

# Extending Dylan's Type System for better Type Inference and Error Detection

hannes; Berlin



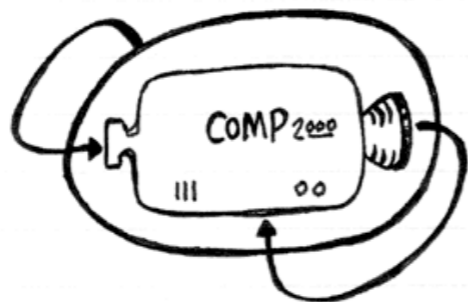
Hey Kids!!

It's

# COMPILER TIME!



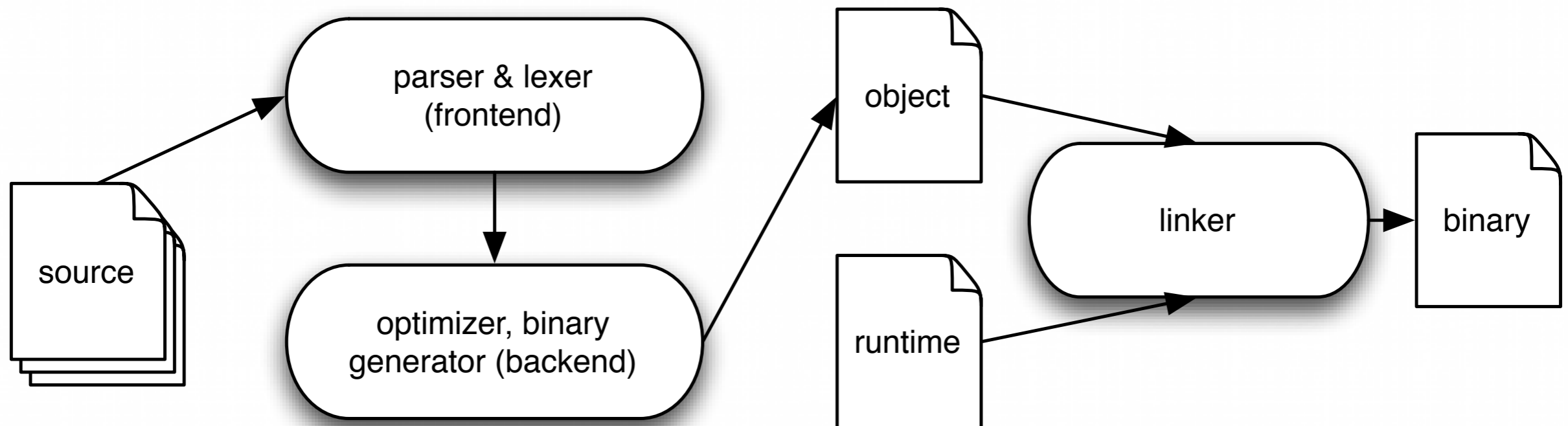
A **COMPILER** IS A PROGRAM THAT, WHEN FED ITSELF AS INPUT, PRODUCES ITSELF.



