctness, part 1, this implies that $\|t\| \xrightarrow{*} \|v\|$. By Lemma 1.2.9, part 1, $\|v\|$ is also a value. Since all values are final (easy), this shows that $\|t\| \downarrow$.

2. The implication from right to left follows directly from Conditional Correctness, part 1. As for the converse direction, assume $\|\tau\| \longrightarrow^* \|\nu\|$. By Soundness evaluation of τ does not get stuck and by part 1 of the corollary there exists a TL-value ν' such that $\tau \xrightarrow{T}^* \nu'$. By Conditional Correctness, part 1, we have $\|\tau\| \longrightarrow^* \|\nu'\|$. Since \longrightarrow is deterministic and we also have $\|\tau\| \longrightarrow^* \|\nu\|$ by assumption, we can conclude that $\|\nu'\| = \|\nu\|$, and we are done.

Inference of value flow information

Given a BL-term τ we are interested in finding a con/decon completion to obtain value flow information about τ . Note, however, that a BL-term τ may have many different con/decon completions, and while each completion provides sound value flow information, some completions provide better information than others. For example, the trivial completion where each label operation in a term has the *same* label ρ_{h} contains basically no useful value flow information: it says that any value created anywhere may be used anywhere. Correct, but trivial. Intuitively, we are interested in a con/decon completion with a maximal number of distinct labels as this gives the most fine-grained value flow information.

1.2.19 EXERCISE: Consider $\tau_0 =$

$$\begin{aligned} &\text{let fst} = \lambda u.\lambda v.u \text{ in} \\ &\quad (\text{let } x = \lambda p.p \text{ tt ff in } \lambda y.\lambda q.q (x \text{ fst}) y) \\ &\quad \text{tt} \end{aligned}$$

again and its con/decon completion $\tau_1 =$

$$\begin{aligned} &\text{let fst} = \lambda_{l_x} u.\lambda_{l_y} v.u \text{ in} \\ &\quad (\text{let } x = \lambda_{l_x} p.((p^{l_x} \text{ tt}_{l_t})^{l_b} \text{ ff}_{l_f}) \text{ in} \\ &\quad \quad \lambda_{l_f} y.\lambda_{l_c} q.((q^{l_q} (x^{l_x} \text{ fst}))^{l_d} y)) \\ &\quad \text{tt}_{l_t} \end{aligned}$$

Does there exist another con/decon completion of τ_0 with more distinct labels or is τ_1 maximal in this sense? \square

1.2.19 SOLUTION: (Sketch) There is a better completion. The two occurrences of tt can be given distinct labels.

Indeed, it can be shown that each BL-term has a con/decon completion such that any other of its con/decon completions can be obtained by applying a label substitution to it. We call it a *principal completion* of the given BL-term. In particular principal completions have the maximal possible number

of distinct labels. Furthermore, principal completions are unique up to re-naming of labels.

We shall not go into any technical details on how to infer principal completions, but present the basic ideas.

A con/decon completion template for an BL-term τ is a con/decon completion of τ where each label variable occurs exactly once. Clearly, each con/decon completion that satisfies the TL-typing rules is a substitution instance (mapping labels to labels) of this template. Furthermore, it can be shown that a substitution gives rise to a well-typed con/decon completion if and only if it satisfies a set of equational constraints between the template labels. That set can be computed in linear time from the con/decon completion and a most general solution of the constraints, likewise, can be computed in linear time. The most general solution, in turn, gives rise to a principal completion when applied to the con/decon completion template. What we have described is the standard method for simple type inference by equational constraint extraction and solution; see e.g. Wand (1987); Henglein (1989) for simple type inference and Mossin (1997, Section 2) for its application to value flow analysis.

The pleasant properties of processing sets of equational constraints, in particular existence of most general/principal solutions and very efficient incremental algorithms for computing them (unification), appears to have led to type systems whose design has been driven to a considerable degree by a desire to deploy efficient unification technology for automatic inference, not only by semantic or logical analysis for capturing relevant semantic information.

Labels as regions

We can think of region variables being bound to memory regions and (re)interpret tagging and untagging operations as follows. The value $\langle v \rangle_p$ denotes a(ny) pointer into region p where v is stored. The tagging operation τ at p is implemented by storing the value of τ in region p . Its result is the pointer to where the value is stored. The untagging operation $\tau ! p$ is implemented as evaluating τ to a pointer, checking that it points into region p and, if it does, retrieving the pointer's value. The TL type system guarantees that all checks succeed and so can be elided at run time.

Now consider the (derivable) judgement $\Gamma \vdash \tau : T$ for a subterm τ in a program. Assume that ρ occurs in τ in tagging and/or untagging operations. If ρ does not occur in Γ then, intuitively, the environment in which τ evaluates contains no values in ρ ; it is empty. If, furthermore, ρ does not occur in T either, then no values stored in ρ are needed by the context of τ ; all the values

<p><i>New terms</i></p> <p>$t ::= \dots$ $\text{new } \rho.t$</p> <p>$p ::=$ \bullet</p> <p><i>New evaluation rules</i></p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"> $t \xrightarrow{ST} t'$ </div>	<p><i>terms:</i> <i>region-scoped term</i></p> <p><i>label expressions:</i> <i>deleted/inaccessible region</i></p>	<div style="border-left: 1px solid black; border-right: 1px solid black; padding: 0 10px;"> $\frac{t \longrightarrow t'}{\text{new } \rho.t \longrightarrow \text{new } \rho.t} \quad (\text{E-NEW})$ $\text{new } \rho.v \longrightarrow [\rho \mapsto \bullet]v \quad (\text{E-NEWBETA})$ </div> <p><i>New typing rules</i></p> <div style="border: 1px solid black; padding: 2px; display: inline-block; margin-right: 10px;"> $\Gamma \vdash t : T$ </div> $\frac{\Gamma \vdash t : T \quad \rho \notin \text{frv}(\Gamma, T)}{\Gamma \vdash \text{new } \rho.t : T} \quad (\text{T-NEWUN SOUND})$
---	--	---

Figure 1-4: Scoped Tagged Language (unsound), STL (Basis: TL)

stored in ρ can be deleted.

This leads us to the introduction of terms with (*lexically*) *scoped regions*: $\text{new } \rho.t$. Here $\text{new } \rho.t$ binds region variable ρ in t . The semantics of $\text{new } \rho.t$ is as follows: allocate a new region, bind it to ρ , evaluate t and, finally, deallocate the region bound to ρ . This results in a stack-oriented memory management discipline for regions: the most recently allocated region is deallocated first.

Figure 1-4 shows $\text{new } \rho.t$ and corresponding evaluation and typing rules, which extend TL to STL. In Rule (T-NEWUN SOUND) function $\text{frv}(\Gamma, T)$ denotes the set of region variables that occur freely in Γ and T .

Rule (E-NEW) expresses that a region-scoped term $\text{new } \rho.t$ is evaluated by reducing t to a value v . During this reduction accesses to region ρ are possible. After evaluation is complete $\text{new } \rho.v$ is reduced to $[\rho \mapsto \bullet]v$ by Rule (E-NEWBETA), where \bullet is substituted for all occurrences of ρ in v . In particular, all occurrences in v of the form $\langle v' \rangle_\rho$ are replaced by $\langle v' \rangle_\bullet$. Since Rule (E-UNTAGBETA) from Fig 1-2 requires a proper region variable, any access to such a value gets stuck. Note in particular that the term $\langle v' \rangle_\bullet ! \bullet$ is stuck. In this fashion the substitution of \bullet for ρ makes all values stored in ρ inaccessible in ensuing computation steps, which models deleting the whole region of values stored in ρ .

The bad news is that STL is unsound: Stuck states can occur. The reason for this is that a term t with derivable judgement $\Gamma \vdash t : T$ may still access region ρ during evaluation even if ρ occurs neither in Γ nor T .

1.2.20 EXAMPLE: Consider the following STL-term $t_f =$:

$$\text{new } \rho_0.\text{let } x = tt \text{ at } \rho_0 \text{ in } \lambda y.\text{if } x ! \rho_0 \text{ then } y \text{ else } ff \text{ at } \rho_1$$

It reduces as follows:

$$\begin{array}{l}
\text{new } \rho_0. \text{let } x = \text{tt at } \rho_0 \text{ in } \lambda y. \text{if } x ! \rho_0 \text{ then } y \text{ else } \text{ff at } \rho_1 \longrightarrow \\
\text{new } \rho_0. \text{let } x = \langle \text{tt} \rangle_{\rho_0} \text{ in } \lambda y. \text{if } x ! \rho_0 \text{ then } y \text{ else } \text{ff at } \rho_1 \longrightarrow \\
\text{new } \rho_0. \lambda y. \text{if } \langle \text{tt} \rangle_{\rho_0} ! \rho_0 \text{ then } y \text{ else } \text{ff at } \rho_1 \longrightarrow \\
\lambda y. \text{if } \langle \text{tt} \rangle_{\bullet} ! \bullet \text{ then } y \text{ else } \text{ff at } \rho_1
\end{array}$$

Note that ρ_0 occurs freely in $\lambda y. \text{if } \langle \text{tt} \rangle_{\rho_0} ! \rho_0 \text{ then } y \text{ else } \text{ff at } \rho_1$ before performing the last reduction step. Its type $\text{bool at } \rho_1 \rightarrow \text{bool at } \rho_1$, however, does not mention ρ_0 . Note that

$$\lambda y. \text{if } \langle \text{tt} \rangle_{\bullet} ! \bullet \text{ then } y \text{ else } \text{ff at } \rho_1$$

is a value; it is *not* stuck. It is easy to see, however, how it can give rise to a stuck state. The program $t_f (\text{tt at } \rho_1)$ is a well-typed STL-program of type $\text{bool at } \rho_1$, yet evaluation gets stuck:

$$\begin{array}{l}
t_f (\text{tt at } \rho_1) \xrightarrow{*} \\
(\lambda y. \text{if } \langle \text{tt} \rangle_{\bullet} ! \bullet \text{ then } y \text{ else } \text{ff at } \rho_1) (\text{tt at } \rho_1) \longrightarrow \\
\text{if } \langle \text{tt} \rangle_{\bullet} ! \bullet \text{ then } \text{ff at } \rho_1 \text{ else } \text{ff at } \rho_1
\end{array}$$

To continue evaluation would require reduction of $\langle \text{tt} \rangle_{\bullet} ! \bullet$ to a Boolean value; $\langle \text{tt} \rangle_{\bullet} ! \bullet$, however, is stuck. \square

In §1.3 we introduce explicit effects into types to capture the accesses to regions needed for evaluation. This is the path taken by (Tofte and Talpin, 1997) in their ground-breaking work on region-based memory management.

Notes on value flow analysis

Even though Reynolds (1969) was first to look at the problem of computing flow for structured data and called it *data set computation*, we follow Schwartz (1975) in using the term *value flow* to emphasize its general applicability to primitive, structured and higher-order data. Schwartz (1975) developed value flow analysis for structured values in the context of SETL and was the first to suggest exploiting lifetime analysis based on value flow analysis for region-based memory management. *Closure analysis*, the term introduced by Sestoft (Sestoft, 1988, 1989), and *control flow analysis*, the term used by Shivers (Shivers, 1988, 1991), focus on the flow of function values (function closures). Note that data and control flow are interdependent for higher-order languages; see Mossin (Mossin, 1997, Section 1.4) for a discussion of this. Shivers coined the term OCFA, which in other literature is also used for the monovariant value flow analyses above (which is different from Shivers'

OCFA, however; see Mossin (1997) for a discussion of this). Another form of monovariant value flow analysis is set-based analysis (Heintze, 1994).

Palsberg and O’Keefe (1995) showed that safety analysis, a constraint-based analysis, characterizes typability in Amadio and Cardelli (1991)’s type system with recursive subtyping, providing a type theoretic characterization of monovariant value flow analysis. Constraint-based value flow analysis for object-oriented languages was pioneered by Palsberg and Schwartzbach (1990, 1994). See Nielson et al. (1999) for a presentation of monovariant value flow analysis based on flow logic and abstract interpretation.

Classical data flow analysis corresponds to value flow analysis for primitive data (only); it has been used in compilers already in the early 60s. See Aho et al. (1986) for its history.

Monovariant value flow analysis is *directional*: values flow from constructor points (value expressions) to uses, but not the other way round. The value flow information for BL as expressed in TL-completions corresponds to a very simple (and inexpressive) value flow analysis: equational flow analysis, where value flows are treated symmetrically (Heintze, 1995). Intuitively, this means all flows are bidirectional: values do not only “flow” from their creation points to their uses as in monovariant value flow analysis (OCFA), but, somewhat weirdly, also the other way round. The type-based presentation of equational value flow analysis in this section owes greatly to Mossin (1997, Section 2) where it is called Simple Flow Analysis.

Polymorphic value flow analysis was developed by Mossin (1997), extending earlier work by Dussart et al. (1995); Henglein and Mossin (1994) on combining subtyping, parametric polymorphism and polymorphic recursion for binding-time analysis. Polymorphic value flow analysis is modular and can be computed asymptotically in the same time as monomorphic value flow analysis. Transitive closure is the algorithmic bottleneck in both. Efficient algorithms are given in Fähndrich et al. (2000); Rehof and Fähndrich (2001) and Gustavsson and Svenningsson (2001).

Polymorphic equational value flow analysis underlies Tofte-Talpin style region-based memory management; see §1.5. Region-based memory management based on “directional” value flow analysis appears possible, but is presently unexplored. The recognition that region inference performs a form of value flow and dependency analysis is folklore; it has been exploited by Helsen and Thiemann (2003) for polymorphic specialization. Developing region-based memory management systematically from type-based value flow analysis appears to new, however. The syntactic modeling of region deallocation by • is due to Helsen and Thiemann (2000) and Calcagno (2001).

See §1.4 and the following sections for more references on region-based memory management.

1.3 Effects

We have seen in the previous subsection that the soundness of a typing rule may depend not only on the results of evaluations, but on certain aspects of the evaluation itself, in other words on *how* a value is computed, not just *which* value is computed. To capture properties of evaluation we introduce *effects*.

Effect type judgements

The basic effect type judgement is

$$\Gamma \vdash t :^{\varphi} T$$

where φ is an *effect expression* (henceforth simply called *effect*) and ${}^{\varphi}T$ is an *effect type* or *type and effect*. The judgement should be read informally as “Under the assumptions Γ , the evaluation of t may have the observable effect φ , and it eventually yields a value of type T , if any.” For program analysis purposes *observable* may also be understood as *interesting*. When an evaluation has no observable effect, we say it has the *empty effect*, written \emptyset , and ${}^{\emptyset}T$ is abbreviated to T .

In a call-by-name language Γ is a sequence of *effect type assumptions* of the form $x : {}^{\varphi}T$ since x may be bound to unevaluated thunks, whereas in a call-by-value language we have *type assumptions* of the form $x : T$ since variables are bound to *values*, whose evaluation is guaranteed to always have the empty effect. Analogously, in a call-by-name language we have general functional types of the form ${}^{\varphi_1}T_1 \rightarrow {}^{\varphi_2}T_2$; in a call-by-value language, however, we can restrict ourselves to functional types of the form $T_1 \rightarrow {}^{\varphi}T_2$.¹

Effect typed language ETL

We shall now present an effect typing system for language ETL. ETL has the same source terms and evaluation rules as STL. The only difference to STL

1. The syntax ${}^{\varphi}T$ has been chosen here for several reasons:

- It expresses that yielding a value of type T is the last “effect” of evaluation; that is it occurs after φ .
- Functional types in a call-by-value language end up being written $T_1 \rightarrow {}^{\varphi}T_2$, which is consistent with the notation used in the literature where the *delayed effect* φ is written above the function type arrow.
- It is consistent with the syntax $M^{\varphi}T$ used in monadic interpretations of type and effect systems in the literature.

