

+

+

Type and effect systems,  
Region-based memory management:  
Type technology for reasoning about resources

Fritz Henglein

henglein@{it.edu,diku.dk}

The IT University of Copenhagen

(from Aug. 1, 2002: DIKU, Univ. of Copenhagen)

Slides available at <http://www.it.edu/people/henglein>

References: See <http://www.diku.dk/topps/space>

+

1

+

+

## GOALS

- Convey a sense of *type technology*:
  - results (theory), methods (design) and techniques (engineering) loosely related to foundations of internet security (FIS);
  - how to think about types and what they can be used for.
- Present state of the art in region-based memory management (RBMM).
- Show how type technology may help address present software trends.

+

+

+

## RELATION TO INTERNET SECURITY

- General: type-based methods for analyzing and transforming code for predictable and secure execution
- Specific: reasoning about and implementing dynamic memory management without garbage collection.
  - Reduced trusted computing base.
  - Engineering and check of dynamic memory behavior (e.g. for SmartCards).
  - Support for application specific, statically checkable memory management in mobile code.

+

+

+

## LANGUAGE-BASED SECURITY

- LBS: Technologies (theories, methods, tools/mechanisms) based on (programming) language theory and technologies for ensuring/analyzing security sensitive properties of software ('code').
- Single most effective LBS technology in practice: type checkers for strongly typed languages (e.g. Java byte code verifier).
- LBS technologies: proof-carrying code, code certification, typed assembly language, region-based memory management.

+

4

+

+

## **OVERVIEW OF LECTURES**

1. Type and effect systems
2. Region-based memory management

+

+

+

# LECTURE 1

## TYPE AND EFFECT SYSTEMS

+

6

+

+

## TYPES AND TYPE SYSTEMS

**Type:** Syntactic representation of an *interesting property* of a program/program fragment, normally of *value (result)* of executing it.

**Type system:** Axioms and rules of inference specifying allowable compositions of program fragments.

**Classical typing disciplines/type systems:** • Simply typed  $\lambda$ -calculus

- ML-polymorphic  $\lambda$ -calculus
- Polymorphic  $\lambda$ -calculus
- Intersection typing
- Subtyping

+

7

+

+

## TYPE JUDGEMENTS

- Judgement  $A \vdash e : \tau$  means
  - Evaluation of  $e$  will not go “wrong” (get stuck, produce any of certain run-time errors), assuming  $A$  (set of assumptions about free variables of  $e$ ).
  - Evaluation of  $e$ , if it terminates, produces a *value* which satisfies the protocol  $\tau$
- Note:
  - $A$  is a set of assumptions (properties) for free variables, usually at most one per variable; can also be thought of as capabilities required at start of computation for computation to succeed.

+

- Type  $\tau$  can be thought of as a protocol (allowed operations) on a value.
- The typing judgement expresses that the value resulting from the execution supports the specified protocol.
- The typing judgement is a predictor about the value (result of computation); it says nothing about the computation (reduction sequence) itself.

+

+

## SIMPLE TYPING

(VAR)

$$A\{x : \tau\} \vdash x : \tau$$

(ABSTR)

$$\frac{A\{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x : \tau'. e : \tau' \rightarrow \tau}$$

(APPL)

$$\frac{A \vdash e : \tau' \rightarrow \tau \quad A \vdash e' : \tau'}{A \vdash ee' : \tau}$$

+

+

+

## TYPE INFERENCE AND TYPABILITY

- Implicit language: expressions (programs) without types (or coercions/evidence in case of subtyping/qualified type systems); e.g. untyped  $\lambda$ -terms:  $e ::= x \mid \lambda x. e \mid ee'$ .
- Explicit language: expressions with all significant types and coercions (e.g. typed  $\lambda$ -terms); provide unique representation of a particular typing derivation of underlying untyped term; e.g. simply typed  $\lambda$ -terms:  $t ::= x \mid \lambda x : \tau. t \mid tt'$ .
- Type inference: Translate implicit language expression to explicit, well-typed language expression (*completion*), if possible, by inserting types (and coercions/evidence), preserving the underlying implicit expression; e.g.  $\lambda x.x$  can be translated to  $\lambda x : \alpha.x$  or  $\lambda x : \beta \rightarrow \gamma. x$ .

