Types for State and Aliasing
(focus: inference, types, aliases, modern CPUs)

A four-lecture introduction

Alan Mycroft
Computer Laboratory, University of Cambridge
http://www.cl.cam.ac.uk/users/am/

EWSCS’2010: 1–5 March 2010
BEWARE!

These are draft notes – containing rather more material than can be presented in the time available.

I’m happy to determine the exact emphasis following audience demand.
Abstract

Dijkstra’s famous maxim “go to statement considered harmful” noted that spaghetti-like code was hard to reason about. However, in the presence of side effects and (especially) concurrency, spaghetti-like data is far harder to understand – not only in terms of the full behaviour of a program but also in terms of lesser properties such as performance prediction on multicore architectures. We present various compile-time techniques for controlling stateful operations and investigate some of their connections. These techniques include sub-structural type systems, effect systems, typestate, session types, contracts, linear and quasi-linear types, monads, separation logic and ownership types.
Structure


- Lecture 4: Quasi-linear types. Our solutions: PacLang, Kilim
Lecture 1: Introduction

- Types versus program analysis.
- The wonderful world of multicore hardware.
Viewpoint of these lectures

These lectures present a *programming language* viewpoint of types, state, aliasing and the like. This is informed by (but different from):

- a type- and semantics-centred viewpoint (more theorems, emphasis on mathematical rather than programming language ‘style’)

- a systems- or software-engineering viewpoint (more details of hardware, more “bug counts per kLoC [thousand lines of code]”)

These lectures attempt to capture *core phenomena and techniques* from a wide range of work and therefore often merely summarise a vast area in one or two slides.

- Theory is therefore a tool rather than an aim in itself!

- I’ll focus on a breadth (ideas) rather than depth (equations).
Low-level languages (assembler, C etc.) are problematic for two reasons

1. unsafe

2. little structuring (C is essentially minimal: global variables, call stack with local variables, heap structs/unions; no support for concurrency)

C++ adds a little to 2. but without helping 1.
Programming language design (2)

More modern programming languages (ML, Haskell, Java, C#) provide constructs and a type system so that programs are safe [during execution all operations are given values of the expected ‘form’ – no memory protection violations etc.].

The constructs are generally richer:

- module systems (interfaces); `private` methods/members
- first class functions with lexical closures in ML/Haskell;
- support for concurrency in Java/C#;
- lazy evaluation in Haskell.

Feature-rich languages enable us to build more complicated systems.

- So what’s wrong?
Programming language design (3)

Question: “what’s wrong with feature-full languages?”
Answer: “feature interaction”.

- Feature interaction is a Software Engineering issue in general.

- First examples from telephone system. E.g. what happens when the user enables both “forward calls when busy” and “call waiting”; a first call is answered and before it has finished a second call arrives.

- While the system defines which takes priority, it’s not obvious to the user – and the more features the more complicated their interaction.
Feature interaction in programming languages

Here are four useful features of programming languages

- lazy evaluation (don’t calculate unnecessary values)
- mutable variables (hardware provides this, so why not use it?)
- aliasing (multiple pointers to the same heap cell)
- concurrency (perhaps parallelism on multi-core processors)

Just because things are useful doesn’t mean they work well together (feature interaction).

Consider having both lazy evaluation and mutable variables:

- perfectly well-defined, we can give a good semantics; but
- humans can’t understand it (and it might be hard to get simple proof techniques for formal reasoning) – no ‘local reasoning’.
Feature interaction in programming languages

Tempting to conjecture:

- Modern languages don’t have any bad feature interaction.

This is false, and the rest of the course will introduce techniques to combat feature-interaction-style problems. (Just like strong types give safety.)

Basically, the issue is controlling the excessive power of some of features and their interaction. Hoare wrote in 1974: “The language should enable a compiler and its run-time system to detect as many cases as possible in which the language concepts break down and produce meaningless results”.

- Richer languages have more ways for concepts to break down!
Feature interaction in programming languages (2)

We’ll first look how monads unified mutability and laziness using this as a metaphor for other changes.
Monads encode State in lazy functional languages

Two examples (the same) of why laziness and side-effects do not mix:

- \((\lambda(x, y). \ldots x \ldots y \ldots)(\text{getchar()}, \text{getchar}())\)
- \((\lambda(x, y). \ldots x \ldots y \ldots)((m := m + 1; e_1), (m := m \times 2; e_2))\)

We’ll focus mainly on the 2nd. Haskell users well know about the \textbf{IO()} monad for the first already; we’ll take a first-principles approach.

The problem is that of state – here the value of global variable \(m\). We should really represent this explicitly and thread it through the computation.

...
But this is clumsy and changing the structure state means changing many lines of program. A mathematical structure *monads* help here.

A monad $M$ is a type-constructor which represents a computation returning a value: $M \text{Int}$ are computations (think side-effecting) which have an integer value.

For our purposes it’s convenient to exemplify $M$ as $\text{ST } t$, the monad of state $t$, so that $\text{ST } t \ t'$ is the set of computations returning $t'$ whose operations have side-effects on a store of type $t$. 
Monads (3)

To avoid side-effects being unpredictable (avoiding feature interaction) we ensure:

- \( \text{M a} \) is an abstract data type (given a value of type \( \text{M a} \) there is no direct access to values of type \( a \)); the only operations on \( \text{M a} \) are:

  - \( \text{return: } \ a \rightarrow \text{M a} \); and

  - \( \text{bind: } \ (\text{M a}) \rightarrow (a \rightarrow \text{M b}) \rightarrow (\text{M b}) \).

Note the type of \( \text{bind} \): it unwraps an \( a \) from an \( \text{M a} \), and provides it to a function which can only re-wrap it (or a computation involving it) back into another monad (here \( \text{M b} \)).

We can add additional operations to a monad so long as they do not disturb the encapsulation (e.g. of mutable state).
Thus the side-effecting *aspects* of a system are serialised in a way required by the programmer, not incidental to the system. Say “think of values of type $M \ a$ as actions resulting in type $a$”

We won’t go into great detail here, but note the State monad $(ST \ s) \ a$, and the IO monad $IO()$.

- `getChar :: IO Char`
- `putChar :: Char -> IO ()`
- `putStr :: String -> IO ()`
We’ve seen how mutability interacts badly with laziness.

But the feature interactions of mutability with both aliasing and threading turn out to be similarly problematic for modern programming languages too – but with little language/compiler support.

The rest of the course will:

- explore these issues; and
- examine some type-like solutions from the literature.
Note that `private` is not in general sufficient to control aliasing.

class A { public int m; }
class B { private A p;
    public B(C c) { p = c.cheat(); }
    public static void f() { p.m++; }
}
class C { private A p;
    public A cheat() { return p; }
    public int f() { int x=p.m; B.f(); return x-p.m; }
    // f() can return non-zero even though p is private
}

Later we’ll see how ownership types can address this.
Types versus program analysis

An ongoing controversy/dichotomy:

- A type system determines (statically) which programs are acceptable and which are not. Non-acceptable programs have no meaning.

- An alternative is to allow all programs, and then use a (static) program analysis to determine ‘nice’ programs. Nice programs can then be optimised. Non-nice programs must still run (maybe compile-time warning).

Obvious examples are Java (statically typed), and an optimising compiler for a dynamically-typed language like Python.
Types versus program analysis (2)

Often both views are possible (so look for examples of type systems in program analyses and program analysis in type systems!).

We’ll look at richer (than usual C/Java/ML) type systems, e.g. \( t \xrightarrow{S} t' \) meaning function from \( t \) to \( t' \) with side effects given by \( S \).

Such user-specified enriched types:

- provide additional compile-time invariants (helpful/useful to specify in interfaces).
- can be verbose for expressive type systems

Often types do not need to be completely specified; e.g. ML type inference (a.k.a. type reconstruction) which is now part of C# (‘var’ declarations).
Types versus program analysis (3)

Program analysis:

- For a large program, optimisations based on a whole program analysis (whether type-based or abstract-interpretation-based) are generally unwise.

- The discontinuity effect: big program program $P$ might have property $Q$ but a small change $P + \Delta P$ might no longer have $Q$. What if $Q$ is needed to parallelise $P$?

Local inference/analysis is good (and generally more continuous), but needs **user-specified anchor points** – interface specifications.

Need language designs which enable the user to express useful properties at interfaces and declarations.
Relation to undecidability. Assume the programming language is Turing powerful, then

- type systems: a decidable type system \textit{must} reject at least one program which would actually run OK without type checking. Consider \texttt{if <tautology> then 42 else true}.

- program analysis: Rice’s theorem says that no decidable program analysis can be exact. Consider \texttt{if <tautology> then }e_1\texttt{ else }e_2\texttt{ where }e_1\texttt{ has property }P\text{ but }e_2\texttt{ does not.}
Types versus program analysis (5)

This talk will focus on types, rather than program analysis.

Many of the types we discuss will be enriched variants of existing type systems (e.g. “non-aliased pointer to an int”).

- software engineering: want designs which have natural-looking types which don’t clutter the actual code excessively.

- relation to logic: “conservative extension” (see later).
A software-engineering view of where we’re going

Examples of things we want to do.

- class Tree { Int val; Tree left, right; }
  
  If $x$ is of type Tree then is it a tree? [Generally no!]
  How could we ensure it is a tree?

- An operating system might implement runnable and sleeping process queues. Can a compiler ensure each process is on exactly one queue? [Mention boogie-like stuff]? 

- Can we check whether concurrent threads have race-free access to unlocked data? [Particularly important on multi-core processors.]

- Can a private memory allocator issue memory, but (statically) know that no allocation remains in use when it is finalised?

- Are any write() operations called without an open()?
A software-engineering view . . . (2)

There’s much commonality between these answers to these questions and how monads encapsulate state in Haskell.

Note the central concern over what a pointer can point to, and whether it is aliased.

Let’s re-look at the course abstract:

Dijkstra’s famous maxim “go to statement considered harmful” noted that spaghetti-like code was hard to reason about. However, in the presence of side effects and (especially) concurrency, spaghetti-like data is far harder to understand – not only in terms of the full behaviour of a program but also in terms of lesser properties such as performance prediction on multicore architectures. . . .
Multi-core processors: the user view

Multi-core is perhaps the *biggest change to processor design* visible to the user for a generation. The only two comparable things (both 1960s) are:

- indexed addressing “movl 4(%eax),%ebx”
- interrupts

What happened in between? Moore’s law annually gave more transistors but these were spent gaining speed (pipelining, caches etc.) [otherwise pretending nothing had happened]. After hitting the (3-5GHz) speed limit on Pentium-style processors on CMOS the marketing departments started selling ‘multicore’ where they had previously sold ‘speed’.
Two ‘obvious’ but *incorrect* statements:

1. If a program with two threads runs on a single-core processor then it will run unchanged on a two-core processor.

2. (After correcting problems in 1.) a two-thread program will run faster on a two-core processor than it runs on a single-core processor.
Multi-core and sequential consistency

On a single-core (implementing tasking using interrupts) the instructions in each thread are interleaved.