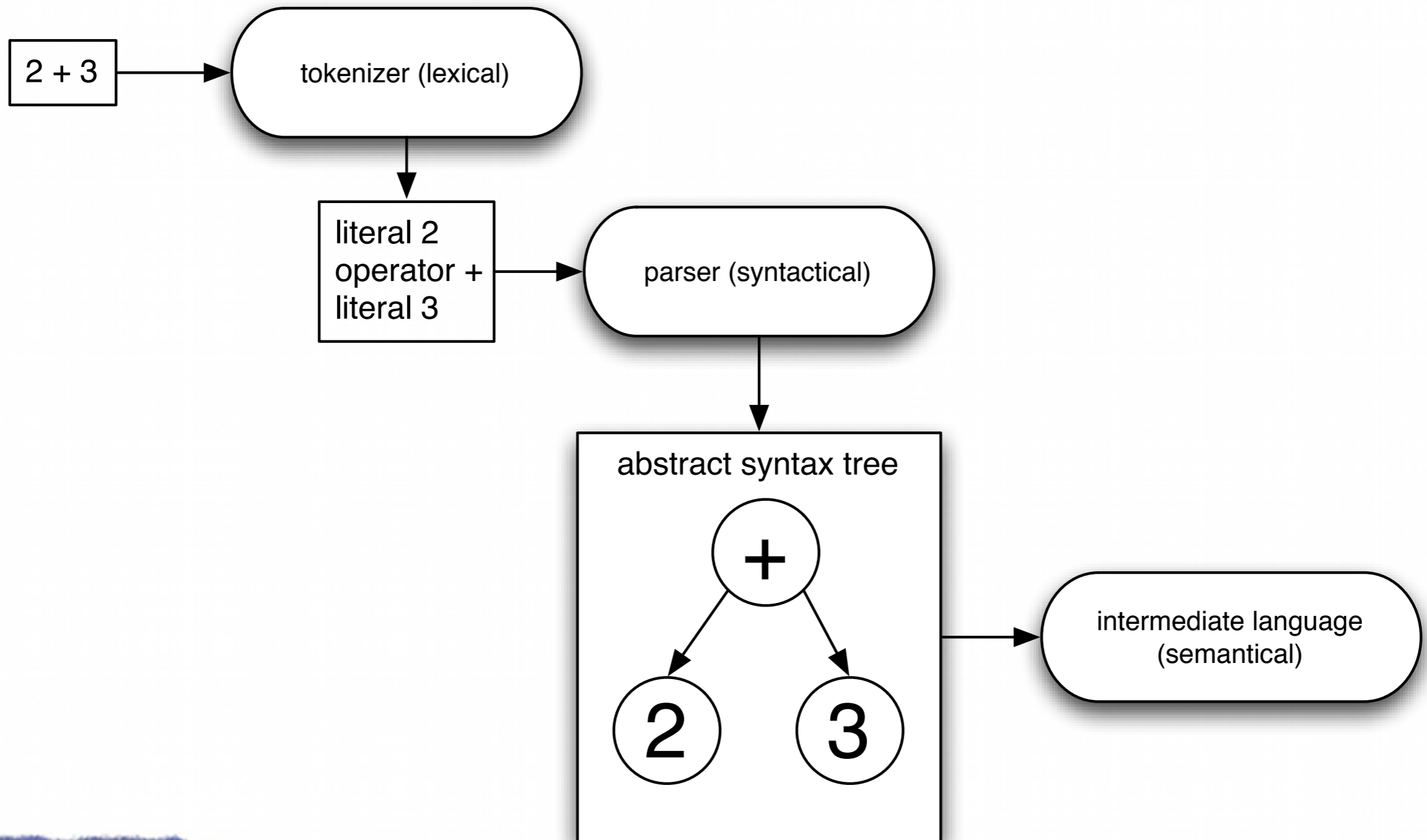
PEOPLE ARE PROGRAMS...