+

+

+

## TYPABILITY

- Typability: Decide whether implicit language expression can be translated to some explicit expression with a type derivation. E.g.,  $\lambda x.x$  is simply typable since  $\{\} \vdash \lambda x : \alpha. x : \alpha \rightarrow \alpha$  can be derived;  $\lambda x.xx$  is not simply typable since it has no simply typed completion; that is, no completion  $t$  such that  $A \vdash t : \tau$  is derivable in simply typed  $\lambda$ -calculus for some  $A, \tau$ .

+

+

+

## LOGICAL RULES AND LANGUAGE RULES

- Language rules: Rules that how to type certain language constructs (e.g.,  $\lambda$ -abstraction, application, conditional expressions, pairs, projections, assignments, etc.). The particular language construct appears in the conclusion of the rule.
- Logical rules: Rules that don't change the subject ( $e$ ) in the implicit version of a type system are *logical rules*; that is, rules about the types themselves, not a particular programming language construct.
- The explicit version of a type system makes application of logical rules explicit in the syntax of the expressions (“explicitly typed terms”).

+

- The implicit version of a type system captures what programs are; that is, what a programmer is expected to write. (We infer properties of programs, which must be given completely. We do not try to infer the programs themselves, as in the field of program extraction from constructive proofs.)
- The difference between the implicit and explicit versions of a type system capture the “logical” parts of reasoning about the properties (types) of an expression (or in type-based analysis the “intensional” properties we are interested in, such as certain implementation details).

+

+

## ML TYPING (EXPLICIT)

ML preterms/explicitly typed terms.

$$\begin{aligned} \tau & ::= \alpha \mid \tau \rightarrow \tau' \quad (\text{monomorphic (simple) types}) \\ \sigma & ::= \tau \mid \forall \alpha. \sigma \quad (\text{polymorphic types (type schemes)}) \\ t & ::= x \mid \lambda x : \tau. t \mid tt' \mid \mathbf{let} \ x = t \ \mathbf{in} \ t' \end{aligned}$$

+

(VAR)

$$A\{x : \sigma\} \vdash x : \sigma$$

(ABSTR)

$$\frac{A\{x : \tau'\} \vdash t : \tau}{A \vdash \lambda x : \tau'. t : \tau \rightarrow \tau}$$

(APPL)

$$\frac{A \vdash t : \tau' \rightarrow \tau \quad A \vdash t' : \tau'}{A \vdash tt' : \tau}$$

(LET)

$$\frac{A \vdash t : \sigma \quad A\{x : \sigma\} \vdash t' : \sigma'}{A \vdash \mathbf{let} \ x : \sigma = t \ \mathbf{in} \ t' : \sigma'}$$

(GEN)

$$\frac{A \vdash t : \sigma}{A \vdash \Lambda \alpha. t : \forall \alpha. \sigma} \quad \alpha \notin FTV(A)$$

(INST)

$$\frac{A \vdash t : \forall \alpha. \sigma}{A \vdash t[\tau] : \sigma[\tau/\alpha]}$$

+

+

## ML TYPING (IMPLICIT)

ML types and (untyped/implicitly typed) expressions

$$e \ :: \ x \mid \lambda x. e \mid ee' \mid \mathbf{let} \ x = e \ \mathbf{in} \ e'$$

Below, rules (VAR), (ABSTR), (APPL) and (LET) are language rules that show how to type all the language constructs we have in this programming language: variable occurrences,  $\lambda$ -abstractions, function application and nonrecursive definitions. Rules (GEN) and (INST) are logical rules.