Terms		Effect typing rules	$\Gamma \vdash t :^\varphi T$
$t ::=$			
v	<i>terms:</i> <i>value expression</i>	$\frac{x \notin \Gamma'}{\Gamma, x : T, \Gamma' \vdash x :^\varphi T}$	(TE-VAR)
x	<i>variable</i>		
$t t$	<i>application</i>	$\frac{\Gamma \vdash bv :^\varphi \text{bool}}{\Gamma \vdash t_1 :^\varphi \text{bool}}$	(TE-BOOL)
$\text{if } t \text{ then } t \text{ else } t$	<i>conditional</i>	$\frac{\Gamma \vdash t_1 :^\varphi \text{bool} \quad \Gamma \vdash t_2 :^\varphi T \quad \Gamma \vdash t_3 :^\varphi T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 :^\varphi T}$	(TE-IF)
$t \text{ at } p$	<i>tagging</i>		
$t ! p$	<i>untagging</i>		
$\text{new } \rho.t$	<i>label-scoped term</i>		
$\text{fix } x.t$	<i>recursion</i>	$\frac{\Gamma, x : T_1 \vdash t :^{\varphi^2} T_2}{\Gamma \vdash \lambda x.t :^{\varphi^1} T_1 \rightarrow^{\varphi^2} T_2}$	(TE-ABS)
$v ::=$	<i>value expressions:</i>		
$\lambda x.t$	<i>abstraction</i>	$\frac{\Gamma \vdash t_0 :^\varphi T_1 \rightarrow^{\varphi} T_2}{\Gamma \vdash t_1 :^\varphi T_1}$	(TE-APP)
bv	<i>truth value</i>		
$\langle v \rangle_p$	<i>tagged value</i>	$\frac{\Gamma \vdash t :^\varphi T \quad p \in \varphi}{\Gamma \vdash t \text{ at } p :^\varphi T \text{ at } p}$	(TE-AT)
$bv ::=$	<i>truth values:</i>		
tt	<i>true</i>	$\frac{\Gamma \vdash t :^\varphi T \quad p \in \varphi}{\Gamma \vdash t \text{ at } p :^\varphi T \text{ at } p}$	(TE-FROM)
ff	<i>false</i>	$\frac{\Gamma \vdash v :^\varphi T}{\Gamma \vdash \langle v \rangle_p :^\varphi T \text{ at } p}$	(TE-CELL)
$p ::=$	<i>label/region expressions:</i>	$\frac{\Gamma \vdash t :^\varphi T \quad p \in \varphi}{\Gamma \vdash t ! p :^\varphi T}$	(TE-NEW)
ρ	<i>label/region variable)</i>		
\bullet	<i>deleted/inaccessible label/region)</i>	$\frac{\Gamma \vdash t :^\varphi T \quad \rho \notin \text{frv}(\Gamma, T)}{\Gamma \vdash \text{new } \rho.t :^{\varphi - \{\rho\}} T}$	(TE-FIX)
<i>Effect expressions</i>			
$\varphi ::= \{\rho, \dots, \rho\}$	<i>effect expressions:</i>		
<i>Types</i>			
$T ::=$	<i>types:</i>	$\frac{\Gamma, x : T \vdash t :^\varphi T}{\Gamma \vdash \text{fix } x.t :^\varphi T}$	
bool	<i>Boolean type</i>		
$T \rightarrow^\varphi T$	<i>function type</i>		
$T \text{ at } p$	<i>tagged value type</i>		

Figure 1-5: Scoped effect typed language ETL(sound).

is its effect type system. Syntax and effect typing rules for ETL are given in Figure 1-5.

An effect is a finite set of region variables. Note that it must not contain

- A judgement $\Gamma \vdash t :^\varphi T$ is intended to express that, assuming the free variables of t are bound to values of types according to Γ , the regions that t accesses during evaluation are included in φ and the result of the evaluation has type T if it terminates.

The typing rules of Figure 1-5 are basically those corresponding to the monomorphic subset of the Tofte-Talpin System, which we shall encounter in §1.5.² They are inspired by a desire to employ equational constraint solving for effect expressions as much as possible.

Since we are only interested in whether a particular region may be accessed during evaluation of a term or not, our effect system does not record the *order* in which effects take place. We simply record the *set* of region variables that may be accessed during evaluation of t . In this sense our effect type system is *control-flow insensitive*. Effect type systems that capture evaluation order in their effects are discussed briefly later.

Soundness

Effects make the region variables accessed during evaluation sufficiently “visible” to ensure that the typing rule for $\text{new } \rho.t$ is sound.

1.3.1 EXAMPLE: Consider the term

$$t_f = \text{new } \rho_0. \text{let } x = tt \text{ at } \rho_0 \text{ in } \lambda y. \text{if } x! \rho_0 \text{ then } y \text{ else } ff \text{ at } \rho_1$$

from Example 1.2.20 again. Whereas it is typable in STL even though it gets stuck when applied to an argument, it is not typable in ETL. To see this, consider the let-expression t_l

$$\text{let } x = tt \text{ at } \rho_0 \text{ in } \lambda y. \text{if } x! \rho_0 \text{ then } y \text{ else } ff \text{ at } \rho_1$$

inside t_f . Its ETL effect type T_l is $\{\rho_0\}(\text{bool at } \rho_1 \rightarrow \{\rho_0\}\text{bool at } \rho_1)$. Note that ρ_0 occurs in the effect, but neither in the function type’s domain nor its range type. This reflects the fact that an application of t_l may access region ρ_0 . Since $\rho_0 \in \text{frv}(\text{bool at } \rho_1 \rightarrow \{\rho_0\}\text{bool at } \rho_1)$ Rule (TE-NEW) is *not* applicable, and so there is no way of inferring a type for t_f , which indeed would be unsound. \square

1.3.2 EXERCISE [★★ \rightarrow]: Give a derivation of $\vdash t_l : T_l$. Argue that *any* ETL-derivable type T for t_l must contain an occurrence of ρ_0 . \square

Generally, we can prove the following soundness theorem.

1.3.3 THEOREM [SOUNDNESS OF ETL]: If $\vdash t :^{\varphi} T$ then evaluation of t does not get stuck. \square

2. The only substantial difference is Rule (TE-APP). It is more restrictive than the corresponding Rule (RT-APP) in the sense that it requires $\varphi_2 \subseteq \varphi$ in Rule (RT-APP) to be solved equationally. Note that, in general, the typing rules of TT in §1.5 are for con/decon completions (only). They can be derived from ETL by merging Rules (TE-FROM) and (TE-CELL) into the other rules.

We shall not prove this result here. The techniques will be presented in §1.5 for a more general type system.

- 1.3.4 EXERCISE [★★★★→]: Prove correctness for ETL. Do so by extending the Conditional Correctness Theorem and the Soundness Theorem for TL to ETL. □

Notes on effect type systems

Type and effect systems are introduced by Lucassen, Gifford and Jouvelot (Gifford and Lucassen, 1986; Lucassen, 1987; Lucassen and Gifford, 1988; Jouvelot and Gifford, 1989) for integrating imperative operations, notably updatable references and control effects, into functional languages. Type and effect inference using unification technology, which is the basis for region inference, is developed by Jouvelot and Gifford (1991); Talpin and Jouvelot (1992, 1994). Tofte and Talpin (1997) develop it into region inference for region-based memory management.

Nielson and Nielson (1994, 1996) pioneered type and effect systems with *behaviors* or *causal* effects, where effect types model order of evaluation. In such systems the language of effect expressions has operators for sequential composition and choice. They also provide for recursively defined effect expressions. The sequential composition operator captures the sequential order of the execution of effects. The choice operator corresponds to choice of one effect or another. This changes the nature of effects substantially as they basically turn into process algebras and thus have a nontrivial theory of their own. Modeling order of execution is key to capturing synchronization properties of concurrent processes, where atomic effects include the sending and receiving of messages. See Amtoft et al. (1999) and Nielson et al. (1999) for references on soundness, inference and applications.

Applications of type and effect systems include verification of cryptographic protocols by effect type checking (Gordon and Jeffrey, 2001b,a, 2002), behavior type systems for asynchronous programming (Igarashi and Kobayashi, 2001; Rajamani and Rehof, 2001; Chaki, Rajamani, and Rehof, 2002; Rajamani and Rehof, 2002) and interference analysis for concurrent threads (see Flanagan and Qadeer (2003) for references).

In terms of the computational λ -calculus of Moggi (1991), types are associated with values and effect types with *computations*; that is, intuitively, an effect type ${}^{\varphi}\mathbb{T}$ corresponds to an (effect indexed) monad type $\mathcal{M}^{\varphi}\mathbb{T}$. This connection is investigated by Semmelroth and Sabry (1999); Wadler (1998); Fluet (2004).

1.4 Region-based memory management

Region-based memory management is a particular way to manage the dynamically (or *heap*) allocated memory of a program. Traditionally, the heap is managed either explicitly by the programmer using constructs such as C's `malloc` and `free`, or automatically by a garbage collector leaving the programmer with only the responsibility of when to allocate memory. Region-based memory management uses explicit instructions for the allocation and deallocation of memory, but the safety of the explicit deallocations is guaranteed by a type system, and in some cases a compile-time analysis called "region inference" (§1.6) can insert the allocation and deallocation instructions automatically.

Basically a *region* is a sub-heap containing a number of heap-allocated values, and the heap is a collection of regions. A region starts out empty and grows when a value is allocated in it. A region can grow independently of the other regions constituting the heap; that is, one can allocate values in all regions currently available. Regions can only shrink when the complete region is deallocated; that is, one does not deallocate individual values, only complete regions.

In summary, we use three region primitives: (1) allocation of a new region, (2) allocation of a value in a region, and (3) deallocation of a complete region (and thereby all values allocated in the region). In contrast to TL in §1.2 we simply elide dereferencing.

A region-annotated language

Our region-annotated language is a lambda calculus with a fixed-point operator and (Boolean) constants, extended with explicit region annotations. Its syntax and evaluation semantics are defined in Figure 1-6. As usual, $\lambda x.t$ and $\text{fix } x.u$ binds the variable x in t and u , respectively. Similarly, $\lambda \rho.u$ and $\text{new } \rho.t$ binds the region variable ρ in u and t , respectively.

By analogy to §1.2 we define basic semantic notions for RAL. We write $t \xrightarrow{\text{RAL}} t'$ if $t \rightarrow t'$ can be derived from the RAL evaluation rules in Figure 1-6. A RAL-term t is *final* if there is no term t' such that $t \xrightarrow{\text{RAL}} t'$. Note that each value expression is final. All other final terms are *stuck*.

We write $t \xrightarrow{*} t'$ if $t \xrightarrow{\text{RAL}*} t'$ and t' is final. We write $t \downarrow$ if there exists t' such that $t \downarrow t'$. If t has no final state and thus does not terminate we write $t \uparrow$. the relation $\xrightarrow{*!}$ between terms is defined by: $t \xrightarrow{*!} t'$ if $t \xrightarrow{*} t'$ and t' is final.

1.4.1 DEFINITION: Define the function "eval_R(·)" from terms to {tt, ff, ⊥, wrong}

Terms		Evaluation	$t \xrightarrow{\text{RAL}} t'$
$t ::=$			
u	value or almost value	$t_1 \longrightarrow t'_1$	
x	variable	$\frac{t_1 \longrightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$	(RE-IF)
if t then t else t	conditional	if tt then t ₂ else t ₃ → t ₂ (RE-IFTRUE)	
fix x.u	recursion	if ff then t ₂ else t ₃ → t ₃ (RE-IFFALSE)	
t t	application	$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2}$	(RE-APP1)
t [p]	region app.	$\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2}$	(RE-APP2)
new ρ.t	region creation	$\lambda x.t \text{ at } \rho \longrightarrow \langle \lambda x.t \rangle_\rho$	(RE-CLOS)
u ::=	almost values:	$\langle \lambda x.t \rangle_\rho v \longrightarrow [x \mapsto v]t$	(RE-BETA)
v	value	$\frac{u \longrightarrow u'}{\text{fix } x.u \longrightarrow \text{fix } x.u'}$	(RE-FIX)
(λx.t) at p	abstraction	$\frac{\text{fix } x.v}{\longrightarrow [x \mapsto \text{fix } x.v]v}$	(RE-FIXBETA)
(λρ.u) at p	region abs.	$\frac{t \longrightarrow t'}{t [p] \longrightarrow t' [p]}$	(RE-RAPP)
v ::=	value expressions:	$\lambda \rho_1.u \text{ at } \rho_2 \longrightarrow \langle \lambda \rho_1.u \rangle_{\rho_2}$	(RE-RCLOS)
bv	truth value	$\langle \lambda \rho_1.u \rangle_{\rho_2} [p] \longrightarrow [\rho_1 \mapsto p]u$	(RE-RBETA)
⟨λx.t⟩ _p	closure	$\frac{t_1 \longrightarrow t'_1}{\text{new } \rho.t_1 \longrightarrow \text{new } \rho.t'_1}$	(RE-NEW)
⟨λρ.u⟩ _p	region closure	$\text{new } \rho.v \longrightarrow [\rho \mapsto \bullet]v$	(RE-DEALLOC)
bv ::=	truth values:		
tt	true		
ff	false		
p ::=	places:		
ρ	region variable		
•	deallocated		

Figure 1-6: Region-annotated language, RAL

by

- $\text{eval}_R(t_0) = \text{bv}$ iff $t_0 \rightarrow_R^{*!} \text{bv}$.
- $\text{eval}_R(t_0) = \perp$ iff there is an infinite sequence $t_1, t_2, \dots, t_i, \dots$ such that $t_i \xrightarrow{\text{RAL}} t_{i+1}$ for $0 \leq i$.
- $\text{eval}_R(t_0) = \text{wrong}$ iff $t_0 \rightarrow^{*!} t$ where t is not a value. \square

Recall that the new $\rho.t$ construct introduces a new region variable ρ . The

variable ρ can be used to annotate value-producing terms within τ . The *allocation* of the new region in our system is implicit; it happens automatically when the execution focus moves inside the new binder. Implicit alpha-conversion makes sure that the new does not capture any foreign region variables before the allocation. On the other hand, *deallocation* is explicit in the evaluation semantics. The (RE-DEALLOC) rule records the fact that a value stored in the deallocated region is no longer available by replacing the region variable with the special marker \bullet . The “dangling pointers” to deallocated values can be manipulated freely as long as one does not attempt to read the values they point to. At that point execution will get stuck, because there is no reduction rule for an expression of the form “ $\langle \lambda x. \tau \rangle_{\bullet} v$ ”. Rule (RE-BETA) that would ordinarily reduce it applies only when the place is a ρ , which explicitly does not include \bullet as in the effect typed language ETL.

Observe that the substitution $[\rho \mapsto \bullet]$ in (RE-DEALLOC) can affect allocation expressions (\dots) at ρ as well as already allocated values $\langle \dots \rangle_{\rho}$. In the former case we end up with a “ (\dots) at \bullet ” expression which asks to allocate something in a region that does not exist anymore. This is impossible, of course, but the *occurrence* of such a subterm is not an error. The error happens if the expression is eventually executed; in which case execution will get stuck because (RE-CLOS) and (RE-RCLOS) demand a ρ rather than a ρ after the “at”. Similarly a \bullet can appear as the actual parameter in a region application, and the application can even be reduced without an error.