Consider possible outputs for the program:

```plaintext
volatile int x=0,y=0;
thread1: { x=1; print "y=",y; }
thread2: { y=1; print "x=",x; }
```

For most executions the program prints x=0, y=1 or x=1, y=0;

- relatively rarely it prints x=1, y=1;
- but it never prints x=0, y=0.

But on multi-core …
Multi-core and sequential consistency (2)

```c
volatile int x=0,y=0;
thread1: { x=1; print "y=",y; }
thread2: { y=1; print "x=",x; }
```

But on multi-core (a particular Intel x86 CPU) typical frequencies might be (over 1 000 000 runs):

<table>
<thead>
<tr>
<th></th>
<th>y=0</th>
<th>y=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=0</td>
<td>500</td>
<td>499500</td>
</tr>
<tr>
<td>x=1</td>
<td>499500</td>
<td>500</td>
</tr>
</tbody>
</table>

Not all executions interleave instructions from the two threads (they do on a single-core processor with interrupt-driven scheduling).

Failure of *sequential consistency*. 
Multi-core and sequential consistency (3)

```c
volatile int x=0,y=0;
thread1: { x=1; print "y=",y; }
thread2: { y=1; print "x=",x; }
```

Why can \(x=0, y=0\) happen?

- On a single thread it is quite valid for a read and a write to distinct locations to be re-ordered
- Single-core processors exploit this for speed (pipelining)
- Manufacturers of multi-core processors want a single CPU of a multicore processor to be as fast as a CPU of a single core.

Solution: programmer’s responsibility to fix such races, e.g. by locking, or using `mfence`.
Find “[Relaxed] Memory Model” for guidance.
MFENCE – Memory Fence

MFENCE

Serializes load and store operations.

**Description**  Performs a serializing operation on all load and store instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes in program order the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible. . . .

[http://www.intel.com/.../documentation/instructions/.../vc172.htm]
Multi-core and sequential consistency (4)

So, to make our program ‘work’ one does:

```c
volatile int x=0,y=0;
thread1: { x=1; mfence(); print "y=",y; }
thread2: { y=1; mfence(); print "x=",x; }
```

Note that `mfence` can take 100 CPU cycles …
Incorrect statement: a two-thread program will run faster on a two-core processor than it runs on a single-core processor.

Reason: caches.
A programmer’s view of memory

CPU \rightarrow \text{MEMORY} \quad \text{1 cycle}

(This model was pretty accurate in 1985.)

A modern view of memory and timings

CPU \rightarrow \text{L1 cache} \rightarrow \text{L2 cache} \rightarrow \text{MEMORY}

2 \rightarrow 10 \rightarrow 200
Multi-core-chip memory models

Today’s model (cache simplified to one level):

```
CPU 1   -->  CACHE 1
          | 2
          v
CPU 2   -->  CACHE 2
          | 200
          v  coherency
MEMORY  
```

Types for State and Aliasing
Why can a program run slower on multicore?

- when CPU1 writes into a cache line, data in CPU2’s cache is discarded.

- if CPU2 now writes this cache line, the line must be reloaded (even from memory) and data in CPU1’s cache discarded.

- A two-core version of virtual memory ‘thrashing’ (or of repeated cache reloading in naive big-matrix multiplication)

It’s harmless (or even good) if two threads running on the same processor core both access a cache line (since all accesses will hit the cache), but . . .

it’s bad if two threads running on different processor cores access the same cache line.

Compare: multicore cache model and multiple-reader/single-writer.
Lecture Conclusion: How should we react?

- Understand these phenomena – to avoid writing buggy/slow code
- Design languages and type-like systems to enable the compiler to reject buggy code at compile time.
- Design languages to make slow code less convenient to write.

The Haskell use of monads is a good example of this process: there was a requirement to have lazy evaluation but still do I/O.

Haskell history: monads were the end of a chain of proposals, including (e.g.) continuation-passing style, and the core theoretical idea was developed into a language feature by adding (for Haskell experts) parametric structure (“type classes”) and syntax (“do”) which make them easier to use.

Let’s try and copy this.
Lecture 2: Languages and Type Systems

Inference rules, correctness (type safety). ⊢, |=.

Program Analysis and Type Inference

Aim: revision on type inference and relation to program analysis

Lecture Plan

To introduce checking and inference of program properties, and properties like safety by considering the special case of traditional type systems.

Although this course concentrates on types rather than program analysis, I’ll try to present material neutrally – e.g. “can we guess a type for ...?”.
Revision: simple (ML-like) languages

\[ e ::= \begin{array}{l}
\quad x \quad \text{variable} \\
\quad c \quad \text{constant} \\
\quad \lambda x.e \quad \text{lambda-abstraction} \\
\quad e_1 \ e_2 \quad \text{application} \\
\quad \textbf{let} \ x = e_1 \ \textbf{in} \ e_2 \quad \text{local declaration} \\
\quad \textbf{if} \ e_1 \ \textbf{then} \ e_2 \ \textbf{else} \ e_3 \quad \text{if-then-else}
\end{array} \]

Constants \( c \) include \( n \in \mathbb{Z} \) (integers) and \( b \in \mathbb{B} \) (booleans). Programs are just closed expressions.

At the moment this is purely functional; we can specify its semantics to be eager (in which case we can also add functions to \( c \) representing side-effecting primitives) or lazy (in which case this would be unwise as humans find such reasoning hard!).
Revision: simple (ML-like) languages (2)

For various purposes we want bigger languages. Possibilities:

- add $x := e$ to allowed expression forms to mutate local variables (might also write $e ; e'$ as useful sugar for $\texttt{let } x = e \texttt{ in } e'$)

- add forms $\texttt{new}(T), e.f$ and (optionally) $e.f := e$ to allow storage allocation; $T$ is some form of (type) descriptor specifying the possible fields $f$ which can be used in the other two forms (ML’s $\texttt{ref}$ type is a special case). Perhaps even add $\texttt{free}(e)$?

- add forms $\xi! e. e'$ to output $e$ on channel $\xi$ before returning $e'$ and $\xi? x. e$ to read from channel $\xi$, binding $x$ the value read within $e$. 
Revision: simple (ML-like) languages (3)

For various purposes we might want smaller languages. Examples:

- we might want to study a lazy language and its optimisations so we might replace *implicit* laziness to *explicit* representations of closures including a flag saying “already evaluated”.

- sometimes higher-order features are easy to consider, sometimes we might want to eliminate them from analysis

- we might want to make “order of evaluation” explicit by explicitly sequencing function calls, thus forbidding \((e \, e')\) in the syntax and requiring

\[
\text{let } x = e \text{ in let } x' = e' \text{ in } x \, x'
\]

(ANF form, similar in spirit to “continuation passing form”)
Revision: simple (ML-like) languages (4)

Which to choose (for framing a research problem)?

• the one which makes your problem easiest to state and to reason about (makes analysis and optimisations easier to express too).

• be brutal: restrict your problem (at least when doing the first round of theory) all all means at your disposal: syntactic restrictions, type restrictions etc.

• note my language is full of danger: the untyped lambda calculus is Turing powerful so exact properties of its evaluation are undecidable even without arithmetic being added via constants $c$. (Might use a type regime to stop $Y$ being definable.)
Revision: simple (ML-like) languages (5)

Example: we have side-effecting primitives and don’t want to worry about order of evaluation (or unnamed values being stored in temporaries on a stack) and higher-order functions give our analysis problems. So we could use

\[
\begin{align*}
  v & ::= x \mid c \\
  e & ::= \text{if } v \text{ then } e_1 \text{ else } e_2 \\
    & \quad \mid \text{let } (x_1, \ldots, x_j) = f(v_1, \ldots, v_k) \text{ in } e \quad (j, k \geq 0) \\
    & \quad \mid (v_1, \ldots, v_k) \quad (k \geq 0)
\end{align*}
\]

to arrange there to be: simple expressions \(v\) and sequenced expressions \(e\) and that function values \(f\) (multi-arg, multi-result) are a special syntactic class which cannot leak into expressions.
Revision: simple (ML-like) type systems

Non-polymorphic types: $t$ ranges over Type:

$$t ::= int \mid bool \mid t \to t$$

$\Gamma$ is a list of type assumptions: variable-type pairs for variables in scope. Have the judgement form $\Gamma \vdash e : t$ defined by typing rules:

(VAR) $\Gamma [x : t] \vdash x : t$

(INT) $\Gamma \vdash n : int$

(BOOL) $\Gamma \vdash b : bool$

(LAM) $\Gamma [x : t] \vdash e : t'$

(APP) $\Gamma \vdash e_1 : t \to t' \quad \Gamma \vdash e_2 : t$

$\Gamma \vdash e_1 \; e_2 : t'$

(COND) $\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t$

$\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t$

Eager or lazy makes no difference to typical type systems (see later).
Can we ‘analyse’ a program for its type?

I.e. given $e$ find $t$ such that $\Gamma \vdash e : t$?

Yes and no! Temporarily treat primitives as free variables:

$$\Gamma_0 = [+ : \text{int} \rightarrow \text{int} \rightarrow \text{int}, \neg : \text{bool} \rightarrow \text{bool}]$$

Then, writing $+$ as infix:

- $\Gamma_0 \vdash \lambda x.x + 1 : \text{int} \rightarrow \text{int}$  \quad (one such $t$)
- $\Gamma_0 \vdash \lambda x.\neg x : \text{bool} \rightarrow \text{bool}$  \quad (one such $t$)
- $\Gamma_0 \vdash \lambda x.x : \text{int} \rightarrow \text{int}$  \quad (many such $t$)
- $\Gamma_0 \vdash \lambda x.x : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$  \quad (another such $t$)
- There is no $t$ with $\Gamma_0 \vdash \lambda x.\neg x + 1 : t$

Some terms have one type, some many and some none (this is the typical situation for type systems – but less so for analyses).
Can we ‘analyse’ a program for its type? (2)

The word ‘analyse’ is traditional for (say) dataflow analysis, but in type-like systems words like ‘checking’, ‘assignment’, ‘inference’ and ‘reconstruction’ are more common.

It’s more a change of emphasis rather than of real meaning:

- Traditionally if a program fails to ‘type check’ then we reject it,
- But if analysing a program for a property (e.g. “may this program produce an out-of-bounds array index?”) then we are quite happy to report “don’t know” but still compile the program (but often with a run-time test).