+

(VAR)

$$A\{x : \sigma\} \vdash x : \sigma$$

(ABSTR)

$$\frac{A\{x : \tau\} \vdash e : \tau'}{A \vdash \lambda x. e : \tau \rightarrow \tau'}$$

(APPL)

$$\frac{A \vdash e : \tau \rightarrow \tau' \quad A \vdash e' : \tau}{A \vdash ee' : \tau'}$$

(LET)

$$\frac{A \vdash e : \sigma \quad A\{x : \sigma\} \vdash e' : \sigma'}{A \vdash \mathbf{let} \ x = e \ \mathbf{in} \ e' : \sigma'}$$

(GEN)

$$\frac{A \vdash e : \sigma}{A \vdash e : \forall \alpha. \sigma} \quad \alpha \notin FTV(A)$$

(INST)

$$\frac{A \vdash e : \forall \alpha. \sigma}{A \vdash e : \sigma[\tau/\alpha]}$$

+

+

## FUNDAMENTAL CHARACTERISTICS: SUBJECT REDUCTION

- Subject reduction: Preservation of types under reduction (execution, simplification): if  $A \vdash e : \tau$  (is derivable) and  $e$  *reduces* (*simplifies/translates*) to  $e'$  then  $A \vdash e' : \tau$  (is derivable).
- Holds for all classical type systems and the “standard” reduction relation of the programming language. Can be considered characteristic of type system (“if it has no subject reduction property, it is not a type system”).
- Significance: Soundness. General idea:
  - Let  $\longrightarrow$  be relation that captures execution; that is, if executing  $e$  results in value  $v$  then there is a reduction sequence  $e = e_0 \longrightarrow e_1 \longrightarrow \dots \longrightarrow e_n = v$  and vice versa.

+

- Write  $\models e : \tau$  whenever it happens to be the case that the value computed by executing  $e$  has type  $\tau$ . Recall that  $\vdash e : \tau$  is intended to capture this correctly, though not completely. That is, if  $\vdash e : \tau$  then  $\models e : \tau$  (*soundness*), but not necessarily conversely (*completeness*): there may be  $e$  with  $\vdash e : \tau$  but where we cannot derive  $\vdash e : \tau$ .
- The subject reduction property guarantees soundness in a “step-by-step” fashion. If  $\vdash e_0 : \tau$ , the subject reduction property implies inductively that  $\vdash e_n : \tau$ . So indeed, whenever we derive  $\vdash e : \tau$  we can conclude that the value resulting from execution of  $e$  indeed has the type  $\tau$ .

+

+

## FUNDAMENTAL CHARACTERISTICS: SUBJECT EXPANSION

- Subject expansion: Preservation of types under *inverse* of reduction (“expansion”): if  $A \vdash e' : \tau$  and  $e$  reduces (simplifies/translates) to  $e'$  then  $A \vdash e : \tau$ .
- Holds for restricted notions of reduction only, dependent on type systems. May furthermore only hold for restricted class of expressions and types (that is only for certain  $e, \tau$ , not all  $e, \tau$  in the above definition).
- Significance: Qualitative measure of expressiveness (“semantic precision”) of a type system.

+

- If a type system has the type invariance property (subject reduction + subject expansion) for some reduction relation then it captures the (equational theory induced by that) reduction relation precisely: whenever  $e = e'$  (that is,  $e$  can be transformed to  $e'$  through some series of reduction and expansion steps) then  $e$  and  $e'$  have the same set of types.
- Examples of reduction relations with a type invariance property:
  - Simple typing: linear and quasi-linear  $\beta$ -reduction;
  - ML-typing: unfolding of nonrecursive definitions (let-expressions);
  - ML-typing with polymorphic recursion: unfolding of recursive definitions (letrec-expressions);
  - intersection typing:  $\beta$ -reduction.

- Type invariance: Basis of powerful technique for showing complexity-theoretic lower bounds for type inference and typability.
- Intuition: The more precision a type system has (measured by how “big” a reduction relation with the type invariance property it has), the harder it is to infer types and, in particular, decide whether an expression has a type at all.

+

+

## FUNDAMENTAL CHARACTERISTICS: PRINCIPAL TYPING

- Principal typing: Ability of a type system to describe all types of an expression by a single type.
  - Version 1 (“Principal type” property, internalized principal typing): if  $A \vdash e : \tau$  for some type  $\tau$  then there is a type  $\sigma$  that describes *all* types of  $e$  under  $A$ ; that is,  $A \vdash e : \sigma$  and for each  $\tau'$  such that  $A \vdash e : \tau'$  we have  $\vdash_L \sigma \implies \tau'$ , where  $\vdash_L \sigma \implies \tau'$  means that  $\tau'$  is implied by  $\sigma$  using the logical rules of the type system only.
  - Version 2 (“Principal typing judgement” property, externalized principal typing): if  $A \vdash e : \tau$  for some type  $\tau$  then there is a typing judgement  $B \vdash e : \sigma$  such that  $(B, \sigma) \implies (A', \tau')$  whenever  $A' \vdash e : \tau'$ , where  $\implies$  is a binary relation on assumption/type pairs (not necessarily part of inference system).