A novel aspect of the region-annotated language, compared to the tagged language described previously, is the presence of *region abstractions*. The intention is that a region abstraction “ $\lambda \rho. u$ ”, where u is an “almost value (see Figure 1-6) ranging over normal values and yet-to-be-allocated abstractions, is the natural counter-part to a normal abstraction only at the level of regions. One can apply such an abstraction to an actual place parameter ρ in which case evaluation proceeds by substituting the place ρ for the formal parameter ρ in u , and then evaluates the result of this substitution. Region abstractions allow one to parameterize a function over the regions necessary for the evaluation of the function. Typically, this means parameterizing over the regions containing the input to the function, and the regions in which the output should be stored. We say that such a function is *region polymorphic* in the region parameters.

For example, consider the following program to compute Fibonacci numbers³

```
fix fib.  $\lambda n.$ 
```

3. In examples we shall allow ourselves to use features such as integers allocated in regions even though they are not part of the formal developments.

```

if n<2 then 1
else fib(n-2)+fib(n-1)

```

One possible region annotation of this program is (ignore everything but the first line for now)

```

fix fib. (λρi. (λρo. (λn.
  if new ρ. (n < (2 at ρ) then 1 at ρo)
  else new ρ1.
    new ρ2.fib[[ρ2]][[ρ1]] (new ρ.n -at ρ2 (2 at ρ))
    +at ρonew ρ3.fib[[ρ3]][[ρ1]] (new ρ.n -at ρ3 (1 at ρ))
  ) at ρi) at ρi) at ρf

```

The point is that the `fib` function expects two region parameters at runtime: one, ρ_i , in which the input `n` is stored, and one, ρ_o , in which the function is supposed to store its result. Thus, any caller of `fib` is required to choose appropriate actual regions for these as witnessed in the two calls to `fib` in the body.

Observe that, since the only way to allocate and deallocate a region is via the `new` construct, it is not possible for the function to deallocate a region associated with a parameter, and similarly, the function cannot itself allocate such a region. The consequence is that the lifetime of regions passed as parameters to a function encompasses the lifetime of the complete function invocation. In order to avoid large, long-lived regions it is therefore important to allow the body of a recursive function to use actual parameters to recursive invocations different from the formal parameters. This is referred to as *region polymorphic recursion* in the literature, as it allows us to choose different instantiations of the polymorphically bound region parameters for different invocations.

Continuing the Fibonacci example above, it is crucial that the two recursive calls to `fib` can each supply their own actual parameters (in this case ρ_2, ρ_1 and ρ_3, ρ_1). Thus, for each call we store the arguments in separate regions whose lifetimes are just the duration of the function call. The results need slightly longer lifetimes, since we need to add those up to give the result of the original call, but they can be stored in the same region. (The example is taken from Tofte and Talpin (1997).)

- 1.4.2 EXERCISE [★★]: What would happen to the region-behavior of the Fibonacci program if region polymorphic recursion were disallowed (ie., if the recursive calls were required to use the formal region parameters as actual region parameters)? □
- 1.4.2 SOLUTION: Both of the two recursive calls would have to specify ρ_i, ρ_o as actual parameters, and so all intermediate arguments and results would end

up in the same two regions; namely the two argument regions supplied to the function at the outermost level.

The original calculus proposed by Tofte and Talpin (let us call it the TT calculus) differs from our RAL in a number of ways. The most conspicuous difference is that the region-creation construct “new $\rho.t$ ” is written “letregion ρ in t ”. A more subtle one is that TT restricts the places where region abstractions and recursive function definitions can occur. Region abstractions are only allowed in the definition of recursive functions, and a recursive function definition must appear in a `let` binding. These restrictions are implicit in the syntax of TT—it combines recursion and region abstraction in a single combined construction

$$t ::= \text{letrec } f[\rho_1, \dots, \rho_k](x) \text{ at } \rho = t_1 \text{ in } t_2$$

which corresponds to the RAL expression

$$\text{let } f = \text{fix } f.(\lambda\rho_1, \dots, \rho_k, \rho'.(\lambda x.t_1) \text{ at } \rho') \text{ at } \rho \text{ in } t_2$$

Using the `letrec` as an abbreviation we can rewrite the Fibonacci example to the following program:

```
letrec fib[ $\rho_i, \rho_o$ ] (n) at  $\rho_f$  =
  if new  $\rho$ . (n < (2 at  $\rho$ ) then 1 at  $\rho_o$ )
  else
    new  $\rho_1$ .
      new  $\rho_2$ . fib[ $\rho_2, \rho_1$ ] (new  $\rho$ . n -at  $\rho_2$  (2 at  $\rho$ ))
      +at  $\rho_o$  new  $\rho_3$ . fib[ $\rho_3, \rho_1$ ] (new  $\rho$ . n -at  $\rho_3$  (1 at  $\rho$ ))
```

- 1.4.3 EXERCISE [★★]: This unfolding of the TT `letrec` uses abstraction over multiple regions at once, which is not actually part of the RAL calculus. Show how n -ary region abstractions can be simulated using our unary ones. \square
- 1.4.3 SOLUTION: An n -ary region abstraction can be converted into a stack of unary ones, but one must decide where the intermediate region closures that the semantics require are allocated. The following solution takes care not to cause any net heap allocation in the translation of an n -ary application:

$$(\lambda\rho_1, \dots, \rho_n.t) \text{ at } \rho \Rightarrow (\lambda\rho'.(\lambda\rho_1 \dots (\lambda\rho_n.t) \text{ at } \rho' \dots) \text{ at } \rho') \text{ at } \rho$$

$$f \llbracket \rho_1, \dots, \rho_n \rrbracket \Rightarrow \text{new } \rho'. f \llbracket \rho' \rrbracket \llbracket \rho_1 \rrbracket \dots \llbracket \rho_n \rrbracket$$

- 1.4.4 EXERCISE [★]: What is the role of the ρ' parameter in the above RAL expansion of TT's `letrec` construction? Can you guess why it is not part of the original TT syntax? \square

- 1.4.4 SOLUTION: When the region abstraction is applied (i.e., each time f is mentioned), a closure must be allocated to contain the region parameters and the free variables of the function body, because the ordinary BL parameter may not be supplied right away. The parameter ρ' selects the region in which this closure will be allocated. It is not part of the TT syntax for `letrec` because this closure allocation is implicit in the `letrec` construct; instead the original syntax for applying the region abstraction is

$$t ::= f[\rho_1, \dots, \rho_k] \text{ at } \rho'$$

which can be expressed as $f \llbracket \rho_1, \dots, \rho_k, \rho' \rrbracket$ in RAL.

- 1.4.5 EXERCISE [★]: What is the role of ρ in the `letrec` construction? Is it really operationally necessary? \square
- 1.4.5 SOLUTION: ρ is the region where a closure for the region abstraction is allocated. This closure contains the values of the free variables of the lambda expression. The intention in TT was that this closure would be consulted when the region abstraction is applied, such that the variables could be moved to the final closure in ρ' . However, due to the syntactic requirement that the region abstraction is applied whenever f is mentioned in t_1 or t_2 , the free variables will actually still be in scope at the application point. Since the body of the region abstraction is also statically known, nothing actually needs to be allocated in ρ , and indeed the ML Kit, a practical realization of TT (refer to Section 1.8), does not allocate this closure. But this was not realized when TT was first formulated.

The dynamic semantics presented by Tofte and Talpin (1997) goes to some length to stress that regions are allocated and deallocated according to a stack discipline. A *runtime configuration* contains a *region environment* that maps region variables to concrete regions (denoted by r), and a *store* that maps (concrete) regions to the values stored in them. Evaluation of a new $\rho.t$ then proceeds as follows: (1) first choose a fresh (concrete) region r and extend the region environment with a binding $\rho \mapsto r$ and the store with a binding $r \mapsto \emptyset$ where \emptyset is the *empty region* containing no values; (2) then proceed with the evaluation of t in this extended runtime configuration; (3) complete the evaluation of the entire term by removing the bindings for ρ and r from the configuration.

That original formulation is closer to an operational understanding of how the region operations work than the *store-less* semantics we use here. On the other hand, the store-less semantics is easier to reason about; a trick due to Helsen and Thiemann (2000) and Calcagno (2001). See Calcagno et al. (2002) for a proof that the two styles of semantics are indeed equivalent.

Reusing deallocated memory

Intuitively it should be safe to reuse deallocated memory (indicated by the special place \bullet) while executing a region-safe program. More formally, assume that t_\bullet is a term containing deallocated values and that t is constructed from t_\bullet by replacing some of these with new values. Then if t_\bullet evaluates to some value or loops indefinitely (ie., it does not go wrong), then so does t .

- 1.4.6 PROPOSITION: Let Val be the set of values and $Dead$ be the subset of values of the form " $\langle \dots \rangle_\bullet$ ". Let the relation \preceq between terms be the compatible closure of $Dead \times Val$. That is, $t_\bullet \preceq t$ if t arises from t_\bullet by replacing some (zero or more) deallocated values by arbitrary new values.

If $t_\bullet \preceq t$ and $\text{eval}_R(t_\bullet) = Y \neq \text{wrong}$, then $\text{eval}_R(t) = Y$, too. \square

Proof: Exercise (★★). \square

- 1.4.6 SOLUTION: By a simple induction over the derivation of the evaluation relation, we may prove that if $t_\bullet \xrightarrow{RAL} t'_\bullet$ and $t_\bullet \preceq t$, then there is a t' such that $t \xrightarrow{RAL} t'$ and $t'_\bullet \preceq t'$.

Apply this lemma to each step of the reduction of the original t_\bullet . In the case $Y = bv$, note that $bv \preceq t$ implies $bv = t$.

Annotating programs with regions preserves meanings

Region annotating a program is the process of adding region annotations to it to make the memory management explicit (see §1.6 for how to do this automatically). Thus, the process takes a program written in BL and produces a program written in RAL. The intention is, of course, that the region-annotated program is supposed to have the same behavior as the original program. In other words, we shall prove that adding region annotations preserves the meaning of the program. We make this precise in the present section. We do so by starting with a region-annotated program and showing that it behaves the same as the program obtained by removing all region annotations (thereby obtaining a program in BL, Figure 1-1),

Analogous to erasure for TL-terms, going from a term in the region-annotated language RAL to a term in the underlying base language BL is a matter of erasing all region annotations.

- 1.4.7 DEFINITION: The *erasure* $\|t\|$ of a region-annotated term t is the BL-term defined by removing the region annotations, as shown in Figure 1-7. \square

The ideal meaning-preservation statement would be: For any region-annotated program t , if $\text{eval}_R(t) = Y$ then $\text{eval}(\|t\|) = Y$ and vice versa. Unfortunately that is not true, since t can go wrong due to memory-management

$\begin{aligned} \llbracket bv \rrbracket &= bv \\ \llbracket \text{if } t_0 \text{ then } t_1 \text{ else } t_2 \rrbracket &= \text{if } \llbracket t_0 \rrbracket \text{ then } \llbracket t_1 \rrbracket \text{ else } \llbracket t_2 \rrbracket \\ \llbracket x \rrbracket &= x \\ \llbracket (\lambda x.t) \text{ at } p \rrbracket &= \lambda x. \llbracket t \rrbracket \\ \llbracket \langle \lambda x.t \rangle_p \rrbracket &= \lambda x. \llbracket t \rrbracket \end{aligned}$	$\begin{aligned} \llbracket t_1 t_2 \rrbracket &= \llbracket t_1 \rrbracket \llbracket t_2 \rrbracket \\ \llbracket \text{fix } x.u \rrbracket &= \text{fix } x. \llbracket u \rrbracket \\ \llbracket \text{new } \rho.t \rrbracket &= \llbracket t \rrbracket \\ \llbracket (\lambda \rho.u) \text{ at } p \rrbracket &= \llbracket u \rrbracket \\ \llbracket \langle \lambda \rho.u \rangle_p \rrbracket &= \llbracket u \rrbracket \\ \llbracket t \llbracket p \rrbracket \rrbracket &= \llbracket t \rrbracket \end{aligned}$
--	--

Figure 1-7: Definition of the erasure function

errors (such as trying to read a value after it has been deallocated) that have no counterpart in $\llbracket t \rrbracket$. What we *can* prove, however, is the following theorem:

- 1.4.8 THEOREM [CONDITIONAL CORRECTNESS]: Let t be a region-annotated program (formally: any term), and assume $\text{eval}_R(t) \neq \text{wrong}$. Then $\text{eval}_R(t) = \text{eval}(\llbracket t \rrbracket)$. \square

In other words, a region-annotated program behaves the same as the original unannotated program, unless it goes wrong. Our semantics for region-annotated programs does not allow us to distinguish between going wrong because of memory-management errors and going wrong due to plain old type errors, but it would be straightforward (though tedious) to extend the semantics with such a notion and then prove that if $\text{eval}(\llbracket t \rrbracket) \neq \text{eval}(t)$ then $\text{eval}(t)$ is memory-wrong rather than type-wrong. Since we are primarily concerned with well-typed programs, we will not pursue that further.

The proof of the theorem proceeds through a series of lemmas:

- 1.4.9 LEMMA: Assume that t is a value *or* an almost-value u . Then $\llbracket t \rrbracket$ is a value for BL. \square

Proof: By structural induction on t . The induction hypothesis is used in the case of region abstractions and closures, which disappear during erasure. (This is why the body of a region abstraction is restricted to be an almost-value rather than an arbitrary term). \square

- 1.4.10 LEMMA [SIMULATION]: Assume $t \xrightarrow{\text{RAL}} t'$. Then either (a) $\llbracket t \rrbracket \longrightarrow \llbracket t' \rrbracket$ or (b) $\llbracket t \rrbracket = \llbracket t' \rrbracket$. \square

Proof: By induction on the derivation of $t \xrightarrow{\text{RAL}} t'$.

For the rules (RE-IF), (RE-APP1), and (RE-APP2), apply the induction hypothesis. If this application yields case (a), use the corresponding context rule from BL. In the case of (RE-APP2), Lemma 1.4.9 ensures that erasure of the function expression is still a value, such that the corresponding BL rule is available.

For the rule (RE-FIX), first observe that since the body of the fix is an almost-value, the only rules that can establish the reduction $u \xrightarrow{\text{RAL}} u'$ are (RE-CLOS) and (RE-RCLOS). Then, by inspection of each of these rules we find $\|u\| = \|u'\|$, and thus also $\|\text{fix } x.u\| = \|\text{fix } x.u'\|$.

For (RE-IFTRUE), (RE-IFFALSE), (RE-BETA), and (RE-FIXBETA), the $\|t\| \longrightarrow \|t'\|$ case applies through the corresponding BL reductions.

For (RE-RAPP), and (RE-NEW), use the induction hypothesis directly.