Program analyses generally need a representation of “property $p$ could not be established” for input to later compiler stages.
Can we ‘analyse’ a program for its type? (3)

Given the language so far, it’s a little hard to talk about ‘type checking’ since the source syntax contains no types, so type-checking $\lambda x. x$ is likely to be problematic (do we ‘guess’ a type $t$ for $x$?). The issue is the LAM rule:

$$(\text{LAM}) \frac{\Gamma \vdash x : t \vdash e : t'}{\Gamma \vdash \lambda x . e : t \to t'}$$

An ‘engineering fix’ is to change the syntax to demand that the user gives types on variable declaration:

$$e ::= x \mid c \mid \lambda x : t . e \mid e_1 e_2 \mid \ldots$$

and modify LAM to:

$$(\text{LAM}') \frac{\Gamma \vdash x : t \vdash e : t'}{\Gamma \vdash \lambda x : t . e : t \to t'}$$

Types for State and Aliasing
Can we ‘analyse’ a program for its type? (4)

With

\[
\frac{\Gamma[x : t] \vdash e : t'}{
\Gamma \vdash \lambda x : t . e : t \rightarrow t'}
\]

(LAM')

type checking is now totally syntax-directed (i.e. can be done with a simple recursive tree walk) with no need to guess any \( t \).

- This means that every program has at most one type.

I.e. the typing-rules are deterministic (define a partial function from closed expressions to types).

We’ll return to type inference or reconstruction in a few slides.

But take away one thing: program analyses might need similar user-level specifications – ‘inference anchors’ – at variable declaration or at interfaces.
Can we ‘analyse’ a program for its type? (5)

Incidentally the LET-rule (not previously given explicitly) has no ‘need-to-guess’ problems as the type $t$ of $x$ in $\texttt{let } x = e_1 \texttt{ in } e_2$ is given by the (previously) calculated type of $e_1$:

$$(\text{LET}) \frac{\Gamma \vdash e_1 : t \quad \Gamma[x : t] \vdash e_2 : t'}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : t'}$$

However, many programming languages insist on a type on all variable introductions. Why:

- to permit separate declaration and initialisation
- emphasis or “redundancy is good for engineering”
- when languages have sub-typing: if class $B$ extends class $A$ in Java then what is the type of $y$ in $\lambda x : B . \texttt{let } y = x \texttt{ in } \ldots$?
Type checking is a simple tree-walk

The syntax-directed nature of the typing rules means that they easily convert into a type-checking (or type-reconstruction later) algorithm:

```haskell
fun typecheck(e: Expr, Gamma: TypingEnv) = case e of
  ExprVar(s) => lookup(s, Gamma);
  ExprInt(n) => TypeInt;
  ExprApp(e,e') => let t = typecheck(e, Gamma);
                 t' = typecheck(e', Gamma);
                 in case t of TypeArrow(_,_) => ... t ... t' ...;
                   _ => raise TypeError;
               end
  ExprLam(s,t,e) => let Gamma' = extend(Gamma, (s,t));
                   t' = typecheck(e, Gamma');
                   in TypeArrow(t, t');
               end
  ...
```

Types for State and Aliasing
What does “having a type” mean?

We could express almost any calculation of some property as a set of inference rules – what makes some good and some bad? E.g.

\[(\text{LAM-BAD}) \quad \Gamma \vdash \lambda x : t \cdot e : \text{int}\]

• Answer: relation to evaluation (i.e. semantics). “Well-typed programs do not go wrong”.

Comparison to dataflow analysis: if we calculate that variable \(x\) is not live at label (program point) \(\ell\) then what does this mean?

Answer: implication that inserting \(x := \text{rand}()\) at \(\ell\) has no effect on the program’s I/O behaviour.

Comparison to logic: inference rules are sound if they are true in all models.
Semantics

[operational, ‘lazy’, small-step, implicit-substitution]

Identify the subset \( v \) of \( e \) corresponding to values:

\[
v ::= c \mid \lambda x. e
\]

Evaluation is here simply \( \beta \)-reduction (and \( \delta \)-reduction):

\[
\begin{align*}
(BETA) \quad & (\lambda x.e)e' \rightarrow e'[e'/x] \\
(COND) \quad & \text{if true then } e_2 \text{ else } e_3 \rightarrow e_2 \\
(APP-CCTX) \quad & e_1 \rightarrow e'_1 \\
& e_1 e_2 \rightarrow e'_1 e_2 \\
(COND-CCTX) \quad & e_1 \rightarrow e'_1 \\
& \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e'_1 \text{ then } e_2 \text{ else } e_3
\end{align*}
\]

Often for human-engineering reasons the context rules are neatly treated by defining a separate notion of “evaluation context”.

Types for State and Aliasing
Termination, non-termination and stuck states

Given \( e \) there are three possibilities:

- **Successful termination giving a value:** \( e \rightarrow^* v \)
  
  e.g. \((\lambda x.x)\text{true}\) or 42.

- **Non-termination:** \( \exists e_i, i \in \mathbb{N} \) \( e = e_0 \rightarrow e_1 \rightarrow \cdots \rightarrow e_n \cdots \)
  
  e.g. \((\lambda x.xx)(\lambda x.xx)\).

- **A stuck state** \( e_n \) is encountered: \( e = e_0 \rightarrow e_1 \rightarrow \cdots \rightarrow e_n \) with there being no \( e' \) such that \( e_n \rightarrow e' \) (with \( e_n \) not being a value).
  
  e.g. \text{true}(\lambda x.x), \text{if} \ \lambda x.x \ \text{then} \ \cdots \) or a free variable like \( x \).

This system was crafted for simplicity and ‘least work’: it is deterministic \((\rightarrow)\) is a partial function\) so only one of these can occur.

We see stuck states as a program “going wrong”.
Meanings of programs and types

Let’s write

\[
[e] = \begin{cases} 
  v & \text{if } e \rightarrow v \\
  \bot & \text{if } e \text{ does not terminate} \\
  \text{wrong} & \text{if } e \text{ becomes stuck}
\end{cases}
\]

as the meaning of \( e \) (we use the denotational semantics notation and indeed we could have defined \([e]\) denotationally instead).

Now let’s define \([t] : \mathcal{P}(Val\bot)\) by

\[
\begin{align*}
\int\!\!\!\lim birk [int] &= \mathbb{Z}_\bot \\
\int\!\!\!\lim birk [bool] &= \mathbb{B}_\bot \\
[t \rightarrow t'] &= \{ \lambda x.e \mid e \in \text{Expr} \}_\bot
\end{align*}
\]

and note that \(\text{wrong} \notin [t]\) for any \( t \).
Well-typed programs do not go wrong

Now we can talk about why type-assignment is sound:

**Theorem (safety of type-assignment)**
Whenever \( \Gamma \vdash e : t \) then \( \llbracket e \rrbracket \in \llbracket t \rrbracket \), in particular evaluation of \( e \) does not go wrong.

Notes:

- I’ve cheated a little in using \( \Gamma \) here while earlier I used \( \Gamma_0 \) containing \(+\) etc.

- It’s simplest to treat such primitives as constants \( c \) having *consistent* types (for the type rules) and behaviours (for the evaluation rules), so e.g. given \(+ : \text{int} \to \text{int} \to \text{int}\) then

\[
(\forall x, y \in \llbracket \text{int} \rrbracket) : x + y \in \llbracket \text{int} \rrbracket
\]
Preservation and Progress

The modern style – pioneered by (Wright and Felleisen 1994) – is to prove type systems sound by preservation and progress lemmas.

**preservation** if \( e : t \) and \( e \rightarrow e' \) then \( e' : t \).

i.e. types are preserved by evaluation steps

**progress** if \( e : t \) and \( e \) is not a value then \( (\exists e')e \rightarrow e' \).

i.e. no stuck states e.g. \((1(2))\) occur in a well-typed program.

By induction these mean that every typeable program \( e : t \) either loops forever or results in a value of type \( t \).
Well-typed programs do not go wrong (2)

Note the similarity of “safety of an analysis”:

\[
\text{whenever } [\ ] \vdash e : t \text{ then } [e] \in [t]
\]

with logical soundness:

\[
(A \vdash \phi) \Rightarrow (A \models \phi)
\]

[proof implies truth].

- A safe/sound/correct system of inference rules (here type checking), only proves things which are true (here evaluation).

- Preservation and Progress are similar to how one proves soundness of inference rules for a logic w.r.t. a model.
Some of the things I didn’t say

I trod carefully through the swamp between “hard core theory” and “viewing the big picture” in this introduction:

• Type assignment is in general not complete:
  \[
  \text{if } \text{true} \text{ then } 42 \text{ else } \text{false} \text{ has no type but evaluates to } 42 \in [\text{int}] \text{ without going wrong. This must be so: in general evaluation is undecidable while we want type assignment to terminate.}
  \]

• I’ve been careless with the semantics of function types, saying
  \[
  [t \rightarrow t'] = \{\lambda x.e \mid e \in Expr\}_\bot. \text{ Hence } [\text{bool} \rightarrow \text{bool}] \text{ wrongly contains values like } \lambda x.x + 1 \text{ and even } \lambda x.(\neg x) + 1.
  \]
  This is harmless for my limited use of it in correctness, but function types should really include recursive constraints on argument values and result values.
Some of the things I didn’t say (2)

We had ‘lazy’ semantics (but actually call-by-name implementation):

\[
\begin{align*}
\text{(Beta)} & \quad (\lambda x.e)e' \rightarrow e'[e'/x] \\
\text{(App-Ctxt)} & \quad e_1 \rightarrow e_1' \\
& \quad e_1 e_2 \rightarrow e_1' e_2
\end{align*}
\]

We can make this eager by replacing BETA with two rules:

\[
\begin{align*}
\text{(Beta-V)} & \quad (\lambda x.e)v_2 \rightarrow e[v_2/x] \\
\text{(App-Rctxt)} & \quad e_2 \rightarrow e_2' \\
& \quad v_1 e_2 \rightarrow v_1 e_2'
\end{align*}
\]

Or we could use BETA, APP-Ctxt and APP-Rctxt to give non-deterministic choice of when an argument is evaluated.