+

- Significance (of internalized principal typing): Key to modular analysis and context insensitivity of type system (different occurrences of a function  $f$  get treated independently of each other).
- Consider library function  $f = e$ . We don't know yet (or ever) all the uses of  $f$ ; that is, where it is going to be applied. Which type should we give  $f$ ? The principal type of  $f$  is best: it works for all uses of  $f$  (occurrences of the name  $f$  instead of writing its code) in the future and works just as well as if we had replaced each occurrence of  $f$  with a separate copy of  $e$ .

+

+

## LOWER BOUND RESULTS

**Lemma 1** (*Type Invariance Lemma, informally stated*) *There is a log-space computable coding of Turing Machines and configurations as  $\lambda$ -expressions such that:*

- *$\beta$ -reduction of  $\text{next}(\text{state})$  to normal form  $\text{state}'$  simulates execution of a (single) Turing Machine step;*
- *$\vdash \text{next}(\text{state}) : \tau$  if and only if  $\vdash \text{state}' : \tau$  for all simple types  $\tau$ .*

**Corollary 2** • *Simple typability is P-hard, and so is any nontrivial TS that includes simple typing; e.g. any reasonable extension with subtyping.*

+

- *ML-typability is EXPTIME-hard, and so is any nontrivial TS that includes it (e.g., polymorphic  $\lambda$ -calculus).*

+

+

## TYPE-BASED PROGRAM ANALYSIS

- Idea: Given a (possibly typed) language, design an application-specific type system for deriving interesting internal properties.
- Examples:
  - Binding-time analysis;
  - Dynamic (soft) typing;
  - Boxing analysis;
  - Strictness analysis;
  - Region-based memory management;
  - Pointer aliasing;

+

- Communication topology analysis;
- Y2K types
- Tainted/untaintedness analysis;
- ...

+

+

## EXAMPLE: BOXING ANALYSIS

$$\begin{aligned} \rho &::= v \mid \pi \quad (\text{representation types}) \\ v &::= \alpha \mid \rho \rightarrow \rho \quad (\text{unboxed types}) \\ \pi &::= \beta \mid v@ \quad (\text{boxed types}) \\ \sigma &::= \rho \mid \forall\beta.\sigma \quad (\text{polymorphic types}) \end{aligned}$$

Add following to ML-typing rules:

$$\text{(BOX)} \quad \frac{A \vdash e : v}{A \vdash e@ : v@}$$

$$\text{(UNBOX)} \quad \frac{A \vdash e : v@}{A \vdash e! : v}$$

and restrict (GEN) and (INST) to generalizing/instantiating with boxed types only.

+

+

+

## BOXING ANALYSIS (EXAMPLE)

Consider  $e \equiv (\mathbf{let} \ x = (2, 3) \ \mathbf{in} \ \lambda y. (fst \ x, y) \ \mathbf{end} )5$ .

A possible boxing completion of  $e$  is

$(\mathbf{let} \ x : (int@ \times int@)@ = (2@, 3@)@ \ \mathbf{in} \ \lambda y : int@. (fst(x!), y)@ \ \mathbf{end} )(5@)$ .

This is the *canonical* boxing completion: keep everything boxed until needed unboxed. Other boxing completions are possible:

- Leroy-completion: keep everything unboxed until needed boxed
- Formally optimal completions: keep data unboxed if used mostly unboxed, boxed if used mostly boxed. Based on proof theory for all possible boxing completions for a program.

+

+

+

## EFFECTS

- In classical type systems types model properties of the results of computations, but not of the computations themselves (“side effects”).
- Idea: Extend classical (value) types with descriptions of computational effects (e.g., “a region is read”, “a value is sent on a channel”, “a reference is updated”).