For (RE-CLOS) and (RE-RCLOS), $\|t\| = \|t'\|$ holds trivially. Similarly for (RE-RBETA) and (RE-DEALLOC), because the erasure hides the effect of region substitutions. \square

- 1.4.11 LEMMA [SIMULATED PROGRESS]: Assume $t \xrightarrow{\text{RAL}} t'$ yet not $\|t\| \longrightarrow \|t'\|$. Then t' is strictly smaller than t , under a size measure where unevaluated abstractions are considered “larger” (e.g., twice as large) than closures. \square

Proof: From the proof of Lemma 1.4.10 it is clear that the derivation of $t \xrightarrow{\text{RAL}} t'$ must consist of a stack of context rules with one of the axioms (RE-CLOS), (RE-RCLOS), (RE-RBETA), or (RE-DEALLOC) at the top. Because the context rules do not themselves add material to the term, it is sufficient to check the lemma for those four axioms. For (RE-CLOS) and (RE-RCLOS), the size measure is explicitly defined to make the lemma true. For (RE-RBETA) or (RE-DEALLOC), the region substitution does not change the size of its argument, whereas the reductions remove either the λ or the new binder. \square

- 1.4.12 LEMMA: Assume $\text{eval}_R(t) = \text{bv}$. Then $\text{eval}(\|t\|) = \text{bv}$, too. \square

Proof: We have that $t \xrightarrow{*} \text{bv}$. By applying Lemma 1.4.10 to each of the reduction steps in turn, we get $\|t\| \xrightarrow{\text{BL}}^* \|\text{bv}\| = \text{bv}$. Since bv has no successor, $\text{eval}(\|t\|) = \text{bv}$. \square

- 1.4.13 LEMMA: Assume $\text{eval}_R(t_0) = \perp$. Then $\text{eval}(\|t_0\|) = \perp$, too. \square

Proof: The assumption gives us an infinite series of reductions

$$t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_i \dots$$

By Lemma 1.4.10, we get for each $i \geq 0$ that either $\|t_i\| = \|t_{i+1}\|$ or $\|t_i\| \xrightarrow{\text{BL}} \|t_{i+1}\|$. Lemma 1.4.11 guarantees that there does not exist an N such that

$\|t_i\| \not\stackrel{\text{BL}}{\rightarrow} \|t_{i+1}\|$ for all $i > N$. Therefore, by choosing certain i 's, we get an infinite series of BL reductions

$$\|t_0\| \stackrel{\text{BL}}{\rightarrow} \|t_{i_1}\| \stackrel{\text{BL}}{\rightarrow} \dots \stackrel{\text{BL}}{\rightarrow} \|t_{i_j}\| \dots$$

Hence $\text{eval}_R(\|t\|) = \perp$. □

Proof: [of Theorem 1.4.8] Assume that $\text{eval}_R(t) \neq \text{wrong}$. Then $\text{eval}_R(t)$ is either bv or \perp , and one of the last two lemmas gives us $\text{eval}(\|t\|) = \text{eval}_R(t)$. □

1.5 The Tofte–Talpin type system

One of the features that differentiates Tofte and Talpin's region language from other region-based systems (such as Hanson (1990), Ross (1967), and Schwartz (1975)) is the presence of a *type system*. The type system is sound (page 35) and thus well-typed programs do not go wrong at runtime. In the present setting, this means that if the term t is well-typed then $\text{eval}_R(t) \neq \text{wrong}$. In particular, well-typed programs are memory safe. In contrast to this, in the other systems mentioned above, the programmer has to establish memory safety manually and this is essentially just as hard as establishing memory safety of C-like `malloc/free` programs.

In Figure 1-8, we define a region type system for RAL, called RTL for "Region-Typed Language".

The judgment $\Gamma \vdash t :^\varphi T$ reads: in type environment Γ the term t has type T and effect φ . The effect captures the regions that have to be live (ie., allocated) for the term to evaluate without memory problems. In types, $\forall x.T$ binds the type variable x in T , $\Pi \rho.^{\varphi}T$ binds the region variable ρ in T and φ , and $\forall \epsilon.T$ binds the effect variable ϵ in T . We denote the sets of free type variables, free region variables, and free effect variables of a type T by $\text{ftv}(T)$, $\text{frv}(T)$, and $\text{fev}(T)$, respectively. These are extended to typing contexts in the obvious manner. We write $[X \mapsto T]$, $[\rho \mapsto p]$, and $[\epsilon \mapsto \varphi]$ for the capture-avoiding substitutions of type T for the type variable x , place p for the region variable ρ , and effect φ for the effect variable ϵ , respectively.

The typing rules in Figure 1-8 are natural extensions of the typing rules for the effect typed language in Figure 1-5 (page 20). The region-annotated language, is however, both type polymorphic and effect polymorphic. The type system includes standard rules for introducing and eliminating type polymorphism and the obvious variations for effect polymorphism. Compared to System F (TAPL Chapter 23) we do not have explicit syntax for these introductions and eliminations. As already mentioned, the language

<i>Type expressions</i>		
$p \in \text{Place}$	<i>places</i>	
$\epsilon \in \text{EffVar}$	<i>effect variables</i>	
$\varphi \in \mathcal{P}_{\text{fin}}(\text{Place} \cup \text{EffVar})$	<i>effects</i>	
$T ::=$	<i>type expressions:</i>	
x	<i>type variable</i>	
bool	<i>Boolean type</i>	
$(T \rightarrow {}^\varphi T, p)$	<i>function type</i>	
$(\Pi \rho. {}^\varphi T, p)$	<i>region func.</i>	
$\forall x.T$	<i>type polymorphism</i>	
$\forall \epsilon.T$	<i>effect polymorphism</i>	
<i>Typing rules</i>	$\boxed{\Gamma \vdash t : {}^\varphi T}$	
$\frac{\Gamma(x) = T}{\Gamma \vdash x : {}^\varphi T}$	(RT-VAR)	$\frac{\Gamma \vdash t_0 : {}^\varphi (T_1 \rightarrow {}^{\varphi_2} T_2, p) \quad \Gamma \vdash t_1 : {}^\varphi T_1 \quad p \in \varphi \quad \varphi_2 \subseteq \varphi}{\Gamma \vdash t_0 t_1 : {}^\varphi T_2}$ (RT-APP)
$\Gamma \vdash \text{bv} : {}^\varphi \text{bool}$	(RT-BOOL)	$\frac{\Gamma, x : T \vdash u : {}^\varphi T}{\Gamma \vdash \text{fix } x.u : {}^\varphi T}$ (RT-FIX)
$\frac{\Gamma \vdash t_1 : {}^\varphi \text{bool} \quad \Gamma \vdash t_2 : {}^\varphi T \quad \Gamma \vdash t_3 : {}^\varphi T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : {}^\varphi T}$	(RT-IF)	$\frac{\Gamma \vdash u : {}^{\varphi'} T \quad \rho \notin \text{frv}(\Gamma) \quad p \in \varphi}{\Gamma \vdash (\lambda \rho.u) \text{ at } p : {}^\varphi (\Pi \rho. {}^{\varphi'} T, p)}$ (RT-RABS)
$\frac{\Gamma, x : T_1 \vdash t : {}^{\varphi_2} T_2 \quad p \in \varphi}{\Gamma \vdash (\lambda x.t) \text{ at } p : {}^\varphi (T_1 \rightarrow {}^{\varphi_2} T_2, p)}$	(RT-ABS)	$\frac{\Gamma \vdash u : {}^{\varphi'} T \quad \rho \notin \text{frv}(\Gamma)}{\Gamma \vdash \langle \lambda \rho.u \rangle_p : {}^\varphi (\Pi \rho. {}^{\varphi'} T, p)}$ (RT-RCLOS)
$\frac{\Gamma, x : T_1 \vdash t : {}^{\varphi_2} T_2}{\Gamma \vdash \langle \lambda x.t \rangle_p : {}^\varphi (T_1 \rightarrow {}^{\varphi_2} T_2, p)}$	(RT-CLOS)	$\frac{p \in \varphi \quad [\rho \mapsto p'] \varphi' \subseteq \varphi}{\Gamma \vdash t \llbracket p' \rrbracket : {}^\varphi [\rho \mapsto p'] T}$ (RT-RAPP)
		$\frac{\Gamma \vdash t : {}^{\varphi, \rho} T \quad \rho \notin \text{frv}(\Gamma, T)}{\Gamma \vdash \text{new } \rho.t : {}^\varphi T}$ (RT-NEW)
		$\frac{\Gamma \vdash t : {}^\varphi T \quad x \notin \text{ftv}(\Gamma)}{\Gamma \vdash t : {}^\varphi \forall x.T}$ (RT-TGEN)
		$\frac{\Gamma \vdash t : {}^\varphi \forall x.T}{\Gamma \vdash t : {}^\varphi [x \mapsto T'] T}$ (RT-TINST)
		$\frac{\Gamma \vdash t : {}^\varphi T \quad \epsilon \notin \text{fev}(\Gamma, \varphi)}{\Gamma \vdash t : {}^\varphi \forall \epsilon.T}$ (RT-EGEN)
		$\frac{\Gamma \vdash t : {}^\varphi \forall \epsilon.T}{\Gamma \vdash t : {}^\varphi [\epsilon \mapsto \varphi'] T}$ (RT-EINST)

Figure 1-8: The RTL region type system

also contains region polymorphism. Region polymorphism is explicit in the syntax because it has operational significance.

Effect polymorphism is the natural complement to type polymorphism and higher-order functions. Consider a higher-order polymorphic function such as `list map`: it takes a function and a list as arguments and applies the function to each element in the list. In the region-free base language, `map` has type $\forall \alpha, \beta. (\alpha \rightarrow \beta) \times \alpha \text{ list} \rightarrow \beta \text{ list}$. What is the effect of applying the equivalent of `map` in the region-annotated language to such arguments? It

certainly has to include the effect. φ say, of applying the argument function, and thus the type of the region-annotated `map` function would have to reflect that in the latent effect of the complete function:

$$\forall \alpha, \beta. (\alpha \rightarrow^\varphi \beta) \times (\alpha \text{ list}, \rho) \rightarrow \{\rho, \rho'\} \cup \varphi (\beta \text{ list}, \rho')$$

(see page 40 for the typing rules concerning lists). However, that would only allow us to apply `map` to functions with latent effect φ . We could of course inspect the complete program and make sure that all effects were large enough that this is not a problem, but this approach would unnecessarily keep many regions alive. Instead, we can employ effect polymorphism to propagate the effect of the functional argument to the effect of the complete evaluation of `map` as in $\forall \alpha, \beta. \forall \epsilon. (\alpha \rightarrow^\epsilon \beta) \times (\alpha \text{ list}, \rho) \rightarrow \{\rho, \rho'\} \cup \epsilon (\beta \text{ list}, \rho')$.

The RTL type system is based on the type system of the TT calculus in Tofte and Talpin (1997).⁴ Compared to TT, RTL has moved effect enlargement upwards in the derivation tree so that the axioms are responsible for introducing proper effects. This simplifies both the presentation of the rules and the soundness proof slightly, and it is possible to establish a meta-property of the type system that allows effects to be enlarged. Moreover, the RTL system, with its System *F*-like polymorphism in types, regions and effects, is more permissive than the TT system with its let-polymorphism. The restrictions present in the original system were there to simplify the region inference algorithm.

The typing rules of RTL can be applied in sequence to obtain a typing of the `letrec` construction in the TT system. Recall, that a TT `letrec`

$$\text{letrec } f[\rho_1, \dots, \rho_k](x) \text{ at } \rho = t_1 \text{ in } t_2$$

is expressed in RAL as

$$\text{let } f = \text{fix } f. (\lambda \rho_1, \dots, \rho_k, \rho'. (\lambda x. t_1) \text{ at } \rho') \text{ at } \rho \text{ in } t_2$$

4. The type system in that paper is not defined explicitly. The paper presents a typed translation from a language resembling our Base Language BL to the TT calculus. From this translation one can extract a type system for TT.

The combined construction can be typed by a stack of RTL rules:

$$\begin{array}{c}
\frac{\Gamma, f \mapsto T_{12}, x \mapsto T \vdash t_1 :^{\varphi_1} T_1}{\Gamma, f \mapsto T_{12} \vdash t_{14} :^{\rho'} T_{14}} \text{ (RT-ABS)} \\
\frac{\Gamma, f \mapsto T_{12} \vdash t_{14} :^{\rho'} T_{14}}{\Gamma, f \mapsto T_{12} \vdash t_{13} :^{\varphi} T_{13}} \text{ (RT-RABS)} \\
\frac{\Gamma, f \mapsto T_{12} \vdash t_{13} :^{\varphi} T_{13}}{\vdots} \text{ (RT-EGEN)} \\
\frac{\vdots}{\Gamma, f \mapsto T_{12} \vdash t_{13} :^{\varphi} T_{12}} \text{ (RT-EGEN)} \\
\frac{\Gamma, f \mapsto T_{12} \vdash t_{13} :^{\varphi} T_{12}}{\Gamma \vdash t_{11} :^{\varphi} T_{12}} \text{ (RT-FIX)} \\
\frac{\Gamma \vdash t_{11} :^{\varphi} T_{12}}{\vdots} \text{ (RT-TGEN)} \\
\frac{\Gamma \vdash t_{11} :^{\varphi} T_{11} \quad \Gamma, f \mapsto T_{11} \vdash t_2 :^{\varphi} T_2}{\Gamma \vdash \text{letrec } f[\rho_1, \dots, \rho_k](x) \text{ at } \rho = t_1 \text{ in } t_2 :^{\varphi} T_2} \text{ (RT-LET)}
\end{array}$$

where

$$\begin{array}{ll}
t_{14} = (\lambda x. t_1) \text{ at } \rho' & T_{14} = (T \rightarrow^{\varphi_1} T_1, \rho') \\
t_{13} = (\lambda \rho_1, \dots, \rho_k, \rho'. t_{14}) \text{ at } \rho & T_{13} = (\prod \rho_1, \dots, \rho_k, \rho'. \rho' T_{14}, \rho) \\
t_{11} = \text{fix } f. t_{13} & T_{12} = \forall \epsilon_1 \dots \forall \epsilon_n. T_{13} \\
& T_{11} = \forall X_1 \dots \forall X_m. T_{12}
\end{array}$$

As usual with let-polymorphism, each time f is mentioned in t_2 , its type scheme must immediately be fully instantiated. This principle is extended to the effect and region abstractions; these must *also* be fully instantiated (applied, in the case of region abstraction) each time f is mentioned in t_2 or t_1 . Thus, the original TT type system does not allow expressions in general to have type $(\prod \rho. \varphi T, \rho)$; in particular region abstractions cannot be passed as parameters to, or returned from, functions.

Syntactic type soundness

We will now prove that typable programs are memory safe. We do so by establishing *type soundness*, ie. that well-typed programs do not go wrong. The type soundness proof is structured as a standard sequence of Substitution, Subject Reduction, and Progress lemmas. This approach was pioneered by Helsen and Thiemann (2000) in the context of region-based languages, and apparently independently discovered by Calcagno (2001) for a big-step semantics. Tofte and Talpin (1997) also proved type soundness, albeit not directly but as a consequence of their “correctness theorem” of region inference, and using a complex co-inductive proof technique.

As usual, we start by showing that one can massage derivations so that the typing context only mentions the free variables of the term, and so that the derivation does not end with one of the instantiation rules.

- 1.5.1 LEMMA: If $\Gamma \vdash t :^\varphi T$, $\text{dom}(\Gamma') = \text{fv}(t)$, and Γ and Γ' agree when both are defined, then $\Gamma' \vdash t :^\varphi T$. \square

Proof: Straightforward induction on the typing derivation. \square

- 1.5.2 LEMMA: Let S be a substitution of the form $[\rho \mapsto p]$, $[\epsilon \mapsto \varphi]$, or $[X \mapsto T]$. If $\Gamma \vdash t :^\varphi T$ can be derived in n steps, then likewise can $S\Gamma \vdash S t :^{S\varphi} S T$. \square

Proof: Left as an exercise (\star). \square

- 1.5.2 SOLUTION: The only interesting issue in the proof is that the substitutions substitute from sets of variables to another syntactic class (from region variables to places, for example). Observe, however, that everywhere in the typing rules something is required to be a type, region, or effect variable (rather than, say, a place or effect), it occurs in a binding context and so is unaffected by substitution.