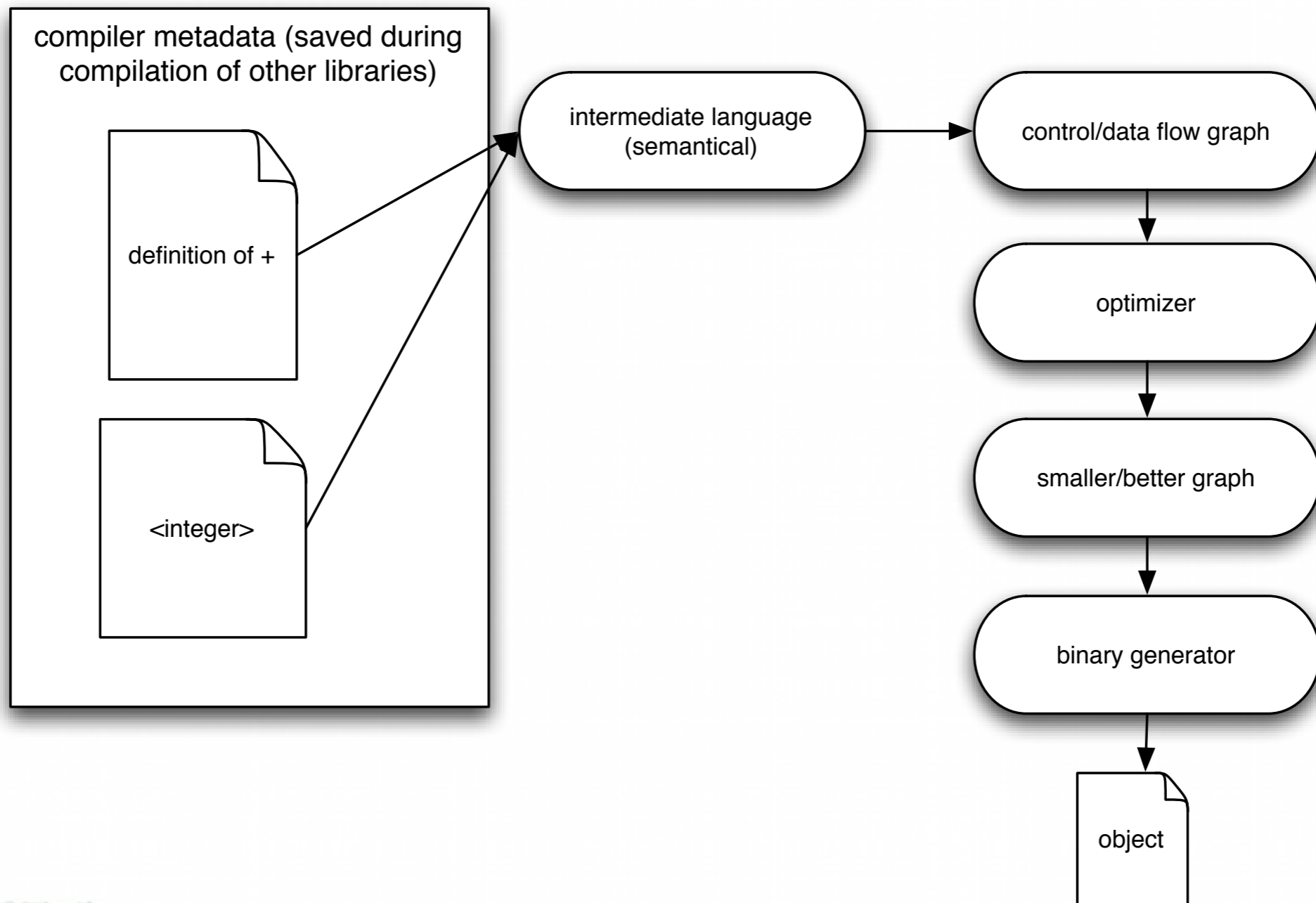
# Compiler overview



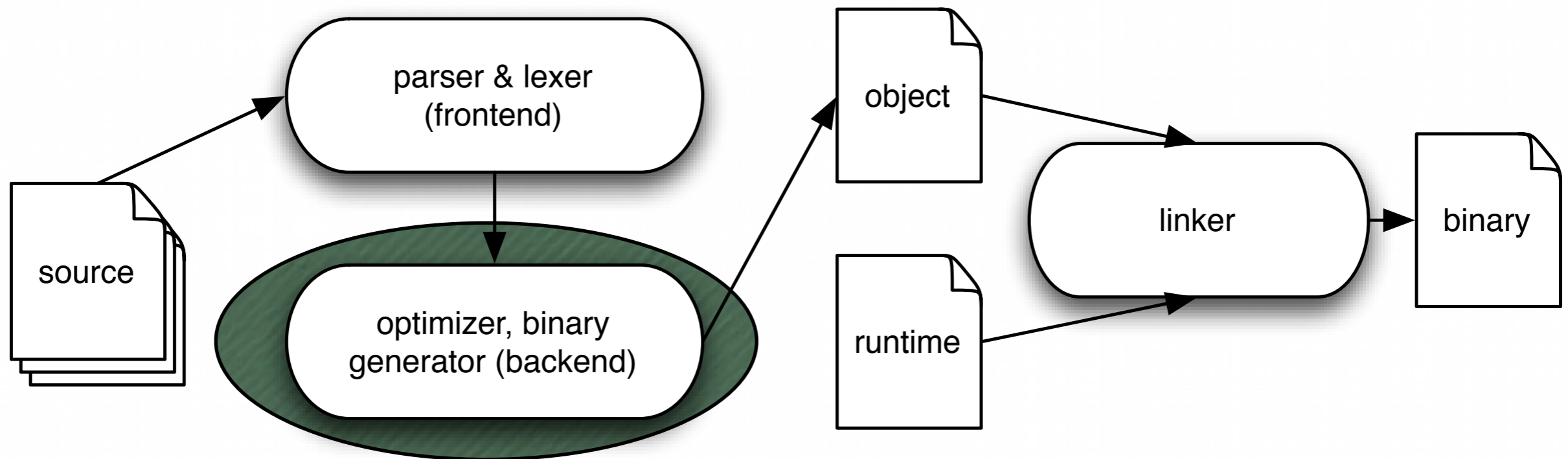
# Frontend - AST



# Backend



# Compiler



- Input: intermediate language
- Output: intermediate language

# Typing

- Types are information of variables at compile time
- Errors can be detected earlier (at compile time)
- Static typing: you've to write down types everywhere (including up/downcasts; Java, Haskell, C, C++)
  - Type information only available at compile time
- Dynamic typing: errors are detected at runtime; you don't have to write any type (Scheme, Lisp, Python, Ruby)
  - Type information at runtime (tag bits/boxed types)



# Dylan overview

- Object-oriented (class-based, mutable state)
- Algol style syntax (contrast to Lisp)
- Metaprogramming (DSL, macros)
- Strictly & dynamically typed (and optional type annotations)
- Classes and methods first-class objects
  - Generic functions, multiple dispatch, multiple return values
- Functional aspects (higher order functions)
- Separation of compile and runtime (contrast to Lisp)



# Intermediate language

- application (call)
- binding (temporary-transfer)
- abstraction (make-closure)