+

+

+

## TYPE AND EFFECT JUDGEMENTS

- Judgement  $A \vdash e : \tau \ \& \ b$  means
  - Evaluation of  $e$  will not go “wrong” (get stuck, produce any of certain run-time errors), assuming  $A$ .
  - Evaluation of  $e$ , if it terminates, produces a *value* which satisfies the protocol  $\tau$ .
  - Evaluation of  $e$  produces effects contained in effect description  $b$ .

+

+

+

## BEHAVIOR EFFECTS

$a ::= \langle \text{atomic effects} \rangle$

$b ::= a \mid \emptyset \mid b; b' \mid b + b' \mid \dots$

+

$$\begin{array}{l}
\text{(VAR)} \quad A\{x : \sigma\} \vdash x : \sigma \ \& \ \epsilon \\
\text{(ABSTR)} \quad \frac{A\{x : \tau\} \vdash e : \tau' \ \& \ b}{A \vdash \lambda x. e : \tau \rightarrow^b \tau' \ \& \ \epsilon} \\
\text{(APPL)} \quad \frac{A \vdash e : \tau \rightarrow^{b''} \tau' \ \& \ b \quad A \vdash e' : \tau \ \& \ b'}{A \vdash ee' : \tau' \ \& \ b; b'; b''} \\
\text{(LET)} \quad \frac{A \vdash e : \sigma \ \& \ b \quad A\{x : \sigma\} \vdash e' : \sigma' \ \& \ b'}{A \vdash \mathbf{let} \ x = e \ \mathbf{in} \ e' : \sigma' \ \& \ b; b'} \\
\text{(IF)} \quad \frac{A \vdash e : \mathit{bool} \ \& \ b \quad A \vdash e' : \tau \ \& \ b' \quad A \vdash e'' : \tau \ \& \ b''}{A \vdash \mathbf{if} \ e \ \mathbf{then} \ e' \ \mathbf{else} \ e'' : \tau \ \& \ b; (b' + b'')}
\end{array}$$

If the sequential order of effects and their multiplicity (how often they occur) is not important, both ; and + can be interpreted as set union; that is, effects become sets of atomic effects.

+

+

## **LECTURE 2**

Region-based memory management

+

+

+

## BOXING ANALYSIS REVISITED

Consider boxing analysis again:

$$e \equiv (\mathbf{let } x = (2, 3) \mathbf{ in } \lambda y. (fst\ x, y) \mathbf{ end } )5.$$

.

Its canonical boxing completion is

$$\begin{aligned} &(\mathbf{let } x : (int@ \times int@)@ = (2@, 3@)@ \\ &\mathbf{ in } (\lambda y : int@. (fst\ (x!), y)@)@ \mathbf{ end } )(5@). \end{aligned}$$

Idea: Make explicit into which region of memory values are written and read from.

+

+

+

## HEAP-BASED MEMORY MANAGEMENT

Idea: Make the part of memory into which values are written and from which they are read explicit.

Single region (heap): Let  $H$  be bound to the handle for the current heap (base register plus first available free memory slot in the heap).

$$(\mathbf{let} \ x : (int@H \times int@H)@H = (2@H, 3@H)@H \\ \mathbf{in} \ (\lambda y : int@H.(fst(x!H), y)@H)@H \ \mathbf{end} )!H(5@H).$$

Note: No data stored in the heap is deallocated explicitly. We do not even have an operation for that! Presumably a *garbage collector* takes care of this behind our backs: Concurrently (but asynchronously) with evaluation it figures out which data in the heap are no longer accessible from the current program state and deallocates them; that is, it recycles their storage area.

+

+

+

## REGION-BASED MEMORY MANAGEMENT

Idea: Put values into multiple regions instead of a single heap.

$$\begin{aligned}
 &(\mathbf{let} \ x : (int@R_1 \times int@R_2)@R_3 = (2@R_1, 3@R_2)@R_3 \\
 &\mathbf{in} \ (\lambda y : int@R_4.(fst(x!R_3), y)@R_5)@R_6 \ \mathbf{end} \ )!R_6(5@R_4).
 \end{aligned}$$

Note: Box (@) and unbox (!) operations for same data must refer to same region.