- 1.5.3 LEMMA: Assume that $\Gamma \vdash v :^\varphi T$ has a derivation in n steps. Then it has a derivation in at most n steps where the last rule used is neither (RT-TINST) nor (RT-EINST). \square

Proof: By induction on n . All but the cases for (RT-TINST) and (RT-EINST) either are direct or are simple applications of the induction hypothesis. Thus, assume that the derivation ends with (RT-EINST) (the case for (RT-TINST) is similar). Apply the induction hypothesis to the derivation of its premiss; this gives a derivation that concludes in a type with the shape $\forall \epsilon. T'$ yet does not end with (RT-EINST) or (RT-TINST). It cannot end with (RT-VAR), (RT-IF), (RT-APP), (RT-FIX), (RT-RAPP), or (RT-NEW) either, because variables, conditionals, applications, fixed-points, region applications, and region creations are not values. The only other rule that can conclude $\forall \epsilon. T'$ is (RT-EGEN), so the whole derivation now must end with

$$\frac{\Gamma \vdash v :^\varphi T' \quad \epsilon \notin \text{fv}(\Gamma, \varphi)}{\Gamma \vdash v :^\varphi \forall \epsilon. T'} \text{ (RT-EGEN)}$$

$$\frac{\Gamma \vdash v :^\varphi \forall \epsilon. T'}{\Gamma \vdash v :^\varphi T} \text{ (RT-EINST)}$$

where $T = [\epsilon \mapsto \varphi'] T'$ for some φ' .

Lemma 1.5.2 now gives a derivation of $[\epsilon \mapsto \varphi'] \Gamma \vdash v :^{[\epsilon \mapsto \varphi'] \varphi} [\epsilon \mapsto \varphi'] T'$ in $n - 2$ steps. But since $\epsilon \notin \text{fv}(\Gamma, \varphi)$, this conclusion is the same as $\Gamma \vdash v :^\varphi T$, so we can use that instead of the original derivation. Now apply the induction hypothesis to this new derivation. \square

1.5.4 LEMMA [CANONICAL FORMS]:

1. if v is a value of type `bool`, then v is of the form bv ;
2. if v is a value of type $(T_1 \rightarrow {}^\varphi T_2, p)$ then v is of the form $\langle \lambda x. t \rangle_p$;
3. if v is a value of type $(\Pi \rho. {}^\varphi T, p)$ then v is of the form $\langle \lambda \rho. u \rangle_p$. □

Proof: Left as an exercise ($\star \leftrightarrow$). □

We next prove some lemmas about effects. Firstly, one can always enlarge the effect attributed to a term and obtain a derivable typing. Secondly, if a value can be typed, then it can also be typed with an empty effect; that is, evaluation of values does not cause any observable effects.

1.5.5 LEMMA: If $\Gamma \vdash t : {}^\varphi T$ and $\varphi \subseteq \varphi'$, then $\Gamma \vdash t : {}^{\varphi'} T$. □

Proof: Straightforward induction on the typing derivation. □

1.5.6 LEMMA: Let v be a value. If $\Gamma \vdash v : {}^\varphi T$ then $\Gamma \vdash v : {}^\emptyset T$. □

Proof: From Lemma 1.5.3 we get a derivation of $\Gamma \vdash v : {}^\varphi T$ that ends with neither (RT-TINST) nor (RT-EINST). Therefore, the derivation must end with one of the typing rules for values. By inspecting each of the rules (RT-BOOL), (RT-CLOS), and (RT-RCLOS) for values we see that we can construct a derivation of $\Gamma \vdash v : {}^\emptyset T$ (by choosing the effect to be empty in each case). □

Having established these basic lemmas we can now prove the lemmas leading to the type soundness result.

1.5.7 LEMMA [SUBSTITUTION]: If $\Gamma, x_1 : T_1 \vdash t : {}^\varphi T$ and $\Gamma \vdash t_1 : {}^\emptyset T_1$, then $\Gamma \vdash [x_1 \mapsto t_1]t : {}^\varphi T$. □

Proof: By induction on the typing derivation for t . The cases are all standard, except for (RT-VAR) where $t = x_1$. By (RT-VAR) itself, $T = T_1$ and since $[x_1 \mapsto t_1]x_1 = t_1$ the second assumption combined with Lemma 1.5.5 gives us the desired derivation. □

1.5.8 PROPOSITION [SUBJECT REDUCTION]: If $\Gamma \vdash t : {}^\varphi T$ and $t \xrightarrow{\text{RAL}} t'$ then $\Gamma \vdash t' : {}^\varphi T$. □

Proof: By induction over the typing derivation.

The rules (RT-VAR), (RT-BOOL), (RT-CLOS), and (RT-RCLOS) cannot occur; these rules require t to have a shape that makes $t \xrightarrow{\text{RAL}} t'$ impossible.

The cases for the rule (RT-ABS) and (RT-RABS) are immediate; the evaluation step must be by rule (RE-CLOS) or (RE-RCLOS), and we can immediately construct typings for the reduct using (RT-CLOS) or (RT-RCLOS).

For the rule (RT-IF), use case analysis on the last step in the derivation of $\tau \xrightarrow{\text{RAL}} \tau'$. The only possibilities are (RE-IF), (RE-IFTRUE), and (RE-IFFALSE). In the two latter cases τ' is one of the branches of the conditional, and the sought-for conclusion is already one of the premises of (RT-IF). In the case of (RE-IF), use the induction hypothesis on the typing of the condition; by reusing the existing typing derivations for the branches we can produce a typing for τ' using (RT-IF) again.

For the rule (RT-APP), again use case analysis on the derivation of $\tau \xrightarrow{\text{RAL}} \tau'$. The possible rules are now (RE-APP1), (RE-APP2), and (RE-BETA). The two former are analogous to (RE-IF) above, so consider (RE-BETA). The term τ must be of the form $\langle \lambda x. \tau_b \rangle_\rho v$, and the premises to (RT-APP) are (1) $\Gamma \vdash \langle \lambda x. \tau_b \rangle_\rho :^\varphi (T' \rightarrow \varphi' T, \rho)$ and (2) $\Gamma \vdash v :^\varphi T'$, where further $\rho \in \varphi$ and $\varphi' \subseteq \varphi$. Now, $\langle \lambda x. \tau_b \rangle_\rho$ is a value and thus by Lemma 1.5.3 there exists a derivation of (1) ending with (RT-CLOS) which must include a (sub-)derivation of (3) $\Gamma, x : T' \vdash \tau_b :^{\varphi'} T$. Applying first Lemma 1.5.6 to (2) yields $\Gamma \vdash v :^\emptyset T'$ and then applying Lemma 1.5.7 to (3) above and this, we get a derivation of $\Gamma \vdash [x \mapsto v] \tau_b :^{\varphi'} T$ as required because $\varphi' \subseteq \varphi$ and we can enlarge the effect (Lemma 1.5.5).

For the rule (RT-FIX), the reduction must be by (RE-FIX) or (RE-FIXBETA). Again, the case for (RE-FIX) is analogous to (RE-IF), so assume (RE-FIXBETA). The term τ must be of the form $\text{fix } x.v$ and the typing derivation has a sub-derivation of (1) $\Gamma, x : T \vdash v :^\varphi T$. First, apply Lemma 1.5.6 to get $\Gamma, x : T \vdash v :^\emptyset T$ and use this and (RT-FIX) to construct a derivation of (2) $\Gamma \vdash \text{fix } x.v :^\emptyset T$. Now, Lemma 1.5.7 applied to (1) and (2) yields $\Gamma \vdash [x \mapsto \text{fix } x.v] v :^\varphi T$.

For the rule (RT-RAPP), the reduction must be by (RE-RAPP) (similar to the other context rules above) or (RE-RBETA). The term τ must be of the form $\langle \lambda \rho_1. u \rangle_\rho \llbracket p \rrbracket$, and the typing derivation has a sub-derivation of (1) $\Gamma \vdash \langle \lambda \rho_1. u \rangle_\rho :^\varphi (\Pi \rho_1. \varphi' T', \rho)$ where further $\rho \in \varphi$, $[\rho_1 \mapsto p] \varphi' \subseteq \varphi$, and $T = [\rho_1 \mapsto p] T'$. Now, $\langle \lambda \rho_1. u \rangle_\rho$ is a value and thus by Lemma 1.5.3 there exists a derivation of (1) ending with (RT-RCLOS) which must include a sub-derivation of (2) $\Gamma \vdash u :^{\varphi'} T'$ where $\rho_1 \notin \text{frv}(\Gamma)$. Applying Lemma 1.5.2 to (2) and $[\rho_1 \mapsto p]$, we get a derivation of $[\rho_1 \mapsto p] \Gamma \vdash [\rho_1 \mapsto p] u :^{[\rho_1 \mapsto p] \varphi'} [\rho_1 \mapsto p] T'$ as required because $[\rho_1 \mapsto p] \Gamma = \Gamma$ (since $\rho_1 \notin \text{frv}(\Gamma)$), $[\rho_1 \mapsto p] \varphi' \subseteq \varphi$ which we can enlarge (Lemma 1.5.5), and $T = [\rho_1 \mapsto p] T'$.

For the rule (RT-NEW), τ reduces by (RE-NEW) or (RE-DEALLOC). Only the latter case is interesting. The term τ must be of the form $\text{new } \rho.v$ and the typing derivation must include a derivation of $\Gamma \vdash v :^{\varphi, \rho} T$ where $\rho \notin \text{frv}(\Gamma, T)$.

Then, by Lemma 1.5.6 we have $\Gamma \vdash v :^{\emptyset} T$. Applying Lemma 1.5.2 we thus get a derivation of $[\rho \mapsto \bullet] \Gamma \vdash [\rho \mapsto \bullet] v :^{[\rho \mapsto \bullet] \emptyset} [\rho \mapsto \bullet] T$, that is $\Gamma \vdash [\rho \mapsto \bullet] v :^{\emptyset} T$ (since $\rho \notin \text{frv}(\Gamma, T)$). Now use Lemma 1.5.5 to recover the original effect φ .

For the rule (RT-TGEN) (the case for rule (RT-EGEN) is analogous) we have a derivation of $\Gamma \vdash t :^{\varphi} T'$ where $x \notin \text{ftv}(\Gamma)$ and $T = \forall x. T'$. Apply the induction hypothesis to this to get $\Gamma \vdash t' :^{\varphi} T'$. Since still $x \notin \text{ftv}(\Gamma)$ we can use (RT-TGEN) to construct a derivation of $\Gamma \vdash t' :^{\varphi} T$ where $T = \forall x. T'$ as required.

For the rule (RT-TINST) (the case for rule (RT-EINST) is analogous) we have a derivation of $\Gamma \vdash t :^{\varphi} \forall x. T'$ where $T = [x \mapsto T''] T'$ for some T'' . Again apply the induction hypothesis to this and get $\Gamma \vdash t' :^{\varphi} \forall x. T'$ and use (RT-TINST) to construct a derivation of $\Gamma \vdash t' :^{\varphi} T$ where $T = [x \mapsto T''] T'$ as required.

□

1.5.9 PROPOSITION [PROGRESS]: If $\emptyset \vdash t :^{\varphi} T$ and $\bullet \notin \varphi$, then either t is a value or there is some t' such that $t \xrightarrow{\text{RAL}} t'$. □

Proof: By induction over the typing derivation of $\emptyset \vdash t :^{\varphi} T$.

If the last rule in the derivation is (RT-TGEN), (RT-EGEN), (RT-TINST), or (RT-EINST), the conclusion follows directly from the induction hypothesis (which is applicable since the typing context remains empty and the effect remains the same in each premise). For example, for (RT-TGEN) we have a derivation of $\emptyset \vdash t :^{\varphi} T'$ where $T = \forall x. T'$. Applying the induction hypothesis to this we get that either t is a value, or there exists some t' such that $t \xrightarrow{\text{RAL}} t'$ as required.

The case (RT-VAR) is impossible (since Γ is empty by assumption).

The cases for (RT-BOOL), (RT-CLOS), and (RT-RCLOS) are immediate.

For (RT-ABS) and (RT-RABS), there are immediate reductions by (RE-CLOS) or (RE-RCLOS), respectively.

For (RT-FIX): If u in $\text{fix } x. u$ is a value v then (RE-FIXBETA) applies and we have a reduction. If u is an abstraction or a region abstraction, then it itself reduces (by either (RE-CLOS) or (RE-RCLOS)), and we obtain a reduction by (RE-FIX). This exhausts the possible syntactic forms of a non-value u .

For (RT-APP), t is an application $t_0 t_1$, and we have derivations of $\emptyset \vdash t_0 :^{\varphi} (T_1 \rightarrow \varphi^2 T, p)$ and $\emptyset \vdash t_1 :^{\varphi} T_1$ for some $p \in \varphi$ and $\varphi_2 \subseteq \varphi$. Apply the induction hypothesis to each of these. This gives either a reduction for at least one of them (in which case we can reduce t by (RE-APP1) or (RE-APP2)), or that t_1 and t_2 are both values. In this latter case, by Lemma 1.5.4 t_0 must have the form $\langle \lambda x_0. t'_0 \rangle_p$. Now we can apply the (RE-BETA) rule to obtain a reduction, because $p \in \varphi$ cannot be \bullet and so must be a region variable.

The cases for (RT-RAPP) and (RT-IF) are similar, but simpler.

For (RT-NEW), t is $\text{new } \rho.t_1$, and we have a derivation of $\emptyset \vdash t_1 :^{\varphi, \rho} T$. Since ρ is (by definition) not \bullet , the induction hypothesis applies to t_1 . We get that either t_1 is a value, in which case we get a reduction using (RE-DEALLOC), or t_1 reduces to another term t'_1 , in which case we get a reduction using (RE-NEW). \square

- 1.5.10 THEOREM: If $\emptyset \vdash t :^{\emptyset} T$, then either (1) there is some value v such that $t \rightarrow^* v$ and $\emptyset \vdash v :^{\emptyset} T$, or (2) for each t' such that $t \rightarrow^* t'$ there is some t'' such that $t' \rightarrow^+ t''$. \square

Proof: Straightforward consequence of Propositions 1.5.8 and 1.5.9. \square

- 1.5.11 COROLLARY [TYPE SOUNDNESS]: If $\emptyset \vdash t :^{\emptyset} \text{bool}$, then it is not the case that $\text{eval}_R(t) = \text{wrong}$. \square

Extensions

The region-annotated language that we have presented so far only has Booleans and functions, but it is straightforward to add most other common types of data to it. As an example, in Figure 1-9 we give the necessary rules for extending the system with lists. The proofs of the metaproperties (in particular type soundness and conditional correctness) carry through to this extended system without any changes to the existing cases.

Rule (RT-CONS) implies that when adding a new element in front of a list, it must have the same type as the elements already there. That is hardly surprising, but note that it implies that the region part of the type must also be the same. Thus, the different elements of a list are always allocated in the same region and therefore will be deallocated at the same time. If a single element of the list turns out to have a long lifetime, the region type system propagates that long lifetime to all the other elements!