- loop
- if
- assignment
  
- check-type, multiple-value, slot-getter/-setter, unwind-protect, bind-exit, etc.



# Assignment - SSA

Problem: what type has a?

```
let a = 42;  
let b = a + a;  
a := "foo";  
let c = concatenate(a, "bar");
```

Former solution (cell)

```
let a = make(<cell>, type: <top>);  
a.value := 42;  
let b = a.value + a.value;  
a.value := "foo";  
let c = concatenate(a, "bar");
```

Single static assignment

```
let a0 = 42;  
let b = a0 + a0;  
let a1 = "foo";  
let c = concatenate(a1, "bar");
```



# Single Static Assignment

Original source

```
let a = 1;  
if (b == 2)  
  a := 2;  
else  
  a := 3;  
end;  
let c = a + a;
```

SSA converted:  
merge of variables after loop,  
if branches (with phi nodes)

```
let a0 = 1;  
if (b == 2)  
  let a1 = 2;  
else  
  let a2 = 3;  
end;  
let a3 = phi(a1, a2);  
let c = a3 + a3;
```

# Optimizations

- Inlining
- Constant folding
- Dead code removal
- Common subexpression elimination
- Tail call elimination
- Call upgrading (GF/polymorphic)

# Inlining

- If method known and small enough

```
define method double  
  (a :: <integer>)  
    2 * a  
end;
```