Where do the regions and their region handles  $R_1, R_2, \dots, R_6$  come from? All 6 regions must exist before evaluation starts and exist until whole evaluation completes. Not much point if all we accomplish is that 6 handles are required instead of 1!

+

+

+

## **MINIZING REGION LIFETIMES**

Idea: Create regions and their region handles 'just in time' before they are used first, and delete regions and their region handles immediately after they have been used last.

+

+

+

## REGION-BASED PROGRAMMING

Idea: Allocate data with similar lifetimes (and temporal locality of reference) in same memory chunk (region).

Benefits:

- Fast deallocation: 1 dealloc = n deletes
- Good data cache behavior
- Real-time management of dynamic (heap) memory
- Predictable, syntactically explicit program-synchronous memory management

+

- Static and profile-based engineering of memory performance
- Semantic garbage deallocation (safe dangling pointers)

+

+

## **REGION SAFETY, LAG AND DRAG**

Region safety: No data read/write from un- or deallocated region.

Lag: Time between allocation of data and first use (read or write)

Drag: Time between last use and deallocation of data

Goal: Static guarantee of region safety while minimizing lag and (especially) drag.

Approach: Type and effect system that guarantees region safety and correctness (preservation of semantics); clever type inference for producing 'good' derivations with little lag and drag.

+

+

+

## LEXICALLY SCOPED REGION-BASED MEMORY MANAGEMENT

Idea: Write **region**  $R\{e\}$  for an expression that first allocates a new region of memory, binds  $R$  to a handle to it, evaluates  $e$  and finally unbinds  $R$  and deallocates the region it points to. Use this to recycle memory. Don't use garbage collection.

```
region  $R_2, R_3, R_6\{$ 
  let  $x : (int@R_1 \times int@R_2)@R_3 = (2@R_1, 3@R_2)@R_3$ 
  in  $(\lambda y : int@R_4.(fst(x!R_3), y)@R_5)@R_6$  end  $)!R_6(5@R_4)$ 
 $\}$ 
```

Note:  $R_1, R_4, R_5$  are 'global' (free). They must exist before computation starts. They also exist once computation ends (with more data stored in them).  $R_2, R_3, R_6$ , on the other hand, are deallocated.

+

+

+

## DANGLING POINTERS 1: A DANGEROUS CASE

```

region  $R_2, R_3$  {
  region  $R_6$  {
    let  $x : (int@R_1 \times int@R_2)@R_3 =$ 
       $(2@R_1, 3@R_2)@R_3$ 
    in  $(\lambda y : int@R_4. (fst(x!R_3), y)@R_5)@R_6$  end )
  }! $R_6$ 
   $(5@R_4)$ 
}

```

Region  $R_6$  is deallocated before function is applied to 5. But executing the body of the function requires access to  $R_6$  for storing the pair of pointers making up the result: Memory failure!

Note:  $R_6$  occurs in the type of the function:  $int@R_4 \rightarrow (int@R_1 \times int@R_4)@R_6$ .

+

+

+

## DANGLING POINTERS 2: A HARMLESS CASE

```

region  $R_3, R_6$  {
  let  $x : (int@R_1 \times int@R_2)@R_3 = (2@R_1, 3@R_2)@R_3$ 
  in  $(\lambda y : int@R_4.(fst(x!R_3), y)@R_5)@R_6$  end )!R_6(5@R_4)
}

```

Region  $R_2$  is deallocated before function is applied to 5. The function thus contains a “dangling” pointer (the second component of the value bound to  $x$ ). In this case that’s okay since the function never dereferences that pointer.

Note:  $R_2$  does not occur in the type of the function:  $int@R_4 \rightarrow (int@R_1 \times int@R_4)@R_6$ .

Idea: Can deallocate a region if it does not occur in the type of an expression. (Idea is good, though technically wrong. We develop it first and then fix it up.)

+

+

+

## REGION INFERENCE: BASIC RECIPE

- Spreading: Make simple-minded canonical boxing completion with “fresh” region variable in each boxing/unboxing operation.
- Unification: Infer equations between region variables by “simple” data flow and unify them; e.g.

**let**  $x = e@R$  **in** ...  $x!R'$  ...

requires  $R = R'$ .