- 1.5.12 EXERCISE [RECOMMENDED, ★★ \rightarrow]: Using Figure 1-9 as a guideline, write rules to extend the system with one or more of: Let bindings (a lambda abstraction allocates a closure on the heap, so a let binding cannot simply be simulated as a β -redex). Pairs and records. Sums and variants. General recursive types (equi- or iso-). Verify that the Conditional Correctness and Type Soundness theorems still hold for your rules. \square
- 1.5.13 EXERCISE [★★★★]: References can be added easily to the type system with rules like

$$\frac{\Gamma \vdash t :^{\varphi} T \quad p \in \varphi}{\Gamma \vdash \text{ref } t \text{ at } p :^{\varphi} (T \text{ ref}, p)} \quad (\text{RT-REF})$$

<p><i>New syntactic forms</i></p> <p>$t ::= \dots$ $(t :: t) \text{ at } p$ <i>terms:</i> $\text{case } t_0 \text{ of } \begin{matrix} \text{nil} \Rightarrow t_1 \\ (x :: x') \Rightarrow t_2 \end{matrix}$ <i>list constructor</i> <i>case on lists</i></p> <p>$v ::= \dots$ $\langle v :: v \rangle_p$ <i>values:</i> nil <i>cons cell</i> <i>empty list</i></p> <p>$T ::= \dots$ $(T \text{ list}, p)$ <i>types:</i> <i>type of lists</i></p> <p><i>New erasure rules</i></p> $\begin{aligned} \ \text{nil}\ &= \text{nil} \\ \ (t_1 :: t_2) \text{ at } p\ &= \ t_1\ :: \ t_2\ \\ \ \langle t_1 :: t_2 \rangle_p\ &= \ t_1\ :: \ t_2\ \\ \ \text{case } t_0 \text{ of } \begin{matrix} \text{nil} \Rightarrow t_1 \\ (x :: x') \Rightarrow t_2 \end{matrix}\ &= \\ \ \text{case } \ t_0\ \text{ of } \begin{matrix} \text{nil} \Rightarrow \ t_1\ \\ (x :: x') \Rightarrow \ t_2\ \end{matrix}\ & \end{aligned}$ <p><i>New evaluation rules</i> $t \xrightarrow{\text{RAL}} t'$</p> $\frac{t_1 \xrightarrow{\text{RAL}} t'_1}{(t_1 :: t_2) \text{ at } p \xrightarrow{\text{RAL}} (t'_1 :: t_2) \text{ at } p} \text{ (E-CONS1)}$ $\frac{t_2 \xrightarrow{\text{RAL}} t'_2}{(v_1 :: t_2) \text{ at } p \xrightarrow{\text{RAL}} (v_1 :: t'_2) \text{ at } p} \text{ (E-CONS2)}$	$\frac{(v_1 :: v_2) \text{ at } p \xrightarrow{\text{RAL}} \langle v_1 :: v_2 \rangle_p}{\text{ (E-CONSALLOC)}}$ $\frac{t_0 \xrightarrow{\text{RAL}} t'_0}{\text{case } t_0 \text{ of nil} \Rightarrow t_1 \mid (x :: x') \Rightarrow t_2 \xrightarrow{\text{RAL}} \text{case } t'_0 \text{ of nil} \Rightarrow t_1 \mid (x :: x') \Rightarrow t_2} \text{ (E-CASE)}$ $\text{case nil of nil} \Rightarrow t_1 \mid (x :: x') \Rightarrow t_2 \xrightarrow{\text{RAL}} t_1 \text{ (E-CASENIL)}$ $\text{case } \langle v :: v' \rangle_p \text{ of nil} \Rightarrow t_1 \mid (x :: x') \Rightarrow t_2 \xrightarrow{\text{RAL}} [x' \mapsto v'] [x \mapsto v] t_2 \text{ (E-CASECONS)}$ <p><i>New typing rules</i> $\boxed{\Gamma \vdash t :^\varphi T}$</p> $\frac{}{\Gamma \vdash \text{nil} :^\varphi (T \text{ list}, p)} \text{ (RT-NIL)}$ $\frac{\Gamma \vdash t_1 :^\varphi T}{\Gamma \vdash t_1 :^\varphi T} \text{ (RT-CONS)}$ $\frac{\Gamma \vdash t_2 :^\varphi (T \text{ list}, p) \quad p \in \varphi}{\Gamma \vdash (t_1 :: t_2) \text{ at } p :^\varphi (T \text{ list}, p)} \text{ (RT-CONS)}$ $\frac{\Gamma \vdash v_1 :^\varphi T \quad \Gamma \vdash v_2 :^\varphi (T \text{ list}, p)}{\Gamma \vdash \langle v_1 :: v_2 \rangle_p :^\varphi (T \text{ list}, p)} \text{ (RT-CONSCCELL)}$ $\frac{\Gamma \vdash t_0 :^\varphi T' \quad T' = (T \text{ list}, p) \quad p \in \varphi \quad \Gamma \vdash t_1 :^\varphi T'' \quad \Gamma, x : T, x' : T' \vdash t_2 :^\varphi T''}{\Gamma \vdash \text{case } t_0 \text{ of } \begin{matrix} \text{nil} \Rightarrow t_1 \\ (x :: x') \Rightarrow t_2 \end{matrix} :^\varphi T''} \text{ (RT-CASE)}$
---	---

Figure 1-9: Extending the system with a list type

$$\frac{\Gamma \vdash t :^\varphi (T \text{ ref}, p) \quad p \in \varphi}{\Gamma \vdash !t :^\varphi T} \text{ (RT-DEREF)}$$

$$\frac{\Gamma \vdash t :^\varphi (T \text{ ref}, p) \quad \Gamma \vdash t' :^\varphi T \quad p \in \varphi}{\Gamma \vdash t := t' :^\varphi \text{unit}} \text{ (RT-ASSIGN)}$$

Of course, these rules need to be combined with the usual value restriction on type *and effect* polymorphism, as discussed on pages 335–336 in TAPL. Which extensions to the semantics and soundness proofs are necessary for proving these rules sound? \square

1.5.13 SOLUTION: The reference operations would need a formal semantics, so we

would have to extend the evaluation and typing judgments with stores and store typings as in Chapter 13 of TAPL. But that would break the lexical scoping of region variables, on which the correct operation of rule (RE-DEALLOC) depends critically. So the entire semantic treatment of region allocation and deallocation needs to be reworked. How to do this can be seen in Calcagno et al. (2002).

The typing rules presented in the above exercise correspond exactly to the way updatable references are handled in the ML Kit. Observe that since the (RT-ASSIGN) rule demands equality between the type of the value stored in the reference and the type of the new value, the rule forces the two values to have the same lifetime. For long-lived references, such as those found with container-classes in object-oriented programs, this behavior is inadequate. No better solution has, however, been proposed yet.

1.6 Region inference

So far, we have said a lot about the region type system and how region-annotated programs are supposed to be executed, but next to nothing about where the region annotations come from.

One easy answer would be, “why, the programmer wrote them”—but alas, this answer would not be easy for the programmer. Realistic programs usually need quite a lot of region abstractions and applications in order to distribute their data over several regions while still being region typeable, and it is not always obvious exactly where they should be put. While it just might be possible to *write* a nontrivial well-typed program in the region-annotated language, it would be quite impossible to *maintain* it.

Therefore, the idea of using a region type system to check the safety of region annotations goes hand in hand with the idea that the region annotations themselves are the product of an automatic compile-time analysis. The human programmer writes a program t in BL, whereupon the compiler will construct a region-annotated program t' such that $\|t'\| = t$ and t' is well-typed in RTL. This process is known as *region inference*, because Tofte and Talpin (1994) viewed it as kin to a type reconstruction (“type inference”) problem.

- 1.6.1 EXERCISE [RECOMMENDED, ★]: One can easily formulate a “trivial” region inference: Just choose a single fixed ρ , annotate each lambda abstraction (and other allocating expressions) in the input program with “at ρ ”, and then wrap the entire program in a single new construction. This evidently always produces a region-annotated program that erases to the input program, but will it always be RTL-typeable? \square

- 1.6.1 SOLUTION: No. It is typeable (if and) only if the input program satisfies our syntactic restriction on the use of the `fix` operator, and is typeable in (region-free) F with recursion, such that the type of the entire program is either `bool` or a type variable. If the input program is ill-typed, it can never be region annotated; a derivation of $\emptyset \vdash t :^\varphi T$ can be converted into a derivation of $\emptyset \vdash \|t\| : \|T\|$ in System F with recursion simply by erasing all of the region-related syntax. ($\|T\|$, is of course, T with the region annotations removed, in a way similar to $\|t\|$). Such an erasure transforms each RTL type rule into either a well-known F rule or the identity rule that concludes any judgment from itself.

Of course, the trivial region inference is worthless from a memory-management point of view. It produces a region-annotated program that never deallocates anything until the entire computation is finished. What we want is the opposite: Region annotations that deallocate data as soon as allowed by the region type system. Unfortunately it is not known whether “best possible region annotations” in this sense always exist, but good approximate solutions are available.

The articles by Tofte and Talpin (1994, 1997) do not themselves present an algorithm for region inference, but the nondeterministic “region inference system” they present has evidently been constructed with an inference algorithm in mind. The inference algorithm was published by Tofte and Birkedal (1998). We do not have the space to describe it in detail, but instead show how it works in the context of an example. Consider the term

```
letrec m(f) = if f(0) then 0 else m(λx.f(x+1)) + 1
in m(λx.x=10)
```

Let us initially assume an that oracle has told us that the “correct” region-polymorphic type scheme for m is

$$\forall \epsilon_1, \epsilon_2. \Pi \rho_1, \rho_2. \{ \rho_2 \} ((\text{int} \rightarrow \{ \epsilon_1 \} \text{bool}, \rho_1) \rightarrow \{ \epsilon_2, \epsilon_1, \rho_1 \} \text{int}, \rho_2).$$

The driving force in the region inference algorithm is an attempt to construct the RTL typing tree for the region-annotated program. The search proceeds much like the familiar Algorithm \mathcal{W} , unifying types as well as region variables and effect positions as we go. We don’t know yet what to with typing-rule premises of the form $\rho \in \varphi$ or $\varphi \subseteq \varphi'$, so let us initially just collect them for further processing.

By the time we have analysed the subexpression $m(\lambda x.f(x+1))$, the unification-based inference has built the typing tree shown in Figure 1-10. (For brevity, we assume a primitive rule for adding one to an integer—it allows concluding $\Gamma \vdash t+1 :^\varphi \text{int}$ from $\Gamma \vdash t :^\varphi \text{int}$).

$$\begin{array}{c}
\frac{\frac{\frac{\Gamma' \vdash x : \wp^4 \text{ int}}{\Gamma' \vdash x+1 : \wp^4 \text{ int}}}{\Gamma' \vdash f : \wp^1 (\text{int} \rightarrow \wp^3 \text{ bool}, \rho_3)} \quad \frac{\Gamma' \vdash f(x+1) : \wp^4 \text{ bool}}{\Gamma \vdash (\lambda x. f(x+1)) \text{ at } \rho_4 : \wp (\text{int} \rightarrow \wp^4 \text{ bool}, \rho_4)}}{\Gamma \vdash m \llbracket \rho_4, \rho_5 \rrbracket : \wp ((\text{int} \rightarrow \wp^4 \text{ bool}, \rho_4) \rightarrow \wp^5 \text{ int}, \rho_5)} \quad \frac{\Gamma \vdash m \llbracket \rho_4, \rho_5 \rrbracket ((\lambda x. f(x+1)) \text{ at } \rho_4) : \wp \text{ int}}{\Gamma \vdash m \llbracket \rho_4, \rho_5 \rrbracket ((\lambda x. f(x+1)) \text{ at } \rho_4) : \wp \text{ int}} \\
\vdots
\end{array}$$

where Γ is $m : \forall \epsilon_1, \epsilon_2. \Pi \rho_1, \rho_2. \{ \rho_2 \} ((\text{int} \rightarrow \{ \epsilon_1 \} \text{ bool}, \rho_1) \rightarrow \{ \epsilon_2, \epsilon_1, \rho_1 \} \text{ int}, \rho_2)$, $f : (\text{int} \rightarrow \wp^3 \text{ bool}, \rho_3)$ and Γ' is $\Gamma, x : \text{int}$.

Collected effect constraints: $\wp_4 \subseteq \wp_5, \rho_4 \in \wp_5, \{ \rho_5 \} \subseteq \wp, \wp_3 \subseteq \wp_4, \rho_3 \in \wp_4, \rho_4 \in \wp, \wp_5 \subseteq \wp, \rho_5 \in \wp$.

Figure 1-10: A partially region-inferred proof tree.

A set of collected effect constraints are also shown on the figure. Luckily the effect polymorphism in the oracle's type scheme has a form that allows the possible instantiations of the effect fields in it to be described with subset and inclusion constraints; in this case $\wp_4 \subseteq \wp_5$ and $\rho_4 \in \wp_5$. (Exercise [★→]: Locate where in the proof tree the other collected constraints come from).

Whenever we choose concrete substitutions for the symbols \wp , \wp_3 , \wp_4 , and \wp_5 that satisfy the constraints, we get a valid proof. What it is proof of depends on the substitutions we choose for \wp and \wp_3 , because these effect meta-variables occur in the conclusion. On the other hand, \wp_4 and \wp_5 do not occur in the conclusion, so we can eliminate them from the constraint set and simplify it to $\{ \rho_4 \in \wp, \rho_5 \in \wp, \rho_3 \in \wp, \wp_3 \subseteq \wp \}$. Much of (Tofte and Birkedal, 1998) is concerned with giving precise rules for such constraint manipulation.

Observe now that the constraints imply that ρ_4 and ρ_5 must appear in the *effect* position of the concluding statement, but there is no reason for any of them to appear in either the type nor the environment. Therefore, we are allowed to insert a *new* around the expression, and thus deallocate the closure for $m \llbracket \rho_4, \rho_5 \rrbracket$ as well as the one for $\lambda x. f(x+1)$ after the call returns. By doing so, we make ρ_4 and ρ_5 invisible from outside the expression, so constraints that mention them can be dropped from the constraint set. On the other hand, ρ_3 cannot be finished off in this way yet, because it occurs (explicitly) in the environment Γ .

The final result of the analysis of the expression is thus the judgment

$$\Gamma \vdash \text{new } \rho_4, \rho_5. (m \llbracket \rho_4, \rho_5 \rrbracket ((\lambda x. f(x+1)) \text{ at } \rho_4)) : \wp \text{ int}$$

plus the (simplified) constraint set $\{\rho_3 \in \varphi, \varphi_3 \subseteq \varphi\}$.

1.6.2 EXERCISE [★★]: Why didn't the region inference insert a new $\rho_5 \dots$ around the subexpression $(\lambda x.f(x+1))$ at ρ_4 ? \square

1.6.2 SOLUTION: The constraints collected during the analysis of that subexpression did not entail ρ_5 being in φ at all. It was only when the two sides of the m application were combined that ρ_5 entered the picture. (The point here is that construction of new must necessarily happen *while* each subterm is analyzed; the raw type tree plus constraints does not immediately show where it is useful to insert new except at the root).

The analysis of the rest of the body of m is unsurprising; it ends with

$$m : \dots, f : (\text{int} \rightarrow {}^{\varphi_3}\text{bool}, \rho_3) \vdash \text{if } f(0) \text{ then } 0 \text{ else } (\dots)+1 : {}^{\varphi}\text{int}$$

and still with the same constraint set $\{\rho_3 \in \varphi, \varphi_3 \subseteq \varphi\}$ (another copy of each of these constraints was produced by the analysis of $f(0)$).