All of these variants have the type soundness property (expressed as \([e] \subseteq [t]\) as \([e]\) is now a set of values); so eager/lazy tends not to matter for simple type systems (but might matter for richer type systems, e.g. modelling store use).
Alternative semantics

[operational, eager, big-step, implicit-substitution]

Again identify subset $v$ of $e$ corresponding to values:

$$v ::= c \mid \lambda x.e$$

Then give evaluation rules:

\[
\begin{align*}
\text{(CONST)} & \quad c \Downarrow c \\
\text{(COND)} & \quad \frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \\
\text{(LAM)} & \quad \frac{}{\lambda x.e \Downarrow \lambda x.e} \\
\text{(APP)} & \quad \frac{e_1 \Downarrow \lambda x.e_3 \quad e_2 \Downarrow v_2 \quad e_3[v_2/x] \Downarrow v_3}{e_1 \ e_2 \Downarrow v_3}
\end{align*}
\]

We can do type-soundness this way, but the problem is that stuck states are indistinguishable from looping unless we add special evaluation rules for them, e.g. $42(\lambda x.x) \Downarrow \text{wrong}$. 

*Types for State and Aliasing*
Denotational Semantics (sketch)

Have CPO of values $D$ given by

$$D = \mathbb{Z}_\bot \oplus \mathbb{B}_\bot \oplus (D \rightarrow D)_\bot \oplus \{wrong\}_\bot$$

Environments $Env$ are simply $Var \rightarrow D$.

Given expression $e$, its (lazy) semantics $\llbracket e \rrbracket \in Env \rightarrow D$ is:

$$\llbracket x \rrbracket_\rho = \rho(x)$$

$$\llbracket \lambda x. e \rrbracket_\rho = in_3(\lambda d. \llbracket e \rrbracket_\rho[d/x])$$

$$\llbracket e_1 e_2 \rrbracket_\rho = \bot \quad \text{if } \llbracket e_1 \rrbracket_\rho = \bot$$

$$= f(\llbracket e_2 \rrbracket_\rho) \quad \text{if } (\exists f)\llbracket e_1 \rrbracket_\rho = in_3(f)$$

$$= in_4(wrong) \quad \text{otherwise}$$

Again we encode $wrong$ as an explicit value.
Denotational Semantics (sketch) (2)

Types are interpreted now a sub-domain of $D$:

\[
\begin{align*}
\llbracket \text{int} \rrbracket &= \{ in_1(n) \mid n \in \mathbb{Z} \}_\perp \\
\llbracket \text{bool} \rrbracket &= \{ in_2(b) \mid b \in \mathbb{B} \}_\perp \\
\llbracket t \rightarrow t' \rrbracket &= \{ in_3(f) \mid f \in D \rightarrow D \text{ s.t. } (\forall d \in \llbracket t \rrbracket) f(d) \in \llbracket t' \rrbracket \}_\perp
\end{align*}
\]

Note the recursive constraints on (even higher order) function types.

Other people use PERs for semantics of types (outside this talk).

Advice: choose your semantic formalism to make proving things you need to prove as easy as possible.
ML-style polymorphic types

Use $\alpha$ to range over type variables, $\tau$ over types, and $\sigma$ over type schemes:

$$
\tau ::= \alpha | \text{int} | \text{bool} | \tau \rightarrow \tau \\
\sigma ::= \tau | \forall \alpha.\sigma
$$

Assumptions $\Gamma$ are now (variable, type scheme) pairs (so that environments can contain functions like $\text{id}$ and $\text{map}$).

Type schemes represent polymorphic types (see later for the differences).
ML-style polymorphism typing rules

(VAR) \( \Gamma[x : \sigma] \vdash x : \sigma \)

(INT) \( \Gamma \vdash n : int \)

(BOOL) \( \Gamma \vdash b : bool \)

(LAM) \( \Gamma[x : \tau] \vdash e : \tau' \) → \( \Gamma \vdash \lambda x.e : \tau \rightarrow \tau' \)

(APP) \( \Gamma \vdash e_1 : \tau \rightarrow \tau' \) → \( \Gamma \vdash e_2 : \tau \) → \( \Gamma \vdash e_1 \, e_2 : \tau' \)

(COND) \( \Gamma \vdash e_1 : bool \) → \( \Gamma \vdash e_2 : \tau \) → \( \Gamma \vdash e_3 : \tau \) → \( \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \)

(LET) \( \Gamma \vdash e_1 : \sigma \) → \( \Gamma[x : \sigma] \vdash e_2 : \tau \) → \( \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau \)

(SPEC) \( \Gamma \vdash e : \forall \alpha.\sigma \) → \( \Gamma \vdash e : \sigma[\tau/\alpha] \)

(GEN) \( \Gamma \vdash e : \sigma \) → \( \Gamma \vdash e : \forall \alpha.\sigma \) (\( \alpha \) not free in \( \Gamma \))

Most rules are syntax-directed but GEN and SPEC are not.
ML-style polymorphism typing rules (2)

Why these subtle rules for $\sigma$ and $\tau$ and in SPEC/GEN?

Soundness (a.k.a. type safety)! Consider (adding pairs to the language for this example):

- $f(x) = \text{if } x \text{ then } x \text{ else } 42$ is ill-typed (not type $\alpha \rightarrow \text{int}$).
- $\text{let } id = \lambda x.x \text{ in } (id(42), id(true)) : \text{int} \times \text{bool}$
- $(\lambda id.(id(42), id(true)))(\lambda x.x)$ is ill-typed
- But if we $\beta$-reduce the previous example we get $((\lambda x.x)(42), (\lambda x.x)(true)) : \text{int} \times \text{bool}$

Note that the last three are are semantically equivalent.
ML-style polymorphism typing rules (3)

Points:

• Type safety for polymorphism is subtle

• The Curry-Howard correspondence between types and logic (formulae = types, proofs = programs) justifies the appropriateness of these rules at a deep level.

• The fact that the two programs above (differing only between ‘let’ and ‘apply-of-lambda’) have identical semantics but different type behaviour is analogous to the fact that expressions like \((x - 1)^2\) and \(x^2 - 2x + 1\) may have identical semantics but it’s simpler to show the former is always positive during program analysis.
ML-style polymorphism typing rules (4)

So, ML polymorphic type-assignment is often called “type inference” or “type reconstruction”. Why?

- the ‘principal type’ property: when we have to ‘guess’ a type for a $\lambda$- or let-binding we can always choose one at least as general as the programmer intended. SPEC then fixes up at uses.

- “type reconstruction” is becoming a more popular phrase – the idea is that the program is platonic and each binding has types, but they have been erased. ‘Reconstruction’ just puts them back.

What’s special about ML-style polymorphism?

- ‘calculate the principal type’ is a partial function (not merely a set of possible types).
How do we implement non-syntax-directed rules like SPEC and GEN, and the ‘guesses’ for LAM?

- The guesses for LAM can be done by unification

- SPEC and GEN rules can be incorporated (respectively) into the VAR and LET rules (and their originals deleted) – this gives a simple-ish tree-walking algorithm.

This restores the view of ‘type inference’ as ‘program analysis’
ML-style polymorphism typing rules (6)

One needs to take care to formulate *correctly* progress and preservation in the presence of “principal type inference”.

Consider

\[ e_1 = (\text{if true then } \lambda x.x \text{ else } \lambda x.x + 1) : int \rightarrow int \]

\[ e_2 = (\lambda x.x) : \alpha \rightarrow \alpha \]

but note that \( e_1 \rightarrow e_2 \).

The resolution is that \( e_2 \) still has type \( int \rightarrow int \), but it has gained an additional (but useless and distracting for proving preservation) type \( \alpha \rightarrow \alpha \).
ML-style polymorphism is too well-behaved

We’ve been spoilt by the nice behaviour of ML type inference. We can erase all the types from a program which type checks and the system can reconstruct a fully-typed program which is at least as general as the original.

But this is quite unusual – it doesn’t hold for many type systems and it’s not going to hold for the sort of syntax-directed program property inference we’re going to consider.
ML-style polymorphism is too well-behaved (2)

Consider extending the ML type system (to the Girard/Reynolds System F) so that types are

\[ \tau ::= \alpha \mid \text{int} \mid \text{bool} \mid \tau \rightarrow \tau \mid \forall \alpha. \tau \]

Thus \( \forall \) can now occur within types not just at outermost level. Here is a term which has two different incomparable types:

\[ \lambda f. \lambda (x, y). (f(x), f(y)) : \forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \times \alpha) \rightarrow (\beta \times \beta) \]

\[ \lambda f. \lambda (x, y). (f(x), f(y)) : \forall \alpha \beta. (\forall \gamma. \gamma \rightarrow \gamma) \rightarrow (\alpha \times \beta) \rightarrow (\alpha \times \beta) \]

Once the types are erased from the lambdas we can’t reconstruct them (it’s actually undecidable).

Remember my earlier remark: [more sophisticated type systems or] program analyses might need ... user-level specifications – ‘inference anchors’ – at variable declaration or interfaces.
Subtyping

Sometimes we want to regard one type as a subtype of another, e.g.

- $A \preceq B$ (if $A$ is a subclass of $B$),
- $\text{int} \preceq \text{real}$, or
- $\forall \alpha. \alpha \to \alpha \preceq \text{int} \to \text{int}$.

This needs a type rule such as

$$\text{(SUBT) } \frac{\Gamma \vdash e : \sigma \quad \sigma \preceq \sigma'}{\Gamma \vdash e : \sigma'}$$

and we expect $\preceq$ to behave like:

$$\text{(TRANS) } \frac{t \preceq t' \quad t' \preceq t''}{t \preceq t''} \quad \text{(FUNSUB) } \frac{t' \preceq t \quad u \preceq u'}{t \to u \preceq t' \to u'}$$
Lecture Conclusion

What have we done?

• given syntax for expressions and types.

• given (syntactic) inference rules giving expressions types ($\vdash$).

• given a semantics to the language ($|=\$).

• noted type soundness ($\vdash \Rightarrow |\$\$) can be proved by “preservation and progress” theorems.

• wondered to what extent types can be inferred.

Next lecture: can we control pointers by types or program analysis?
Lecture 3: Types to control spaghetti data

Spaghetti data is a problem:

- for Software Engineering and
- for local reasoning.

particularly when a distant [i.e. unexpected!] pointer is used to mess with our data.

Solutions: ownership types, substructural types, separation logic, fractional ownership, typestate.

Example enriched type: effect systems. Relation to monads.
Controlling where pointers point

There is a clash of two worlds:

- Object-oriented systems say “everything is an object”, any object can refer to any other object.

- Both humans and compilers (and program comprehension tools in IDEs) want to answer questions like “are there any other pointers [apart from those I expect] to this object?”.

Such questions about aliasing make OO programs hard to understand.

- Think ‘feature interaction’ – mutations happen, but their interaction with aliasing makes their effects hard to understand.

- We want local (or modular) reasoning
Controlling where pointers point (2)

Examples:

- Suppose \( a = \text{concat}(b, c) \); means append list \( c \) to list \( b \) by modifying the last member of \( b \).