- Look at resulting type and typing assumption with explicit region variables to figure out which expressions can be wrapped with **region**  $R\{\dots\}$

+

- If  $R$  does not occur in  $\{x_1 : \tau_1 \textcircled{R}_1, \dots, x_n : \tau_n \textcircled{R}_n\} \vdash e : \tau_0 \textcircled{R}_0$  but it occurs in  $e$  then:
  - $R$  does not need to exist when evaluation of  $e$  starts (it does not occur in the assumptions);
  - $R$  does not need to exist when the evaluation of  $e$  ends (it does not occur in the value of  $e$  and thus cannot be *observed* (required) by any continuation that uses it.

In this case, wrap **region**  $R\{\dots\}$  around  $e$ .

+

+

## **REGION INFERENCE: SOUNDNESS PROBLEM**

Not all region variables required by a continuation (computation that uses a value) show up in the type of the expression that produces that value.

Problem: Lexical closures! Type of a function does not mention any of the types of its free variables, consequently neither of the regions in those types. But access to values of those variables and thus to their regions may be required when function is called!

+

+

+

## DANGLING POINTERS 3: A SINISTER CASE

```

region  $R_2, R_6$  {
  region  $R_3$  {
    let  $x : (int@R_1 \times int@R_2)@R_3 = (2@R_1, 3@R_2)@R_3$ 
    in  $\lambda y : int@R_4.(fst(x!R_3), y)@R_5)@R_6$  end
  }
   $!R_6(5@R_4)$ 
}

```

Note:  $R_3$  does not occur in the type of the function:  $int@R_4 \rightarrow (int@R_1 \times int@R_4)@R_6$ . Nonetheless, it is required every time the body of the function is executed; that is, every time the function is applied. Because  $R_3$  is deallocated before the function is called, the function application has a memory failure (it dereferences a dangling pointer).

+

+

+

## EFFECTS TO THE RESCUE

Basic idea: Don't just write down type of value produced by body of function, but also all regions accessed during evaluation (effect).

Bad idea: Include types of all free variables in  $\lambda$ -abstraction. Functions with same functionality (same standard type) obtain suddenly very different types! E.g., two functions with same standard ("normal") type  $int \rightarrow int$  may have completely different region types if we include the types of their free variables.

Better: Include regions (not whole types) of free variables of formal parameter and free variables required for evaluation of function body.

Capture as explicit *atomic effects* on regions:  $get(R)$  and  $put(R)$ .

+

+

+

## DANGLING POINTERS 4: SINISTER CASE DEBUNKED

Consider function definition in our example, with explicit type and effect.

We have

$$\{x : (int@R_1 \times int@R_2)@R_3, y : int@R_4\} \vdash \\ (fst(x!R_3), y)@R_5)@R_6 : (int@R_1 \times int@R_4)@R_6 \ \& \ \{get(R_3), put(R_6)\}$$

and thus

$$\{x : (int@R_1 \times int@R_2)@R_3\} \vdash \\ \lambda y : int@R_4. (fst(x!R_3), y)@R_5)@R_6 : \\ int@R_4 \rightarrow \{get(R_3), put(R_6)\} (int@R_1 \times int@R_4)@R_6 \ \& \ \emptyset$$

The effect on the function type arrow is a *delayed* effect. It occurs when the function is *called*, not when it is *defined*.

Note:  $R_3$  occurs in the delayed effect. So we do not wrap **region** $R_3\{\dots\}$  around the function and we are safe.

+

+

+

## TOFTE/TALPIN REGION INFERENCE SYSTEM (CORE ONLY)

$a ::= \{put(\rho)\} \mid \{get(\rho)\}$  (atomic effects)

$b ::= \epsilon \mid \emptyset \mid a \mid b \cup b'$  (effects)

$\mu ::= \tau @ \rho$  (type-and-places)

+

(VAR)

$$A\{x : \mu\} \vdash x : \mu \ \& \ \emptyset$$

(ABSTR)

$$\frac{A\{x : \mu\} \vdash e : \mu' \ \& \ b}{A \vdash (\lambda x : \mu. e) \textcircled{R} : (\mu \rightarrow^b \mu') \textcircled{R} \ \& \ \{\text{put}(R)\}}$$