- would expect:
  - direct call to primitive \* with two integer values, "a" and "2"
- -> faster binary (fewer calls)

# Constant folding

- `let foobar = 42 * 42;`
- Computation side-effect free
- $\Rightarrow$  `let foobar = 1764`
- $\rightarrow$  faster (smaller binary, computations already done)

# Dead code removal

- if path known to be not taken
- -> smaller binary size

```
define method dead ()  
  if (#f)  
    23;  
  else  
    42;  
  end;  
end;
```

# Common subexpression elimination

```
define method cse  
  (a :: <integer>)  
    values(42 + a, (42 + a) * 2)  
end;
```

- compute subexpression only once
- side-effect free
- => let x = 42 + a; values(x, x \* 2)

# Tail call elimination

```
define method tail-call
(x :: <integer>)
  if (x == 0)
    1
  else
    tail-call(x - 1)
  end;
end;
```

- instead of call, jump
- thus, loop!
- space and time (register save/restore, stack frame)

# Call upgrade (GF)

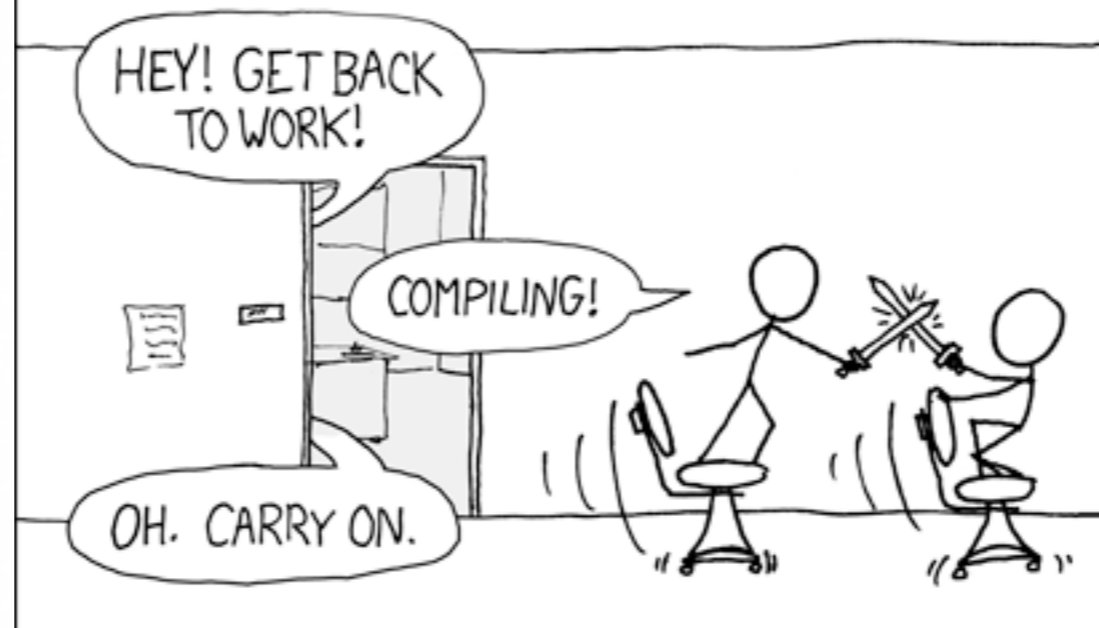
- choose direct method instead of generic function
- GF dispatch expensive at runtime (iterate through all methods of GF and compute distance, find most specific)

```
define method \+ (x :: <integer>, y :: <integer>)  
=> (result :: <integer>)  
  primitive-+(x, y);  
end;
```

```
define method \+ (a :: <string>, b :: <string>)  
=> (result :: <string>)  
  concatenate(a, b);  
end;
```

```
let foobar = "foo" + "bar";  
let bar = 42 + 42;
```

THE #1 PROGRAMMER EXCUSE  
FOR LEGITIMATELY SLACKING OFF:  
"MY CODE'S COMPILING."