  What’s the effect of \( \text{concat}(b, b) \)? Did we intend this?

- \[
\text{class A} \{ \ ... \ \};
\text{class B} \{ \ ... \ \};
\text{class C} \{ \text{private A p;}
\text{public int f()} \{ \text{int x=p.m; B.f(); return x-p.m; } \}
\text{<rest>}
\}
\]

  How private (‘encapsulated’) is \( p \)? [lecture 1 example]

- Might concurrent threads write to the same object?
Some approaches

Two or three separate solutions to this sort of problem have arisen from different communities. These can be classified roughly:

- Ownership types: arose from OO community. Like parameterised types. Refine Java-like `Box<Int>` into `Box<owner,Int>`.

- Sub-structural type systems: arose from theorists, e.g. programming variants of Girard’s *Linear Logic*. A key concept is that attempting to read a variable twice in a scope may be illegal, or allow modification only on the second occurrence. Instances are: linear types, quasi-linear types, uniqueness types, separation logic.

We’ll concentrate mainly on the latter.
Some approaches (2)

A main difference in expressivity centres around

- sub-structural types tend to concentrate on situations where an object has zero or one pointers to it (though recent works on ‘fractional ownership’ have refined this).

- ownership types express encapsulation rather than ‘number of pointers’.

- ownership types been more pragmatically-inspired while sub-structural type systems cover a wider range of possible systems (e.g. possible to capture “polynomial execution time” in a sub-structural system).
Some approaches (3)

Another subject which I want to mention, but not actively discuss, is dual to ownership. E.g.

- Are any `o.write()` operations called without an `o.open()` call having preceded it (and no intervening `o.close()` too)?

An approach to this is so-called ‘typestate’. A recent paper is “Typestate-Oriented Programming” [Aldrich et al. 2009] who write

“Typestate-Oriented Programming … where objects are modelled not just in terms of classes, but in terms of changing states. Each state may have its own representation and methods which may transition the object into a new state. A flow-sensitive, permission-based type system helps developers track which state objects are in.”

but they also note that “sharing” [aliasing] remains a “challenge”
Ownership types

Object-orientation encourages *encapsulation*: ‘public’, ‘private’ and ‘protected’ qualifiers control exports *statically*.

Being able to do *local reasoning* requires further knowing that: all (relevant) objects pointed to by given object are accessible only within that object (i.e. outside objects cannot interfere).

[Note a choice: relevant means ‘important for reasoning: i.e. sub-object rather than (say) a member of a list.]

This is true encapsulation, but not expressible in Java/C#.

Known also as “owners as dominators”.

A big field is that of *static ownership types*.
Owners as dominators

class EmbeddedCPU { ... }  
class Brakes { private EmbeddedCPU cpu; }  
class Engine { private EmbeddedCPU cpu; }  
class Car { private Engine e; private Brakes b; ... }

• For safety, I want no other Car to have access to my Car’s Engine, Brakes or EmbeddedCPU.

• But I’m (probably) happy with my Engine and Brakes both having access to the same EmbeddedCPU.

So, some sharing is OK, but not others (and remember that private alone is insufficient to control access).
Owners as dominators (2)

Definition: given a graph with root $r$ then node $n$ dominates node $n'$ if every path from $r$ to $n'$ passes through $n$.

![Diagram showing ownership relations between Car, Engine, Brakes, and EmbeddedCPU]

Want to require instances of \texttt{Car} to dominate (i.e. \textit{own}) instances of \texttt{Engine}, \texttt{Brakes} and \texttt{EmbeddedCPU} (often called \texttt{rep} objects). Note that neither \texttt{Engine} nor \texttt{Brakes} own \texttt{EmbeddedCPU}.
Owners as dominators (3)

Dominator are dynamic (run-time) properties. Can we enforce domination-based ownership statically? [Yes!]

I’ll discuss briefly one of the early systems that did this: “Ownership types for flexible alias protection” (Clarke, Potter Noble, OOPSLA’98).
Static Ownership types

Ideas (Clarke, Potter Noble, OOPSLA’98):

- Have a tree of contexts ranged over by $M$, root context is $\text{norep}$. Symbols $\text{rep}$ and $\text{owner}$ represent special, implicit, contexts (like $\text{this}$ represents objects).
  
  Each object owns a context, and is owned by the context it lives within. (Note object-oriented not class-oriented.)

- Type of values are $c(M|M^*)$ with $c$ a class name; with $M$ the owner and $M^*$ additional context parameters. E.g. $\text{Foo<norep>}$.

- static visibility constraint limits access to fields/methods having $\text{rep}$ in their type.
Static Ownership types (2)

Correctness of (Clarke, Potter Noble, OOPSLA’98).

- Preservation and Progress lemmas:
- set up dynamic execution model and check ownership at run-time
- prove that ownership types are preserved during execution (preservation)
- prove that dynamic links respect ownership types during execution (progress)

Standard type-like argument.
Static Ownership types (3)

Properties of (Clarke, Potter Noble, OOPSLA’98):

- static nesting of contexts.
- no run-time cost.
- object-level version of class-based public/private.
- ownership is static, and hence can’t be transferred.
- can model container objects where having some parts are encapsulated (rep) and others are owner by the user via context parameters.
- problems with typing *iterators* addressed in later paper
Substructural type systems

Quite general (and a huge subject field). Basically replace rules like

\[
\frac{\Gamma \vdash e \quad \Gamma \vdash e'}{\Gamma \vdash ee'} \quad \text{with} \quad \frac{\Gamma \vdash e \quad \Delta \vdash e'}{\Gamma, \Delta \vdash ee'}.
\]

Substructural types (includes linear types, uniqueness types, separation logic):

- (effectively) control the number of pointers to an object (often zero/one)

- allow easy ownership transfer

- variables can have different types at different points during execution.
Intermission

Before looking at substructural types in detail, let’s look at a type system for representing side-effects.

Note the need to be able to express something in a language (e.g. aliasing and/or side-effects) before it can be discussed/controlled.

Compare ‘Newspeak’ in George Orwell’s “1984”. Newspeak was incapable of expressing political dissent in words: thus “thought-crime” was no longer possible.
Type and Effect Systems

These are more richly-extended type systems which can model side-effects or similar computational effects. For example: start with a language and type system as before but add channels $\xi$ with (integer-only) read $\xi?x.e$ and write $\xi!e.e$ operations:

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \mid \xi?x.e \mid \xi!e_1.e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3.$$  

Reading and writing might have type rules as follows:

$$(\text{READ}) \quad \frac{\Gamma[x : \text{int}] \vdash e : t}{\Gamma \vdash \xi?x.e : t}$$  

$$(\text{WRITE}) \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \xi!e_1.e_2 : t}.$$
Type and Effect Systems (2)

One useful question for program optimisation is “can this expression have side-effects?” (note this is non-trivial for higher order functions, just like determining whether a program may “go wrong”).

Here the set of effects $F$ are the reads and writes which take place and on which channel they do so:

$$F \subseteq \{W_\xi, R_\xi \mid \xi \text{ a channel}\}.$$

The issue to model is that program

$$\xi!1. \lambda x. \xi!2. x$$

has an immediate effect of writing to $\xi$ but also a latent effect of writing to $\zeta$ via the resulting $\lambda$-abstraction (when/if it is called).
Extend judgements to be $\Gamma \vdash e : t, F$ ($F$ is the immediate effect of evaluating $e$), and extend types to be

$$t ::= \text{int} \mid t \xrightarrow{F} t'.$$

Here $F$ is the latent effect of a function (crystallised when it is called) encoded into its type.
(VAR) $\frac{\Gamma[x : t] \vdash x : t, \emptyset}{\Gamma \vdash x : t, \emptyset}$

(LAM) $\frac{\Gamma[x : t] \vdash e : t', F}{\Gamma \vdash \lambda x.e : t \rightarrow t', \emptyset}$

(APP) $\frac{\Gamma \vdash e_1 : t \xrightarrow{F''} t', F \quad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash e_1 \ e_2 : t', F \cup F' \cup F''}$

(READ) $\frac{\Gamma[x : \text{int}] \vdash e : t, F}{\Gamma \vdash \xi ?x.e : t, \{R_{\xi}\} \cup F}$

(WRITE) $\frac{\Gamma \vdash e_1 : \text{int}, F \quad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash \xi !e_1.e_2 : t, F \cup \{W_{\xi}\} \cup F'}$

Parts in black are the traditional type system, parts in red are the additions to model effects – note particularly rules APP and LAM.
But again it’s not so simple . . .

The obvious rule for if-then-else is:

\[
\frac{\Gamma \vdash e_1 : \text{int}, F \\ \Gamma \vdash e_2 : t, F' \\ \Gamma \vdash e_3 : t, F''}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t, F \cup F' \cup F''}.
\]

Note that the second \( \cup \) implies that effects may be overestimated.

But there’s still a problem:

\[
\text{if } x \text{ then } \lambda x. \xi!3. x + 1 \text{ else } \lambda x. x + 2
\]

is ill-typed (the types of \( e_2 \) and \( e_3 \) mismatch because their latent effects differ).
Again sub-typing (or sub-effecting) comes to the rescue:

\[ \frac{\Gamma \vdash e : t \xrightarrow{F'} t', F}{\Gamma \vdash e : t \xrightarrow{F''} t', F} \text{ (provided } F' \subseteq F'' \text{)} \]

A more general form is to define subtypes by two rules: \( t \subseteq t \) and a (co- and contra-variant) rule for function types:

\[ \frac{t'_1 \subseteq t_1 \quad t_2 \subseteq t'_2}{t_1 \xrightarrow{F} t_2 \subseteq t'_1 \xrightarrow{F'} t'_2} \text{ (provided } F \subseteq F'' \text{)} \]
Type and Effect Systems (7)

As defined the effects of a program form a set. Examples:

\[ \xi ? x. \zeta!(x + 1). 42 : \text{int}, \{R_\xi\} \cup \{W_\zeta\} \]

and

\[ \zeta!7. \xi ? x. 42 : \text{int}, \{W_\zeta\} \cup \{R_\xi\}. \]

However, we can vary the effect algebra by choosing \( F \) other than a set and defining the \( \emptyset, \{\cdot\} \) and \( \cup \) operators on effects (so that e.g. \( \cup \) is interpreted as non-commutative sequencing.

See also “session types” for \( \pi \)-calculus protocols – similar to regular expressions.
Effect system safety

Just treat as augmented type system:

- augment (‘instrument’) the semantics $\llbracket e \rrbracket$ for terminating programs to give a pair $(v, f)$ with $f$ the set of effects which actually occur during evaluation.