(APPL)

$$\frac{A \vdash e : (\mu' \rightarrow^{b''} \mu) \textcircled{R} \ \& \ b \quad A \vdash e' : \mu' \ \& \ b'}{A \vdash ee' : \mu \ \& \ b \cup b' \cup \{\text{get}(R)\} \cup b''}$$

(LET)

$$\frac{A \vdash e : \mu \ \& \ b \quad A\{x : \mu\} \vdash e' : \mu' \ \& \ b'}{A \vdash \mathbf{let} \ x = e \ \mathbf{in} \ e' : \mu' \ \& \ b \cup b'}$$

(LETREGION)

$$\frac{A \vdash e : \mu \ \& \ b \quad (\rho \notin FRV(A, \mu))}{A \vdash \mathbf{region} \ \rho\{e\} : \mu \ \& \ b - \{\text{get}(\rho), \text{put}(\rho)\}}$$

+

+

## EXPRESSIVENESS PROBLEMS

- Functions with equal standard type may (still) have different region types due to differences in effects — must ensure that they can be unified: Add effect subtyping (done in limited form in TT due to desire for reducing region inference to unification of equational constraints).
- Arguments to different calls of same function must be in same region — data used in first call can only be allocated after last call to same function: Add region and effect polymorphism (abstraction over region variables and effect variables).
- Data in all recursive calls end in same region, first deallocated after first call returns: Add polymorphic recursion (for region and effect polymorphism).

+

Resulting system quite complex:

- Arrow effects, effects, regions
- Very complex correctness proof (simpler region safety proofs exist now, but region safety is only half of the story; correctness requires that the explicit version with explicit regions computes the same results as the given program without any regions)
- Complex region inference (no principal typing)
- Requires good understanding of region system and programming tricks

+

+

## REGION-BASED SYSTEMS

- ML Kit with Regions (Hallenberg, Elsmann, Tofte et al., for SML)
- Cyclone (D Grossman, Morrisett et al., for C programs)
- RC (Gay, Aiken)
- Vault (Fähndrich/DeLine)
- SMLServer (Hallenberg, Elsmann)
- RProlog (Makholm)

+

- RegJava (Christiansen, Henglein, Niss, Velschow)
- RegFun (Henglein, Makholm, Niss)

+

+

## **PERSPECTIVES**

- Software trends
- Type technology

+

+

+

## SOFTWARE TRENDS

**Multi-language interaction:** requires explicit interface specifications

**Late binding:** ● of data to names: higher-order (incl. oo) programming, dynamic dispatch, imperative update of shared resources;

● of software to names: dynamic linking, mobile code, run-time code generation;

● of interfaces to names: interface negotiation using name services;

● of implementations to interfaces;

● of data formats to wire protocols: e.g., document brokers

+

So less and less is statically known (easily predictable)! Easier to generate flexible systems, but more difficult to understand what they do.

+

+

## TYPE TECHNOLOGY COUNTERMEASURES

How may type technology help mitigate the negative aspects of current software trends?

- Explicit interfaces (types)
- Name abstraction built in: higher-order programming support for free
- Modularity through polymorphism and principal typing (component architecture support)
- Good possibilities for integration with staging (do some things statically, some dynamically)
- Rather well-understood efficient inference technology

+

+

+

## TYPE TECHNOLOGY CHALLENGES

Current challenges/developments:

- Treatment of concurrency (behavior types; Kobayashi; Rajamani/Rehof, CAV 2001, 2002; Gordon, CSFW 2001, 2002)
- Aliased update and control-flow sensitive reasoning (e.g., TAL papers; Foster et al., PLDI 2002; Fähndrich/DeLine, PLDI 2002)
- Integration with rich domain theories such as arithmetic (e.g. proof-carrying code, Necula/Lee; Appel/Felty, Shao et al.)
- Resource management that is not syntactically (lexically) scoped (type and effect systems; resource-sensitive logics; capabilities, e.g. Walker/Crary/Morrisett, Henglein/Makholm/Niss)

+

- Local reasoning about (side) effects (spatial logics, O'Hearn et al.; Reynolds)