This gives the immediate type $((\text{int} \rightarrow {}^{\varphi_3}\text{bool}, \rho_3) \rightarrow {}^{\varphi}\text{int}, \rho')$ for m , where ρ' was added by the `letrec` construction itself. Since neither ρ_3 nor ρ' appears in the (empty) environment for `letrec`, we can abstract over them and make them region parameters. Finally we can abstract over effects by introducing an effect variable for each effect meta-variable in the simplified constraint set that is not connected to the environment by subset constraints. In the effect-polymorphic type, each φ position becomes the set of effect and region variables that must be in the effect, according to the constraints.⁵ Thus the effect abstraction simply encapsulates the (transitive closure of the) simplified constraint set.

It turns out our oracle was right! The abstracted type of m is exactly what it said it would be, with ϵ_1 corresponding to φ_3 , ϵ_2 to φ , ρ_1 to ρ_3 , and ρ_2 to ρ' . Now the analysis of the body of the `letrec` is unsurprising; we get the following region-annotated program:

```
letrec m[ρ₁](f) =
  if f(0) then 0
  else new ρ₄, ρ₅.(m [ρ₄] at ρ₅ ((λx.f(x+1)) at ρ₄)) + 1
in new ρ₆, ρ₇.(m [ρ₆] at ρ₇ ((λx.x=10) at ρ₆))
```

5. The way this is done in the published formulation of the algorithm includes considering one effect variable in each latent effect to special; it is called the *handle* and is the one that corresponds to the entire effect. The distinction between the handle and other effect variables is present in the original TT calculus even though it has no special role in the soundness and correctness proofs.

Where did the oracle get its prediction of m 's type from? Tofte and Birkedal (1998) construct it by *Mycroft iteration*: First, m 's body is analysed under the optimistic assumption that m itself will have the type scheme

$$\forall \epsilon_1, \epsilon_2. \Pi \rho_1, \rho_2. \{\rho_2\} ((\text{int} \rightarrow \{\epsilon_1\} \text{bool}, \rho_1) \rightarrow \{\epsilon_2\} \text{int}, \rho_2)$$

that is, with no constraints at all between the various region and effect parts of polymorphic instances. If the type scheme constructed after the initial iteration does not match (which in this case it doesn't), a new iteration is tried with the new type scheme as assumption, and so forth until a fixpoint is reached.

The trick, of course, lies in ensuring that a fixpoint *is* eventually reached; it might well be that it just produces a list of ever larger type schemes. The original (Tofte and Birkedal, 1998) region inference algorithm solved this problem by heuristically omitting certain opportunities for region and effect abstractions such that the iterative computation of a fixpoint for the recursive function's type scheme could be guaranteed to terminate. The cost of this approach is that completeness fails: Example programs can be constructed for which the region inference algorithm lead to region annotations that are not the best possible.

Later Birkedal and Tofte (2001) rephrased the algorithm in terms of constraint solving. This reworked algorithm seems to be complete in the sense that for any region-annotated term t that can be TT-typed, the inference algorithm's output on $\|t\|$ will have as least as good space behavior as t . However, Birkedal and Tofte prove only a weaker "restricted completeness" result; full completeness in the sense described here was not considered in the article.

Another restricted case with an easy region inference problem is known. It is when the input program can be typed with *first order types*, that is, such that neither the argument nor the return type for any function includes a function type itself. Then there is no reason to use effect polymorphism, and one never needs to generalize over region variables that appear only in latent effects. (For since there is only one arrow in each type, such a variable could just as well have been discharged by a new within the lambda abstraction). These two facts lead to a bounded representation of the latent effect: We simply need to know for each of the p positions in the argument and return types whether the actual p is in the latent effect. That solves the termination problem, and with a bit of ingenuity one does not even need fixpoint iteration for finding the best type scheme.

This principle has been used to derive a region inference for an adaptation of the Tofte–Talpin system for a Prolog dialect which is naturally first-order; see (Makholm, 2000, Chapter 10).

- 1.6.3 EXERCISE [★]: Why does effect polymorphism not make sense in a first-order program? □
- 1.6.3 SOLUTION: Effect polymorphism serves to enforce relations between the effect parts of different arrow constructions in the polymorphic variant of a type. In a first-order program, there is at most one arrow in each type, so there is no need for explicit effect polymorphism.

1.7 More powerful models for region-based memory management

Unfortunately, even with region-polymorphic recursion the Tofte–Talpin model (as expressed either as RTL or the original TT) is not quite strong enough to achieve reasonable object lifetimes. At fault is the very idea of *new*—that the lifetime of a region must coincide with the time it takes to execute some subexpression of the original program. To see how this is a problem, let us look at how the Tofte–Talpin system treats the classic “Game of Life” example. The task is to simulate a cellular automaton for n generations, starting from a specified state. This is a typical case of iterative programming, and the problems we will discover are common for iterative programs in general.

The standard way of programming an iteration in a functional language is to use tail recursion:

```
let rec nextgen(g) = ⟨read g; create and return new generation⟩
let rec life(n,g) = if n=0 then g
                  else life(n-1,nextgen(g))
```

We shall leave the details of `nextgen` unspecified here and in the following discussion. Furthermore we make the simplifying assumption that a single region holds all the pieces of a generation description, and for the sake of the argument we shall assume that the iteration count n needs to be heap-allocated, too.

The ordinary TT region inference algorithm annotates the Game of Life example as follows:

```
letrec nextgen[ρ](g) = ⟨read g from ρ; create new gen. at ρ⟩
letrec life[ρn,ρg](n,g) =
  if n=0 then g
  else new ρ'n
      in life[ρ'n,ρg](n-1) at ρ'n, nextgen[ρg](g)
```

There are two major problems here. First, the recursive call of `life` is not a tail call anymore because it takes some work to deallocate a region at the end

of new. Therefore all the ρ'_n regions will pile up on the call stack during the iteration and be deallocated only when the final result has been found.

Second any typeable region annotation of the program must let the `nextgen` function construct its result in the same region that contains its input. This means that the program has a serious space leak: all the intermediate-generation data will be deallocated only when the result of the iteration is deallocated.

Both of these problems are caused by the fact that `new` aligns the lifetime of its region with the hierarchical expression evaluation. Several solutions for this have been proposed, but because their formal properties have not been as thoroughly explored as the TT calculus, we will only present them briefly.

Region resetting in The ML Kit

The ML Kit's region implementation (Birkedal et al., 1996; Tofte et al., 1998) is based on the TT system. Its solution to the tail recursion problem is based on a concept of *resetting* a region, meaning that its entire contents are deallocated whereas the region itself continues existing.

After a TT region inference, a special *storage-mode analysis* (Birkedal et al., 1996) that runs after region inference amends the region annotations to control resetting: Each “at ρ ” annotation gets replaced by either “atbot ρ ”, meaning first reset the region and then allocate the new object as the new oldest object in the region, or “attop ρ ”, meaning allocate without resetting the region.

With this system one can rewrite the original Life program as follows to obtain better region behavior:

```
let rec copy(g) = ⟨read g; make fresh copy⟩
let rec life'((n,g) as p)
  = if n=0 then p
    else life'(n-1, copy(nextgen(g)))
let rec life(p) = snd (life' (p))
```

where `copy` (whose body is omitted here for brevity) takes apart a generation description and constructs a fresh, identical copy. Region inference and storage-mode analysis will then produce the region annotations

```
letrec nextgen[ $\rho, \rho'$ ](g) = ⟨read g from  $\rho$ ; new gen. at  $\rho'$ ⟩
letrec copy[ $\rho', \rho$ ](g) = ⟨read g from  $\rho'$ ; fresh copy atbot  $\rho$ ⟩
letrec life'[[ $\rho_n, \rho_g$ ]]((n,g) as p)
  = if n=0 then p
    else life'[[ $\rho_n, \rho_g$ ]]((n-1) atbot  $\rho_n$ ,
                          new  $\rho'_g$ )
```



```

in copy[ $\rho'_g$ , atbot  $\rho_g$ ]
  (nextgen[ $\rho_g$ ,  $\rho'_g$ ]( $g$ ))
letrec life[ $\rho_n$ ,  $\rho_g$ ]( $p$ ) = snd (life'[ $\rho_n$ ,  $\rho_g$ ]( $p$ ))

```

Letting `life'` return the entire `p` instead of just `g` forces region inference to place all of the `ns` in the same region ρ_n . A memory leak in ρ_n is prevented by the `atbot` allocation, whose effect is that the region ρ_n is reset prior to placing `n-1` in it.

The memory leak in ρ_g is prevented with the introduction of the `copy` function. Now the new generation can be constructed in a temporary region ρ'_g that gets deallocated before the recursive call; once the old generation is not needed anymore, the new generation is copied into ρ_g with the `atbot` mode which frees the old generation. (The `atbot` annotation in the passing of the region parameter serves to allow `copy` to actually reset the region; the need for this extra annotation has to do with aliasing between region variables.)

The storage-mode analysis works by changing the region annotation for an allocation to `atbot` if the value to be allocated is the only *live* value whose type includes the region name, as determined by a simple local liveness analysis. Neither a formal definition of the storage-mode analysis nor a proof that it is safe has appeared in the literature, but it is described briefly by Birkedal et al. (1996), together with a number of other analyses that the ML Kit uses to implement the region model efficiently.

This solution does make it possible for iterative computations to run in constant space (assuming, in the Life example, that the size of a single `g` is bounded), but it is by no means obvious that precisely these were the changes one needed to make to the original unannotated program to improve the space behavior. Furthermore, inserting such region optimizations in the program impede maintainability because they obscure the intended algorithm.

Aiken–Fähndrich–Levien’s analysis for early deallocation

Aiken et al. (1995) extend the TT system in another direction, decoupling dynamic region allocation and deallocation from the introduction of region variables with the new construct.

In the AFL system, entry into a new block introduces a region variable, but does not allocate a region for it. During evaluation of the body of `new`, a region variable goes through precisely three states: unallocated, allocated, and finally deallocated. After a TT region inference (and possibly also storage-mode analysis as in the ML Kit), a constraint-based analysis—guided by a higher-order data-flow analysis for region variables—is used to insert ex-

PLICIT region allocation $\llbracket \text{alloc } \rho \rrbracket$ and deallocation commands $\llbracket \text{free } \rho \rrbracket$ into the program. Ideally the $\llbracket \text{alloc } \rho \rrbracket$ happens right before the first allocation in the region, and $\llbracket \text{free } \rho \rrbracket$ just after the last read from the region, but sometimes they need to be pushed farther away from the ideal placements, because the same region annotations on a function body must match all call sites.

With this system the Life example can be improved by rewriting the original program to

```
let rec copy(g) = ⟨read g; make fresh copy⟩
let rec life(n,g) = if n=0 then copy(g)
                  else life(n-1,nextgen(g))
```

where the only difference from the original program is that the base case returns a fresh copy of its input rather than the input itself. This program is analyzed as⁶

```
letrec nextgen[ρ, ρ′](g)
  =  $\llbracket \text{alloc } \rho' \rrbracket$  ⟨read g from ρ; new gen. at ρ′⟩  $\llbracket \text{free } \rho \rrbracket$ 
letrec copy[ρ, ρ′](g)
  =  $\llbracket \text{alloc } \rho' \rrbracket$  ⟨read g from ρ; fresh copy at ρ′⟩  $\llbracket \text{free } \rho \rrbracket$ 
letrec life[ρn, ρg, ρ′](n,g)
  = if n=0
    then  $\llbracket \text{free } \rho_n \rrbracket$  copy[ρg, ρ′](g)
    else new ρ′n, ρ′g
       in life[ρ′n, ρ′g, ρ′]
          (  $\llbracket \text{alloc } \rho'_n \rrbracket$  (n-1) at ρ′n  $\llbracket \text{free } \rho_n \rrbracket$ ,
            nextgen[ρg, ρ′g](g) )
```

Because deallocation of each region is done explicitly and not by *new*, the body of *new* is a tail call context, and the regions containing the old *n* and *g* can be freed as soon as *n-1* and *nextgen(g)* have been computed. Without rewriting the original program this would not be the case, because a function must either *always* free one of its input regions or *never* do it.

Imperative regions: the Henglein–Makholm–Niss calculus

Recently Henglein et al. (2001) published a region system that completely severs the connection between region lifetimes and expression structures by

6. The syntax here is not identical with the one used by Aiken et al. (1995); for example, they write “*free_after* ρ *t*” for what we write as “*t* $\llbracket \text{free } \rho \rrbracket$ ”.

eliminating the new construct. Instead, the region annotations form an imperative sublanguage for manipulating region handles asynchronously with respect to the expression structure.

The HMN system does not, as the two previously sketched solutions, build on top of the Tofte–Talpin model and its region inference algorithm; instead it has its own region type system (proved sound by Niss (2002)) and inference algorithm (Makholm, 2003). Starting anew means that the system is conceptually simpler while still incorporating the essential features of ML Kit-like resetting and AFL-style early deallocation as special cases. On the other hand, the theory has not yet been extended to higher-order functions.

In the HMN system it is possible to handle the Game of Life with no rewriting at all. A function can pass regions as output (indicated by o : below) as well as receive them as input (indicated by i :); HMN region inference produces

```

letrec nextgen[i : ρ; o : ρ'](g)
    = [[new ρ'] ⟨read g from ρ; new gen. at ρ'⟩ [[release ρ]]
letrec life[i : ρn, ρg; o : ρ'](n, g)
    = if n=0 then [[release ρn]] g [[ρ' := ρg]]
      else life[i : ρ'n, ρ'g; o : ρ']
          ([[new ρ'n]] (n-1) at ρ'n [[release ρn]],
           nextgen[i : ρg; o : ρ'g](g))

```

where each iteration of `life` decides for itself whether to release the region it gets as its second parameter or to return it back to the caller.

The $[[\rho' := \rho_g]]$ operation serves the same purpose as the `copy` operation in the AFL solution, but is very cheap at runtime - it just renames the region that was previously called ρ_g to ρ' , whereupon it is returned to the caller.

The renaming of regions means that the region-annotated types of values can change during the execution of the program. To manage that, the HMN region type system is based around a typing judgement with the shape

$$\Psi \vdash \{\Delta_1; \Gamma_1\} \tau : \mathbb{T} \{\Delta_2; \Gamma_2\}$$

where the contexts Γ_1 and Γ_2 describe the types of the local variables before and after τ is evaluated. The sets of region variables Ψ , Δ_1 and Δ_2 describe the available regions variables.

Other advanced features of the HMN system include reference-counted regions with a linear type discipline for region handles, “constant” region parameters that correspond to the region abstraction of the Tofte–Talpin model, and a subtyping discipline for regions that allows extensive manipulation of dangling pointers.

Other models

A number of more powerful region models and associated region type systems have been proposed without an accompanying inference algorithm.

Walker et al. (2000) have developed a region model with a region type-system for a continuation-passing style language, intended to be used for translating the Tofte–Talpin execution model to certified machine code. To handle the CPS transformation of region abstractions, a very advanced type system with bounded quantification over regions and effects was necessary. The final system is much stronger than the Tofte–Talpin system itself, but little is known about how to make automatic region inference utilize this extra strength.

Walker and Watkins (2001) have developed a region type system in which region references can be stored in data structures such as lists. They are still not completely first-class, because they must have linear types (see Chapter ??), but the system is strong enough to reason about *heterogenous* lists (i.e., lists whose elements are allocated in different regions).