- safety of a closed expression is now

$$([ ] \vdash e : t, F) \Rightarrow (v \in \llbracket t \rrbracket \land f \subseteq F \text{ where } (v, f) = \llbracket e \rrbracket)$$

Exercise: what does safety mean for non-terminating programs?
Many systems can be seen as types and effects.

An important one to us is that of effects being the set of memory locations an expression can read from or write to.

We may run (say) \( e_1 \) and \( e_2 \) occurring in \( e_1 + e_2 \) in parallel in ML whenever the set of writes in one expression do not intersect the writes and reads of the other.

Similarly for many effect systems if \( f : int \overset{\emptyset}{\rightarrow} int \) then \( f(1) + f(1) \) may be optimised to \( \text{let } x = f(1) \text{ in } x + x \).
Type and Effect Systems/Conservative Extension

When enriching a type system (e.g. with effects) we might want to ensure all our existing programs continue to type-check in the enriched system (perhaps by giving unhelpful annotations such as “all functions might have side-effects on anything at all”).

Note a similar idea in logic: take languages of formulae \( \mathcal{L} \) and \( \mathcal{M} \) having theorems \( \Theta_\mathcal{L} \) and \( \Theta_\mathcal{M} \) with \( \mathcal{L} \subseteq \mathcal{M} \); we say \( \mathcal{M} \) is a conservative extension of \( \mathcal{L} \) if \( \Theta_\mathcal{L} = \mathcal{L} \cap \Theta_\mathcal{M} \).

But beware: we need to generalise the definition of “conservative extension” slightly to account for enriched type systems when \( \mathcal{L} \) is only a subset of \( \mathcal{M} \) after an ‘erasure’ operation (such as removing effects from all arrows).
Region types

We can define a type-and-effect system in which effects are values like $\text{read}_\rho$, $\text{write}_\rho$, $\text{alloc}_\rho$ where $\rho$ represents a region. Regions are abstract disjoint areas of memory.

There’s not enough time to discuss this more here, but see papers by Tofte and Talpin or the book chapter by Henglein, Makholm, Niss.
Type and Effect Systems are Monads

An aside: it’s worth observing that types and effects are formally monads [Wadler and Thiemann, 1998].

We create a set of type constructors, $M^F$, one for each effect $F$. Let $B$ be a base type; the translation is:

$$
\begin{align*}
\llbracket B \rrbracket &= B \\
\llbracket t \xrightarrow{F} t' \rrbracket &= \llbracket t \rrbracket \to M^F \llbracket t \rrbracket \\
\llbracket e : t, F \rrbracket &= e : M^F \llbracket t \rrbracket
\end{align*}
$$

Old judgements $\Gamma \vdash e : t, F$ now simply become $\Gamma \vdash e : \tau$ where

$$
\tau ::= B \mid \tau \to \tau \mid M^F \tau
$$
Can variables change their type?

In the systems we’ve seen above the type assumptions for a variable are made when they are declared \textit{and never change}.

This quite a special case. Let’s look at some examples.
When variables change type

• \( x := "abc"; \ldots; x := 3; \) in a dynamically typed language.

• \texttt{movl \%eax,0(\%eax)} loading the content of a cell from a pointer.

Here the key issue is that a variable (\( x \) or \( \%eax \)) is being \textit{redefined} – and in general its new value may be of a different type.

Solution: transform to \textit{“SSA: single static assignment”} form (entertainingly, we can see transformation to SSA as being an inverse of register allocation). The above cases become

• \( x_1 := "abc"; \ldots; x_2 := 3; \)

• \texttt{movl \%eax_2,0(\%eax_1)}

Technology: we need \( \phi \) functions at control path merges.

Appel: SSA is functional programming.
When variables change type (2)

Transforming to SSA form enables some programs to be now typed which were not typeable before.

However it’s not a general solution for programs fully exploiting the possibilities of dynamic typing, e.g.

```plaintext
f(p)
{
    var x;
    x = "abc";
    if (p) x := 3;
    ...
    // here x can hold a string or an int.
    if (p) g(x); else h(x);
}
```
When variables change type (3)

A similar situation happens in Floyd-Hoare logic with judgements of the form \( \vdash \{P\} C \{Q\} \) where \( C \) is a command – the values of a variable are constrained by \( P \) before \( C \) and then by \( Q \) afterwards.

Note particularly the if-then-else rule

\[
(\text{COND}) \quad \vdash \{P \land e\} C \{Q\} \quad \vdash \{P \land \neg e\} C' \{Q\} \quad \vdash \{P\} \text{if } e \text{ then } C \text{ else } C' \{Q\}
\]

and consider cases with \( e \) being \( 10 < x \) and \( P \) being \( x < 20 \), i.e. on entry to the if-then else we have \( 10 < x \) and on entry to \( C \) we have \( 10 < x < 20 \).

Monadic predicates like \( P \equiv (x > 10) \) are logically types, so the type of \( x \) effectively changes during control flow.
Aside: Floyd-Hoare rules

Syntax:

\[ C ::= x := e \mid C; C' \mid \text{if } e \text{ then } C \text{ else } C' \mid \text{while } e \text{ do } C \]

Floyd-Hoare (partial correctness) rules:

\[
\begin{align*}
(\text{ASS}) & \quad \vdash \{Q[e/x]\}x := e\{Q\} \\
(\text{SEQ}) & \quad \vdash \{P\}C\{Q\} \quad \vdash \{Q\}C'\{R\} \\
(\text{COND}) & \quad \vdash \{P \land e\}C\{Q\} \quad \vdash \{P \land \neg e\}C'\{Q\} \\
(\text{WHILE}) & \quad \vdash \{P \land e\}C\{P\} \\
(\text{PREPOST}) & \quad \vdash \{P\}C\{Q\} \quad \vdash \{P'\}C\{Q'\} \quad \text{provided } P' \Rightarrow P \land Q \Rightarrow Q'
\end{align*}
\]
Aside: Floyd-Hoare is a type-like system

It may not appear so instantly, but writing

$$\vdash \{P\}C\{Q\}$$

as a syntactic variant (but having the same meaning)

$$\vdash C : [P, Q]$$

makes this more obvious.
Aside: can we do Floyd-Hoare inference?

The Floyd-Hoare rules are similar to type-rules – they are syntax-directed with the exception of the pre-post rule:

\[(\text{PREPOST}) \vdash \{P\}C\{Q\} \quad \text{provided } P' \Rightarrow P \land Q \Rightarrow Q'\]

So, can we do inference for this system like for type systems? Answer: no, but ...

Provided we are told the invariant \(P\) which holds for each WHILE loop (compare the idea of “anchoring a type-like inference”) then there is a most-general ascription of properties to each command. (See ‘weakest precondition’, ‘strongest postcondition’ and ‘verification condition generation’.)
When variables really change type (4)

One might be tempted to think: “if I don’t use simple assignment then variables do not change type”.

However, storage allocation provides a justification to have extended type-like properties which do change when a variable is merely read:

```plaintext
let x = new() in
    x.f = 1; x.g = 2; // legal
    free(x) // only a read of x itself
    x.g = 3; // illegal
...
```

The bit-pattern in x did not change, but we can see its type changing from “pointer to record” to “pointer to unallocated store”. Let’s see how variables may be made “legal on first-use only” (more later).
Standard type system

\( \text{VAR} \quad \Gamma[x : t] \vdash x : t \)

\( \text{INT} \quad \Gamma \vdash n : \text{int} \)

\( \text{BOOL} \quad \Gamma \vdash b : \text{bool} \)

\( \text{LAM} \quad \Gamma[x : t] \vdash e : t' \quad \Gamma \vdash \lambda x. e : t \to t' \)

\( \text{APP} \quad \Gamma \vdash e_1 : t \to t' \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_1 \ e_2 : t' \)

\( \text{COND} \quad \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t \quad \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \)

We’ve seen this system before.
Equivalent type system

Type assumptions $\Gamma, \Delta$ are now multi-sets (i.e. permutable lists):

(VAR) $[x : t] \vdash x : t$

(INT) $[] \vdash n : int$

(BOOL) $[] \vdash b : bool$

(LAM) $\Gamma, [x : t] \vdash e : t' \quad \frac{}{\Gamma \vdash \lambda x.e : t \rightarrow t'}$

(APP) $\frac{}{\Gamma \vdash e_1 : t \rightarrow t' \quad \Delta \vdash e_2 : t \quad \Gamma, \Delta \vdash e_1 e_2 : t'}$

(COND) $\frac{}{\Gamma \vdash e_1 : bool \quad \Delta \vdash e_2 : t \quad \Delta \vdash e_3 : t \quad \Gamma, \Delta \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$

(WEAKEN) $\frac{}{\Gamma \vdash e : t \quad \Gamma, \Delta \vdash e : t}$

(CONTRACT) $\frac{}{\Gamma, \Delta, \Delta \vdash e : t \quad \Gamma, \Delta \vdash e : t}$

What’s the point?

Answer: substructural type systems.
Substructural type systems

(WEAKEN) and (CONTRACT) are known as structural rules.

\[
\frac{\Gamma \vdash e : t}{\Gamma, \Delta \vdash e : t} \quad \text{(WEAKEN)}
\]

\[
\frac{\Gamma, \Delta \vdash e : t}{\Gamma, \Delta, \Delta \vdash e : t} \quad \text{(CONTRACT)}
\]

They control how the assumptions \( \Gamma \) are can be passed within an inference tree. [There’s a similarity between combinators \( S \) and \( K \) with (CONTRACT) and (WEAKEN) respectively.]

Using both WEAKEN and CONTRACT gives a system equivalent to the previous one.

Without CONTRACT every variable must be used at most once.

Without WEAKEN every variable must be used at least once.

Without either (although note we still allow permutation of \( \Gamma \) by treating it as a set) we get a pure linear type system.

[There are richer systems when pairing is added.]
Separation Logic

This generally means an richer form of Floyd-Hoare logic in which there is an additional binary logical connective \( * \) [actually a binary modality]. \( P * Q \) informally holds when \( P \) and \( Q \) both hold and no locations are shared between their formula (i.e. substructural).