<http://xkcd.com/303/>

# Type vs Class

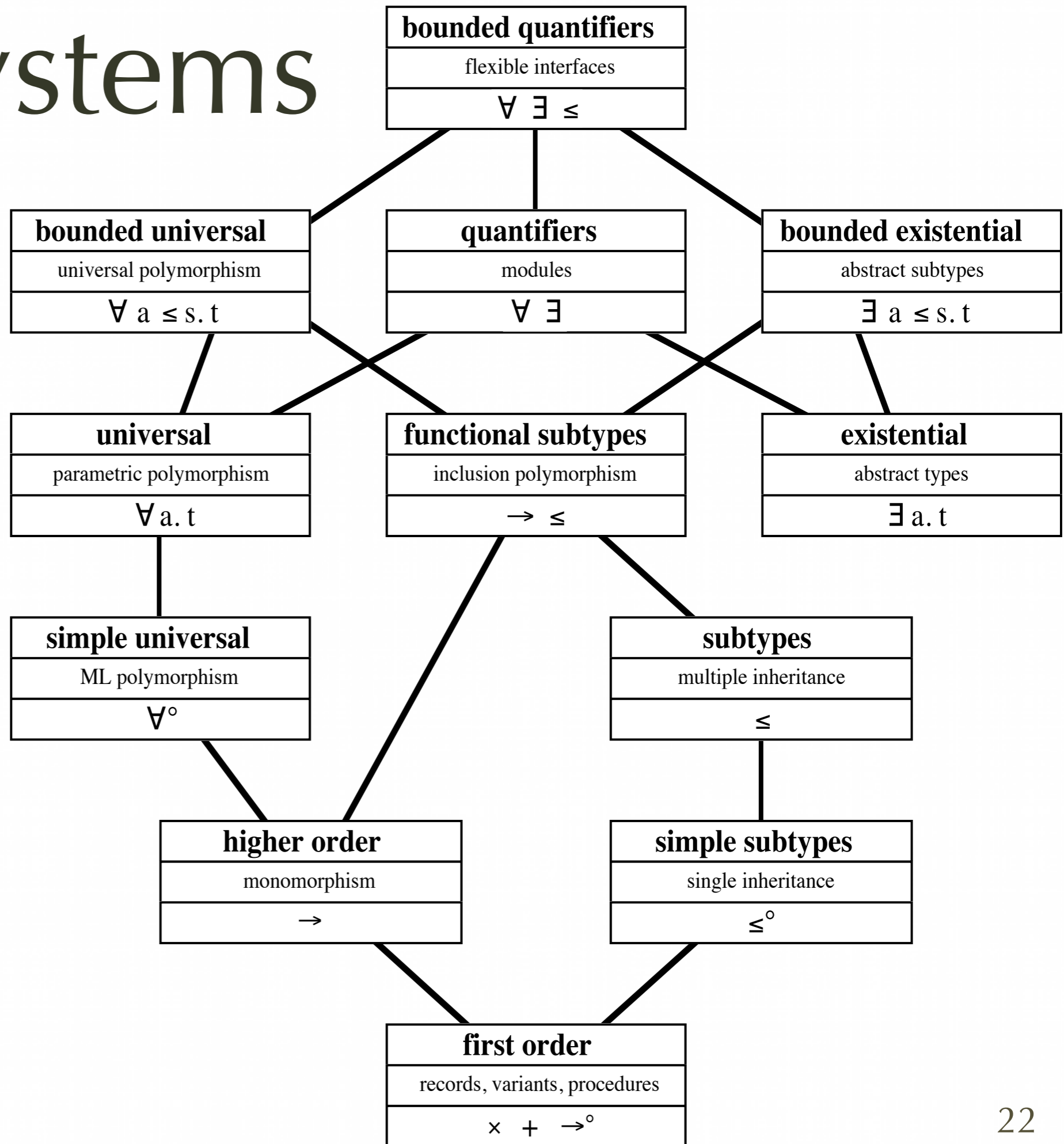
- Every class is a type (<integer>)
  - Data structure
- But not every type is a class (false-or(<integer>))
  - Arguments to algorithms
    - `element (key, table :: <table>) => (value :: false-or(<value>))`