Another, more restricted, way of allowing heterogenous structures is found in the Cyclone system (Grossman et al., 2002). Cyclone employs a kind of *subtyping* on region lifetimes and a simpler notion of effects: A region variable ρ *outlives* another region variable ρ' if the lifetime of ρ encompasses the lifetime of ρ' . In that event, a value allocated in the region denoted by ρ can safely be used instead of the same value allocated in the region denoted by ρ' . Cyclone supports subtyping of values according to this principle.

1.8 Practical region-based memory management systems

The ML Kit

The ML Kit (<http://www.it-c.dk/research/mlkit>) essentially implements the theory described in §1.4 to §1.6 with two important extensions: firstly it includes region resetting and a storage-mode analysis as already described in §1.7, and secondly it includes a multiplicity inference allowing the compiler to allocate finite regions on the runtime stack (Birkedal et al., 1996). The *multiplicity analysis* is a type-based analysis that determines, for all regions, whether they are finite or infinite. A *finite region* is a region into which the analysis can determine that there will only be written one value ever; all other regions are *infinite*. The importance of finite regions is that they can be stack-allocated since it is known in advance how large they are. Furthermore, since regions in the Tofte and Talpin region language follow a stack-discipline aligned with the expression structure of the program, such

regions can even be allocated on the normal runtime stack giving a particularly simple and efficient implementation. The latest incarnation of the ML Kit even includes a garbage collector (Hallenberg et al., 2002) suitable in situations where it is not practical to make a program more region friendly. See Tofte et al. (2001) for a comprehensive introduction to programming with regions in the ML Kit, and Tofte et al. (2003) for an excellent survey of the interplay between theory and practice in the context of the ML Kit.

Cyclone

Cyclone (<http://www.cs.cornell.edu/projects/cyclone/>) is a dialect of C that is designed to prevent safety violations. It uses regions both as a memory management discipline and as a way to guarantee safety (through a type soundness result). Cyclone includes three kinds of regions: a single *global* (or “heap”) region; *stack* regions (corresponding to stack frames allocated from statement blocks); and *dynamic* regions (corresponding to the lexically scoped regions we have seen in the present chapter).

Instead of having effect variables as in the Tofte and Talpin system, Cyclone uses an operator on types (with no operational significance) called “regions of”. The `regions_of` operator represents the region variables that occur free in a type; the crucial trick is that the `regions_of` operator applied to a type variable is simply left abstract until the type variable is instantiated. Intuitively, instead of propagating the effect of functional arguments via effect variables, they are propagated via the `regions_of` operator. Returning to the `map` example on page 34 we get the following Cyclone type for `map`:

$$\forall \alpha, \beta, \epsilon. (\alpha \rightarrow^{\epsilon} \beta) \times (\alpha \text{ list}, \rho) \rightarrow \{\rho, \rho'\} \cup \text{regions_of}(\alpha \rightarrow^{\epsilon} \beta) (\beta \text{ list}, \rho').$$

For practical reasons a major aspect in the design of Cyclone was to make it easy for C programmers to write Cyclone applications and to port legacy C code to Cyclone. In particular, requiring programmers to write region-annotations as seen in the present chapter is out of the question. Cyclone addresses this by combining inference of region annotations with defaults that work in many cases.

Cyclone began as a compiler for producing *typed assembly language* (see Chapter ??) and as such can be seen as one way to realize *proof-carrying code* (see Chapter ??). The region aspects of Cyclone are described by Grossman et al. (2002); a system overview is given by Jim et al. (2002).

Other systems

The ML Kit and Cyclone are both mature systems. Several research prototypes demonstrating various principles for region-based memory management have been described in the literature.

One trend has been to adapt Tofte and Talpin's system (and its spirit) to other languages. Christiansen and Velschow (1998) describe *RegJava* which is a simple, region-annotated core subset of Java and an accompanying implementation. Makhholm and Sagonas (2002) extend a Prolog compiler with region-based memory management and region inference based on Henglein et al. (2001).

Another trend has been to experiment with the fundamental assumption that regions should be allocated and deallocated according to a stack-discipline. In this direction the present authors have constructed a prototype implementation of the system of Henglein et al. (2001) for a small functional language with function pointers (but not lexical closures containing free variables). Cyclone can also be seen as the practical realization of some of the ideas in the Calculus of Capabilities (Walker et al., 2000).

Finally, region-based memory management without the guarantees of memory safety offered by region type systems have a long history. The basic idea, of gaining extra efficiency by bulk allocations and deallocations, is certainly natural. Systems using region-like abstractions for their memory management date as far back as 1967 (Ross, 1967; Schwartz, 1975; Hanson, 1990). In contrast to these special purpose region abstractions, the GNU C Library provides an abstraction, called *obstacks*, to application programmers (GNU, 2001).

Also in this line of work is Gay and Aiken's *RC* compiler translating region annotated C programs to ordinary C programs with library support for regions (Gay and Aiken, 2001). At runtime, each region is equipped with a reference count keeping track of the number of (external) references to objects in the region. The operation for deleting a region can then flag instances that attempt to delete a region with non-zero reference count. A type system provides the compiler the opportunity to remove some of the reference count operations, but it does not guarantee memory safety as the type systems discussed in this chapter do.

References

- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
- Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Programming Language Design and Implementation*, volume 30(6), pages 174–185, 18–21 June 1995. URL <http://www.cs.berkeley.edu/~aiken/publications/papers/pldi95.ps>.
- Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In POPL POPL (a), pages 104–118.
- Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency*. IC Press, 1999.
- Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258:299–392, 2001. URL <http://www.it-c.dk/people/birkedal/papers/conria.ps.gz>.
- Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Principles of Programming Languages*, pages 171–183. ACM Press, 21–24 January 1996. ISBN 0-89791-769-3. URL <http://www.it-c.dk/people/birkedal/papers/reginm.ps.gz>.
- Cristiano Calcagno. Stratified operational semantics for safety and correctness of region calculus. In POPL POPL (b), pages 155–165. ISBN 1-58113-336-7. URL <ftp://ftp.disi.unige.it/person/CalcagnoC/regions.ps>.
- Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information & Computation*, 173(2):199–221, 2002. URL <http://www.swen.uwaterloo.ca/~shelsen/calcagno-helsen-thiemann-landc-20%01.pdf>.
- Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Principles of Programming Languages*, pages 45–57. ACM Press, 16–18 January 2002. ISBN 1-58113-450-9.

- Morten Voetmann Christiansen and Per Velschow. Region-based memory management in Java. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), 1998. URL <ftp://ftp.diku.dk/diku/semantics/papers/D-395.ps.gz>.
- Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135, Heidelberg, Germany, 25–27 September 1995. Springer-Verlag. ISBN 3-540-60360-3.
- Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Programming Language Design and Implementation*, volume 35(5) of *SIGPLAN Notices*, pages 253–263, New York, NY, USA, 18–21 June 2000. ACM Press. ISBN 1-58113-199-2.
- Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003. ISBN 1-58113-662-5.
- Matthew Fluet. Monadic regions. In *Proc. 2nd Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE), Venice, Italy, January 2004*. URL <http://www.diku.dk/topps/space2004>.
- David Gay and Alexander Aiken. Language support for regions. In *Programming Language Design and Implementation*, volume 36(5) of *SIGPLAN Notices*, pages 70–80, New York, NY, USA, 20–22 June 2001. ACM Press. ISBN 1-58113-414-2. URL <http://www.cs.berkeley.edu/~dgay/papers/pldi01.ps>.
- David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LISP and Functional Programming*, pages 28–38. ACM Press, 4–6 August 1986.
- GNU. GNU C library, version 2.2.5, 2001. URL http://www.gnu.org/manual/glibc-2.2.5/html_mono/libc.html.
- Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW 2001), Cape Breton*, pages 145–159, 2001a.
- Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Electronic Notes in Theoretical Computer Science*, 45:22 pages, 2001b. <http://www.elsevier.nl/locate/entcs/volume45.html>.
- Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW 2002), Cape Breton*, pages 77–91, 2002.
- Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, pages 282–293. ACM Press, 2002.

- Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects*, volume 2053 of *LNCs*, pages 63–83, Heidelberg, Germany, 21–23 May 2001. Springer-Verlag. ISBN 3-540-42068-1.
- Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, June 2002. Berlin, Germany.
- David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, 1990.
- Nevin Heintze. Set based analysis of ML programs. In *Proc. ACM Conf. on LISP and Functional Programming (LFP)*, Orlando, Florida, pages 306–317. ACM, ACM Press, June 1994.
- Nevin Heintze. Control-flow analysis and type systems. In Alan Mycroft, editor, *Proc. Static Analysis Symposium (SAS)*, Glasgow, Scotland, volume 983 of *Lecture Notes in Computer Science (LNCS)*, pages 189–206. Springer-Verlag, 1995.
- Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In Alan Jeffrey, editor, *ACM Workshop on Higher Order Operational Techniques in Semantics*, volume 41(3) of *Electronic Notes in Theoretical Computer Science*, pages 1–20. Elsevier, September 2000. URL <http://www.elsevier.nl/locate/entcs/volume41.html>.
- Simon Helsen and Peter Thiemann. Polymorphic specialization for ML. Available from www.helsen.org, 2003.
- Fritz Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, April 1989. Available as NYU Technical Report 443, May 1989, from New York University, Courant Institute of Mathematical Sciences, Department of Computer Science, 251 Mercer St., New York, N.Y. 10012, USA.
- Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 175–186, Firenze, Italy, September 2001. ACM Press. URL <http://www.diku.dk/~hniss/publications/ppdp2001-abstract.html>.
- Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301, Heidelberg, Germany, 11–13 April 1994. Springer-Verlag. ISBN 3-540-57880-3.
- Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *POPL* POPL (b), pages 128–141. ISBN 1-58113-336-7.
- Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX Association, 2002.

- Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Programming Language Design and Implementation*, volume 24(7), pages 218–226, 21–23 June 1989.
- Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *POPL POPL (a)*, pages 303–310.
- Achim Jung and Allen Stoughton. Studying the fully abstract model of PCF within its continuous function model. In M. Bezem and J.M. Groote, editors, *Proc. Typed Lambda Calculi and Applications (TLCA)*, volume 664 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 1993.
- Ralph Loader. Finitary PCF is not decidable. *Theoretical Computer Science*, 266(1-2): 341–364, September 2001.
- J.M. Lucassen. *Types and Effects towards the Integration of Functional and Imperative Programming*. PhD thesis, Massachusetts Institute of Technology (MIT), August 1987. Technical Report MIT-LCS-TR-408.
- John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages*, pages 47–57. ACM Press, January 1988.
- Henning Makholm. Region-based memory management in Prolog. Master’s thesis, Department of Computer Science, University of Copenhagen (DIKU), March 2000. URL <ftp://ftp.diku.dk/diku/semantics/papers/D-421.ps.gz>. DIKU Technical Report 00/09.
- Henning Makholm. *A language-independent framework for region inference*. PhD thesis, Department of Computer Science, University of Copenhagen (DIKU), 2003.
- Henning Makholm and Kostis Sagonas. On enabling the WAM with region support. In Peter J. Stuckey, editor, *Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 163–178, Heidelberg, Germany, 29 July–1 August 2002. Springer-Verlag. ISBN 3-540-43930-7.
- Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991. Presented at LICS ’89.
- Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, 1997. URL <http://www.diku.dk/research/published/97-1.ps.gz>. Technical Report DIKU-TR-97/1.
- Flemming Nielson and Hanne Riis Nielson. From CML to its process algebra. *Theoretical Computer Science*, 155:179–219, 1996.
- Flemming Nielson, Hanne Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
- Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Principles of Programming Languages*, pages 84–97. ACM Press, 1994. ISBN 0-89791-636-0.

- Henning Niss. *Regions are Imperative: Unscoped Regions and Control-Flow Sensitive Memory Management*. PhD thesis, Department of Computer Science, University of Copenhagen (DIKU), 2002.
- Jens Palsberg. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 20–27. ACM Press, 2001. ISBN 1-58113-413-4.
- Jens Palsberg and Patrick O’Keefe. A type system equivalent to flow analysis. In *Principles of Programming Languages*, pages 367–378. ACM Press, 1995. ISBN 0-89791-692-1.
- Jens Palsberg and Michael Schwartzbach. Type substitution for object-oriented programming. In N. Meyrowitz, editor, *Proc. Conf. Object-Oriented Programming: Systems, Languages, and Applications and European Conf. on Object-Oriented Programming*, pages 151–160, Ottawa, Canada, October 1990. ACM Press.
- Jens Palsberg and Michael Schwartzbach. *Object-oriented Type Systems*. John Wiley & Sons, 1994.
- POPL. *Principles of Programming Languages*, January 1991a. ACM Press.
- POPL. *Principles of Programming Languages*, 17–19 January 2001b. ACM Press. ISBN 1-58113-336-7.
- Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. In *SAS 01: Static Analysis*, LNCS 2126, pages 375–394. Springer-Verlag, 2001.
- Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 166–179, Heidelberg, Germany, July 27–31 2002. Springer-Verlag. ISBN 3-540-43997-8.
- Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL reachability. In *POPL* POPL (b), pages 54–66. ISBN 1-58113-336-7.
- John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461, Edinburgh, Scotland, 1969. North Holland.
- Douglas T. Ross. The AED free storage package. *Communications of the ACM*, 10(8): 481–492, 1967.
- Jacob T. Schwartz. Optimization of very high level languages (parts I and II). *Computer Languages*, 1(2 & 3):161–194, 197–218, 1975.
- Miley Semmelroth and Amr Sabry. Monadic encapsulation in ML. In *4th International Conference on Functional Programming*, volume 34(9) of *SIGPLAN Notices*, pages 8–17, Paris, France, September 1999. URL <http://www.cs.indiana.edu/~sabry/papers/ml-encap.ps>.
- Peter Sestoft. Replacing function parameters by global variables. Technical Report 88-7-2, DIKU, University of Copenhagen, October 1988. SE88.

- Peter Sestoft. Replacing function parameters by global variables. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 39–53. ACM Press, September 1989.
- Olin Shivers. Control flow analysis in Scheme. In *Programming Language Design and Implementation*, volume 23(7), pages 164–174, 22–24 June 1988.
- Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, May 1991.
- Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming (JFP)*, 2(2), 1992.
- Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994.
- Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, 1998. URL http://www.itu.dk/research/mlkit/kit_general/toplas98.ps.gz.
- Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. Region-based memory management in perspective. To appear, 2003.
- Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeldt Olsen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, October 2001.
- Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeldt Olsen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, Department of Computer Science, University of Copenhagen (DIKU), 1998. URL <http://www.it-c.dk/research/mlkit/kit3/manual.ps.gz>.
- Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, January 1994.
- Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.
- Philip Wadler. The marriage of effects and monads. In *3rd International Conference on Functional Programming*, volume 34(1) of *SIGPLAN Notices*, pages 63–74, Baltimore, Maryland, September 1998. URL <http://homepages.inf.ed.ac.uk/wadler/papers/effects/effects.ps.gz>. Journal version submitted to *ACM Transactions on Computational Logic* (2003).
- David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000. URL <http://www.cs.princeton.edu/~dpw/capabilities-toplas.pdf>.
- David Walker and Kevin Watkins. On regions and linear types. In *6th International Conference on Functional Programming*, pages 181–192. ACM Press, 3–5 September 2001. ISBN 1-58113-415-0. URL <http://www.cs.princeton.edu/~dpw/papers/lr.pdf>.

M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.