Central is the fact that following Floyd-Hoare rule for concurrent execution is unsound:

\[
(PARBAD) \quad \frac{\{P\}C\{Q\} \vdash \{P'\}C'\{Q'\}}{\vdash \{P \land P'\}C||C'\{R \land Q'\}}
\]

(consider \( C \) and \( C' \) both updating variable \( x \)), but the separation-logic formula

\[
(PAR) \quad \frac{\{P\}C\{Q\} \vdash \{P'\}C'\{Q'\}}{\vdash \{P * P'\}C||C'\{R * Q'\}}
\]

is sound.
Separation Logic (2)

Separation logic is a form of sub-structural reasoning. Rules such as

\[
\Gamma \vdash e \quad \Delta \vdash e' \\
\frac{}{[x \leftrightarrow y] \vdash e \, e'}
\]

cannot (if they are to be sound) use assumption \([x \leftrightarrow y]\) that \(x\) points to \(y\) in both \(\Gamma\) and \(\Delta\).
To model single-writer multiple-reader protocols some authors allow *fractional ownership*

When a task has gathered a permission of 1.0 it gets read/write access, but this permission can be split and passed to other tasks. A permission of $0 < x < 1$ gives read-only permission.

We can then share read-only access to $x$:

\[
[x \xrightarrow{p} y] \vdash e \quad [x \xrightarrow{q} y] \vdash e'
\]

\[
\frac{}{[x \xrightarrow{p+q} y] \vdash e \; e'}
\]

So, when aliases exist updates are forbidden. (Note this single-writer/multiple-reader usage gives best cache performance on multicore.)
Lecture Conclusion

What have we done?

• identified *distant* aliases to *our* data as a problem.

• two incomparable solutions
  – ownership types: allow many pointers but only from “approved places”.
  – substructural types: each use of a variable changes the type for subsequent uses, e.g. linear types (only one pointer to each object).

• described a scheme (effect systems) which can express run-time side-effects within a type system.

Next lecture: a simple practical substructural type system for multi-core processor with no shared memory.
Lecture 4: Practical solutions

Alias Analysis and Parallelisation

Kilim, PacLang.
Surely we can just parallelise?

[The “dusty deck” problem.]
Consider an MP3 player containing code:

```c
for (channel = 0; channel < 2; channel++)
  process_audio(channel);
```

or even

```c
process_audio_left();
process_audio_right();
```

This can only be parallelised if neither call writes to a variable read or written by the other.
Can we know what variables are read/written?

A telling example (due to Paul Kelly)

\[
\text{for } (i = 0; i < n; i++) v[i]->field++; 
\]

Can this be parallelised? Depends on knowing that each cell of \( v[] \) points to a distinct object.

Want techniques to show this ("alias analysis").
Can we know what variables are read/written?

To some extent, but (in increasing difficulty):

- we can treat a simple variable as accessed in a procedure if it is accessed in *some* control flow path.

- but what about arrays? [may need arithmetic properties on subscript-expressions – big area, see polyhedral analysis and Presburger arithmetic.]

- but what about pointers? The OO style says “every object is a pointer”. Far from easy. We’ll look at a couple of techniques.

The important idea is “given two memory references, *may* they access the same memory location?”.
Alias analysis – some approaches

- do a “points-to” analysis [Andersen 1994] which associates each variable with (a description of) a set of locations.

- can now just say “\(x\) and \(y\) may alias if their results from points-to analysis is not provably disjoint”.

- shape analysis (Sagiv, Wilhelm, Reps) – a program analysis with elements being abstract heap nodes (representing a family of real-world heap notes) and edges between them being \(must\) or \(may\) point-to. Nodes are labelled with variables and fields which may point to them.

These techniques can become very expensive for large programs “alias analysis is undecidable in theory and intractable in practice”. Simpler techniques tend to say “I don’t know” too often.
Goto considered harmful revisited

Why did Dijkstra say this 40 years ago?

- because programmers constructed spaghetti code (the control flow was hard for humans to read, but not for machines!) using goto

- solution: structured programming techniques – while-do, if-then-else exceptions and the like.

By analogy, one would expect “structured object techniques” to be now the norm (so that the heap was not just a spaghetti-like collection of pointers)?

Unrestricted pointers considered harmful!
Consequences of purely linear types

In a purely linear type system every path must reference every variable exactly once. In practice too strong – need readonly access too:

\[
\text{f(Obj x) \{ if (x.field) < 5) return x; else \{ dispose(x); return new Obj(6); \}}
\]

Beware: there also is a subtlety concerning partial and total correctness. Suppose the function call \text{loop()} loops forever. But does \text{h()} reference \text{x} and \text{y} linearly?

\[
\text{h(x,y) \{ if (x) \{ print y; loop(); \}}
\]
\[
\text{else \{ loop(); print y; \}}
\]

But this system is still clumsy. How can we pass a value to two functions?
Consequences of purely linear types (2)

How can we pass a value to two functions?

**Answer 1:** we’re all heroic programmers! Re-write

\[
g(x,y) = \text{if } (x.m > y) \text{ then } 0 \text{ else } 1 \\
f(x) = g(x,10) + g(x,20)
\]

into

\[
g(x,y) = \text{if } (x.m > y) \text{ then } (0,x) \text{ else } (1,x) \\
f(x) = \text{let } (r,x’) = g(x,10) \text{ in } r + g(x’,20)
\]

**Answer 2:** Kobayashi’s quasi-linear types, OO community’s ‘borrowed’ objects: A linear pointer value is allowed to be

(i) passed in to a function which promises not to hold onto it;

(ii) used (consumed) afterwards.
What does linearity (or quasi-linearity) buy?

- Pointer control
- Implementation either by copy or by reference – perfect for multi-core with non-uniform memory. Equivalently linear message passing is equivalent to shared memory with linear access.
A non-cache-coherent NUMA case study

We explored the ideas of controlling data distribution using types for Intel’s IXDP ‘network processor’ engine.

- Multi-core, fastest memory is only accessible on a per-processor basis.
- Slower shared memory.
- Faster hardware queues for message passing.
Simplified diagram of IXDP2400 NP evaluation board.

Receive packets from network

Transmit packets to network
It is easy to program?

• No. Tools give hardware-like timing diagrams over where your data is!

• Need to be both hardware and software expert (not to mention protocol stacks) – expensive. Much programming is done at assembly level.

• How do I write portable code which runs in the IXP2400 and a different network processor?

• Key Intel remark:
  “In most designs processing of a packet by multiple threads simultaneously is rare, but using multiple threads to process packets at different points in the packet’s lifetime is common.” (Locking is expensive.)
Strategy: types to control memory use

- Formalise a concurrent, first-order, C-like imperative language; claim it sufficiently expressive and architecture neutral.

- Define a simple, intuitive and usable linear type system; restricted aliases within a thread are allowed (Algol-like pass into functions but not out of functions); no aliases allowed between threads.

- Paper summarises an IPv4 router; claim that C network programmers can understand it.
Expressions (note ANF form):

\[
\begin{align*}
v & \leftarrow i \mid b \mid x \\
e & \leftarrow \text{if } v \text{ then } e_1 \text{ else } e_2 \\
     & \quad \mid \text{let } (x_1, \ldots, x_j) = f(v_1, \ldots, v_k) \text{ in } e \quad (j, k \geq 0) \\
     & \quad \mid \text{return } (v_1, \ldots, v_k) \quad (k \geq 0) \\
d & \leftarrow (\sigma_1, \ldots, \sigma_j) f(\rho_1 x_1, \ldots, \rho_k x_k) \{e\} \quad (j, k \geq 0) \\
     & \quad \mid \text{packetQueue } q; \\
p & \leftarrow d_1 \ldots d_n
\end{align*}
\]

Types:

\[
\rho, \sigma, \tau \leftarrow \text{int} \mid \text{bool} \mid \text{packet} \mid \text{!packet}
\]
What do types mean?

- *packet*: owner of packet value (1st class)
- *!packet*: alias of packet value (2nd class) ['borrowed’ pointer]

So, given:

```c
extern void g(packet), h(!packet);
```

- `test1(packet p) { g(p) }` is OK
- `test2(!packet p) { g(p) }` is not OK
- `test3(packet p) { h(p) }` is not OK
- `test4(packet p) { h(p); g(p) }` is OK
- `test4'(packet p) { let _ = h(p) in g(p) }` is OK
- `test5'(!packet p) { let _ = h(p) in h(p) }` is OK
What do types mean (2)?

Given:

```c
extern void r1(!packet, !packet), r2(packet, !packet);
```

- `test6(!packet p) { r1(p,p) }` is OK
- `test7(packet p) { r2(p,p) }` is not OK
- `test8(packet p, packet q) { r2(p,q); kill(q) }` is OK
Type operators (for linearity)

\[
\begin{align*}
\text{int} + \text{int} &= \text{int} \\
\text{bool} + \text{bool} &= \text{bool} \\
\text{!packet} + \text{!packet} &= \text{!packet}
\end{align*}
\]

\[
\begin{align*}
\text{int} ; \text{int} &= \text{int} \\
\text{bool} ; \text{bool} &= \text{bool} \\
\text{!packet} ; \text{!packet} &= \text{!packet} \\
\text{!packet} ; \text{packet} &= \text{packet}
\end{align*}
\]

- ‘+’ models concurrent use (e.g. \text{let ... = f(x,x) in ...})
- ‘;’ models sequential use (e.g. \text{let ... = f(x) in g(x)})
- Extend to environments pointwise.
Linear typing rules

\[
\begin{align*}
\{ x : \tau \} \vdash x : \tau & \quad \emptyset \vdash i : \text{int} & \quad \emptyset \vdash b : \text{bool} \\
\Gamma_1 \vdash v_1 : \rho_1 & \ldots & \Gamma_k \vdash v_k : \rho_k & \quad \Gamma_0, x_1 : \sigma_1, \ldots, x_j : \sigma_j \vdash e : \vec{\tau} \\
(\Gamma_1 + \ldots + \Gamma_k); \Gamma_0 \vdash \text{let} (x_1, \ldots, x_j) = f(v_1, \ldots, v_k) \text{ in } e : \vec{\tau} & \quad \text{provided } \mathcal{F}(f) = (\rho_1 \times \ldots \times \rho_k) \rightarrow (\sigma_1 \times \ldots \times \sigma_j) \\
\Gamma_1 \vdash v_1 : \tau_1 & \ldots & \Gamma_j \vdash v_j : \tau_j & \quad \Gamma_1 + \ldots + \Gamma_j \vdash \text{return} (v_1, \ldots, v_j) : \tau_1 \times \ldots \times \tau_j & \quad \text{provided no } \tau_j \text{ is } !\text{packet} \\
\Gamma_1 \vdash v : \text{bool} & \quad \Gamma_2 \vdash e_1 : \vec{\tau} & \quad \Gamma_2 \vdash e_2 : \vec{\tau} & \quad \Gamma \vdash e : \vec{\tau} & \quad \Gamma, x : \rho \vdash e : \vec{\tau} & \quad \text{provided } \rho \text{ is not } !\text{packet}
\end{align*}
\]
Linear typing rules (2)

\[
\begin{array}{c}
\{x_1: \rho_1, \ldots, x_k: \rho_k\} \vdash e: \sigma_1 \times \ldots \times \sigma_j \\
\vdash (\sigma_1, \ldots, \sigma_j) f(\rho_1 x_1, \ldots, \rho_k x_k)\{e\} \\
\vdash d_1 \ldots \vdash d_n \\
\hline
\vdash d_1 \ldots d_n \\
\end{array}
\]