# Types

- lambda calculus
  - abstraction, application, binding
- tuple, record types
- subtyping
- gradual typing
- [dependent types]

# Type systems

Cardelli, Wegner "On Understanding Types, Data Abstraction and Polymorphism"



# Gradual typing

- Integration of dynamic typing into formal type system
- Typed lambda calculus
- Dynamic type ?, type known at run time, not compile time

## Type Consistency

$$\boxed{\begin{array}{c} \frac{}{\gamma \sim \gamma} \quad \frac{}{\tau \sim ?} \quad \frac{}{? \sim \tau} \quad \frac{\tau_1 \sim \tau_3 \quad \tau_2 \sim \tau_4}{\tau_1 \rightarrow \tau_2 \sim \tau_3 \rightarrow \tau_4} \end{array}}$$

# Type inference input

- Type input:
  - Annotations: `let foo :: <integer> = bar();`
  - Calls: method signatures:
    - `define method bar () => (res :: <integer>)`
  - Literals: `let foobar = 42;`
  - Occurrence: `if (x == 42) x * x else "foo" end;`

# Type inference

based on Hindley-Milner, solution with Huet type graph

- Every data flow node has a correspondent type variable
- Constraint generation
- Solver
  - Least upper bound
  - Propagation of tuples, arrows to their children
- Finally: assignment for each type variable

# Huet Type Graphs

## Constraint Generation

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_g x : \tau | \{\}} \text{C-VAR}$$

$$\Gamma \vdash_g c : \text{typeof}(c) | \{\} \text{C-CNST}$$

$$\frac{\Gamma \vdash_g e_1 : \tau_1 | C_1 \quad \Gamma \vdash_g e_2 : \tau_2 | C_2 \quad (\beta \text{ fresh}) \quad C_3 = \{\tau_1 \simeq \tau_2 \rightarrow \beta\} \cup C_1 \cup C_2}{\Gamma \vdash_g e_1 e_2 : \beta | C_3} \text{C-APP}$$

$$\frac{\Gamma(x \mapsto \tau) \vdash_g e : \rho | C}{\Gamma \vdash_g \lambda x : \tau. e : \tau \rightarrow \rho | C} \text{C-ABS}$$

# Huet Type Graphs Solver

Input: Constraints  $C$

**while not**  $C.empty$

$(x \simeq y) := C.pop$

find representative nodes ( $u$  and  $v$ )

case  $stype(u) \simeq stype(v)$  of:

$u_1 \rightarrow u_2 \simeq v_1 \rightarrow v_2 \Rightarrow C.push(u_1, v_1); C.push(u_2, v_2)$

$u_1 \rightarrow u_2 \simeq ? \Rightarrow C.push(u_1, ?); C.push(u_2, ?)$

$\tau \simeq var \mid \tau \simeq ? \mid \gamma \simeq \gamma \Rightarrow pass$

$_ \Rightarrow error: inconsistent types$

$G =$  quotient graph by equivalence class

**if**  $G$  is acyclic

return  $\{ u \mapsto stype(u) \mid u \text{ a node in the graph } \}$

else error

# Dylan - Types

- Classes (multiple inheritance)
- Union (`false-or(<integer>)`)
- Singleton (`x == #"foo"`)
- Bounded quantification (collections, integers, classes)
  - vector of integer, integer between 0 and 16, `subclass(<number>)`
- Method signatures (also variable arity)
  - tuple type (required), record type (optional keyword arguments)
  - `=>` tuple type (required values)