\vdash packetQueue q;
Built-ins

int ⟨arith-op⟩(int x, int y); bool ⟨bool-op⟩(bool x, bool y);
bool ⟨rel-op⟩(int x, int y);
packet new(int size); void kill(packet p);
packet q.deq(); void q.enq(packet p);

int read(!packet p, int offset);
void update(!packet p, int offset, int value);
Examples

Valid:

let packet x = new(10) in
let () = update(x,0,5) in
let () = update(x,1,10) in
let () = transmit.enq(x) in return ()

Invalid:

let packet x = new(10) in
let () = kill(x) in
let () = transmit.enq(x) in return ()
Chemical Abstract Machine Semantic Values

Molecules take the following forms:

\[ M \leftarrow \left\{ \{ \vec{x} \} e \right\}_f \]  (function definition)

\[ \langle \langle Q \rangle \rangle_q \]  (queue)

\[ \langle H \rangle \]  (shared heap)

\[ \langle e, \Sigma \rangle \]  (thread: expression and its stack frames)

Initial state \( \text{init}(P) \) consists of molecules as below:

- \( \left\{ \{ \vec{x} \} e \right\}_f \) for each definition \( f(\vec{x})\{e\} \)
- an empty queue \( \langle \langle e \rangle \rangle_q \) for each queue definition \( \text{packetQueue} q \)
- an empty heap \( \langle \rangle \)
- a thread \( \langle e_j, \epsilon \rangle \), for each definition \( \text{main}_j()\{e_j\} \)
Operational Semantics for PacLang

Molecule reactions (reaction in context)

\[
\frac{M_1 \mid \ldots \mid M_n \leadsto M'_1 \mid \ldots \mid M'_n}{\Delta_1 \mid \ldots \mid \Delta_k \mid M_1 \mid \ldots \mid M_n \rightarrow \Delta_1 \mid \ldots \mid \Delta_k \mid M'_1 \mid \ldots \mid M'_n}
\]

Boring: here all reactions involve a thread and a non-thread.

Core language primitives:

\[
[\{\bar{x}\}e_1]_f \mid (\text{let } \bar{y} = f(\bar{v}) \text{ in } e_2, \Sigma) \leadsto [\{\bar{x}\}e_1]_f \mid (e_1\{\bar{v}/\bar{x}\}, (\{\bar{y}\}e_2) \bullet \Sigma)
\]

\[
(\text{let } x = v_1 \text{ op } v_2 \text{ in } e, \Sigma) \leadsto (e\{v/x\}, \Sigma) \text{ where } v = v_1 \text{ op } v_2
\]

\[
(\text{return } \bar{v}, (\{\bar{x}\}e) \bullet \Sigma) \leadsto (e\{\bar{v}/\bar{x}\}, \Sigma)
\]

\[
(\text{if } \text{true } \text{then } e_1 \text{ else } e_2, \Sigma) \leadsto (e_1, \Sigma)
\]

\[
(\text{if } \text{false } \text{then } e_1 \text{ else } e_2, \Sigma) \leadsto (e_2, \Sigma)
\]
Operational Semantics for PacLang (2)

Packet and Queue manipulation:

\[ \langle H \rangle | (\text{let } x = \text{new}(i) \text{ in } e, \Sigma) \leadsto \langle H[\alpha \mapsto \text{empty}_i] \rangle | (e\{\alpha/x\}, \Sigma) \]

where \( \alpha \) is a fresh name

\[ \langle H \rangle | (\text{let } () = \text{kill}(\alpha) \text{ in } e, \Sigma) \leadsto \langle H\backslash \alpha \rangle | (e, \Sigma) \]

\[ \langle H \rangle | (\text{let } x = \text{read}(\alpha, i) \text{ in } e, \Sigma) \leadsto \langle H \rangle | (e\{H(\alpha)(i)/x\}, \Sigma) \]

\[ \langle H \rangle | (\text{let } () = \text{update}(\alpha, i, v) \text{ in } e, \Sigma) \leadsto \langle H[\alpha \mapsto H(\alpha)[i \mapsto v]] \rangle | (e, \Sigma) \]

\[ \langle \alpha \bullet Q \rangle_q | (\text{let } x = q.\text{deq}() \text{ in } e, \Sigma) \leadsto \langle Q \rangle_q | (e\{\alpha/x\}, \Sigma) \]

\[ \langle Q \rangle_q | (\text{let } () = q.\text{enq}(\alpha) \text{ in } e, \Sigma) \leadsto \langle Q \bullet \alpha \rangle_q | (e, \Sigma) \]
Unique Ownership

Unique Ownership Property:

- Every packet in the heap is always referenced by exactly one thread

This means

- No space leaks

- Can use call-by-value or call-by-reference in enq/deq.
Unique Ownership – formally

- \( pr(M) = \) the set of packet references, \( \alpha \), occurring in molecule \( M \).

- \( \text{UniqueOwn}(M_1 | \ldots | M_n | \langle H \rangle) \iff pr(M_1) \uplus \ldots \uplus pr(M_n) = \text{dom } H \)

- If program \( P \) is well-typed then every state, \( W \), encountered during execution of \( P \) satisfies unique ownership:

  \[ \vdash^P P \land \text{init}(P) \rightarrow^* W \Rightarrow \text{UniqueOwn}(W) \]

  Note how this is a form of “preservation and progress”.

- In English: every heap allocated packet is owned by exactly one process or queue and this packet reference does not get ‘lost’.
PacLang – review

Why did we make the design choices?

- Wanted a C-like language; PacLang could be seen as a desugared version.

- avoided higher-order features – not trivial and only special cases wanted for the C heritage.

- chose a semantic model (Chemical abstract machine and a substitution semantics) which made the statement and proof of unique ownership easy. [Think about the issues involved with “reference $\alpha$ is bound to variables in two processes, but only one of these variables is live . . . ”.]
Extending PacLang to structured data

PacLang only handles allocation and deallocation of packet-sized buffers.

Wanted a language which could:

- provide Actor-like (Erlang) isolation: each process has its own address space
- enable structured data to be transferred between actors.
- (less important here) be Java compatible.

Solution: Kilim [ECOOP’2008].
Kilim and its type system

Problematic issue:

```c
f() { List x = Cons(1, Nil);
    List y = Cons(2, x);
    send(x);
    ... // what is the status of y here?
}
```

Solution: identify a class of values which are transmissible.

- these have no internal sharing – only tree structure
- they may have multiple stack references into them, but when part or all of them is transferred to another process then the types of all such the stack references change.
Kilim and its type system (2)

Identified hardware-efficient-relevant types (‘isolation types’):

- “passing full access to another process”
- “passing a read-only copy”

Mental model: (ccNUMA) multicore is like multiple-reader single-writer –

- caches allow occasional change of read/write ownership
- caches allow efficient multiple readers when no writer
Kilim – core calculus syntax

(All implemented via marker interface Message in Java.)

\[
\text{FuncDcl} ::= \text{free}_\text{opt} \ m(\vec{p} : \vec{\alpha}) \ \{ \ (lb : \text{Stmt})* ; \ \}\n\]

\[
\text{Stmt} ::= x ::= \text{new} \quad | \quad x ::= y \\
| \quad x ::= y.f \quad | \quad x.f ::= y \quad | \quad x ::= \text{cut}(y.f) \\
| \quad x ::= y[.] \quad | \quad x[.] ::= y \quad | \quad x ::= \text{cut}(y[.]) \\
| \quad x ::= m(\vec{y}) \quad | \quad \text{if/goto } \vec{lb} \quad | \quad \text{return } x
\]

\(x, y, p \in \text{variable names} \quad f \in \text{field names}\)

\(lb \in \text{label names} \quad m \in \text{function names}\)

\(sel \in \text{field names} \cup \{[.]\} \quad [.\] \text{pseudo field name for array access}\)

\(\alpha, \beta \in \text{isolation qualifier} \{\text{free, cuttable, safe}\}\)

\(\text{null} \text{ is treated as a special readonly variable}\)
Kilim isolation qualifiers

**free:** A *free* object is guaranteed to be a tree root (no heap pointers to it), but not vice-versa. Sources: *new*, using *cut* on a *cuttable* or *free* object, and a method parameter marked as *@free*.

**cuttable:** Object is not necessarily a root, but it can be *cut* (a *free* subtree taken from it and nulled out). Sources: field access to a *cuttable* or *free* object, downgrading a *free* object by assigning it to a field of another (it is no longer a root), and a method parameter marked *@cuttable*.

**safe:** The *safe* capability is granted to a method parameter marked *@safe* or (transitively) to any object reachable from it. A *safe* object may not be structurally modified or further heap-aliased or *sent* to another actor, but non-pointer fields can be modified.
The **cut** operator severs a subtree from its (*cuttable*) parent thus:

\[
y = \text{cut}(x.\text{sel}) \quad \overset{\text{def}}{=} \quad y = x.\text{sel}; x.\text{sel} = \text{null};
\]

Crucially it also marks \( y \) as *free* – the right-hand-side above would not do this (especially on arrays). Merely doing \( y = x.\text{sel} \) would merely leave \( y \) as *cuttable*, the \( x.\text{sel} = \text{null} \) makes good the semantic promise that \( x \) has no pointers to it.
Lecture Conclusion

What have we done?

- Looked at two implemented systems using linear types to control aliasing, so that datastructures (packets in PacLang, whole trees in Kilim) have a single (live) pointer to them at critical points.

- Note that “single pointer” with “no use afterwards by caller” implies call-by-reference and call-by-value are equivalent. This also means:

  - RMI/RPC (remote method invocation) is semantically equivalent to a local call; and

  - using shared memory is equivalent to message passing for such data (reminder: multicore CPUs simulate shared memory in the cache by using message passing in cache-coherence protocols).
Course Conclusion

- This course has been somewhat of a world tour, visiting places where compile-time invariants are necessary, but have been neglected in the past.

- Multicore hardware (its effective use, and even its correct use) depends on knowing more about where pointers point.

- Can be seen as the feature interaction of complex sharing structures with mutability further complicated by the complex multi-core feature interaction of concurrency and mutability.

Overall motto: “(unrestricted) pointers considered harmful!”