# Dylan - Additional Types

- Polymorphic type variables (variable arity;  $\text{identity}(A)(x :: A) \Rightarrow (x :: A)$ )
- Bounded quantification (functions)
  - $\text{foo}(x :: \langle \text{integer} \rangle \Rightarrow \langle \text{string} \rangle) \Rightarrow (\text{result} :: \langle \text{string} \rangle)$
- Occurrence typing:
  - method returns a boolean, if true, the argument was of a given type
  - $\text{instance?}(O <: \langle \text{type} \rangle)(x, \text{type} :: O) = O \Rightarrow (\text{result} :: \langle \text{boolean} \rangle)$



# Dylan - Extensions for Type Inference

- Extensions for solver
  - Subtyping
  - Tuple types
  - Record types
  - Type variables
- Control flow nodes
  - If
  - Loop
  - Phi-node
  - Multiple value

# Subtyping

$$S <: S \text{ S-REFL}$$

- transitive, reflexive

$$\frac{S <: U \quad U <: T}{S <: T} \text{ S-TRANS}$$

- top type

$$S <: \top \text{ S-TOP}$$

- function type (co/contravariant)

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \text{ S-ARROW}$$

- singleton

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \text{ T-SUB}$$

- limited

- gradual type

$$? <: ? \text{ S-?}$$

# Solver Extensions

- Tuple type
  - Propagate to children, as arrow
  - If variable arity, generate new type nodes of specific type
- Type variables
  - Instantiate for each call site, then try to infer type variable
- Subtyping
  - Look whether one type is subtype of the other, use more specific

# Control Flow Extensions

- If
  - Taking occurrence typing into account
- Loop
  - Inferring loop-body with outer type of variables which get assigned (and loop variables)
  - If outer type equals inner type, use that
- Phi-node
  - Merge types of both branches

# Control Flow Extensions - Multiple Values

- Values
  - Transformation from input values into single multiple value; Tuple type of input values equals type of multiple value
- Extract single value
  - Extracted value type is equal to corresponding node of tuple type from multiple value

# Polymorphic to Monomorphic update

```
define method single-map (A, B)
  (fun :: A => B, l :: limited(<list>, of: A))
  => (result :: limited(<list>, of: B))
  if (l.empty?)
    #()
  else
    list(fun(l.head), single-map(fun, l.tail))
  end
end

single-map(method(x) x + 1 end, #(1, 2, 3))
```

# Conclusion & Further Work

- Implemented formal type system into dynamic programming language
- Gap between static and dynamic typing is getting smaller
- Global type inference for global variables and slots
- Integration of dependent types (remove bounds checks)
- Coercion semantics
- Higher-rank polymorphism



# References - Books

- Aho, Sethi, Ullman: **Compilers: principles, techniques, tools**; Prentice Hall 1986
- Shalit, Andrew: **Dylan Reference Manual**; Apple press, 1994
- Pierce, Benjamin C.: **Types and programming languages**; MIT press, 2002
- Pierce, Benjamin C. [Editor]: **Advanced topics in types and programming languages**; MIT press, 2005
- Muchnick, Steven S.: **Advanced compiler design & implementation**; Academic press, 1997



# References - Papers

- Siek, Jeremy et al: **Gradual Typing** (<http://ecee.colorado.edu/~siek/gradualtyping.html>)
  - Gradual typing for functional languages (Scheme Workshop 2006)
  - Gradual typing for Objects (ECOOP 2007)
  - Gradual typing with unification-based inference (DSL 2008)
  - Threesomes, With and Without Blame (STOP 2009)
- Tobin-Hochstadt, Sam et al: **Typed Scheme** (<http://www.ccs.neu.edu/home/samth/typed-scheme/>)
  - Well-typed programs can't be blamed (ESOP 2009, Scheme Workshop 2007)
  - The design and implementation of Typed Scheme (POPL 2008)
  - Variable-arity polymorphism (ESOP 2009)

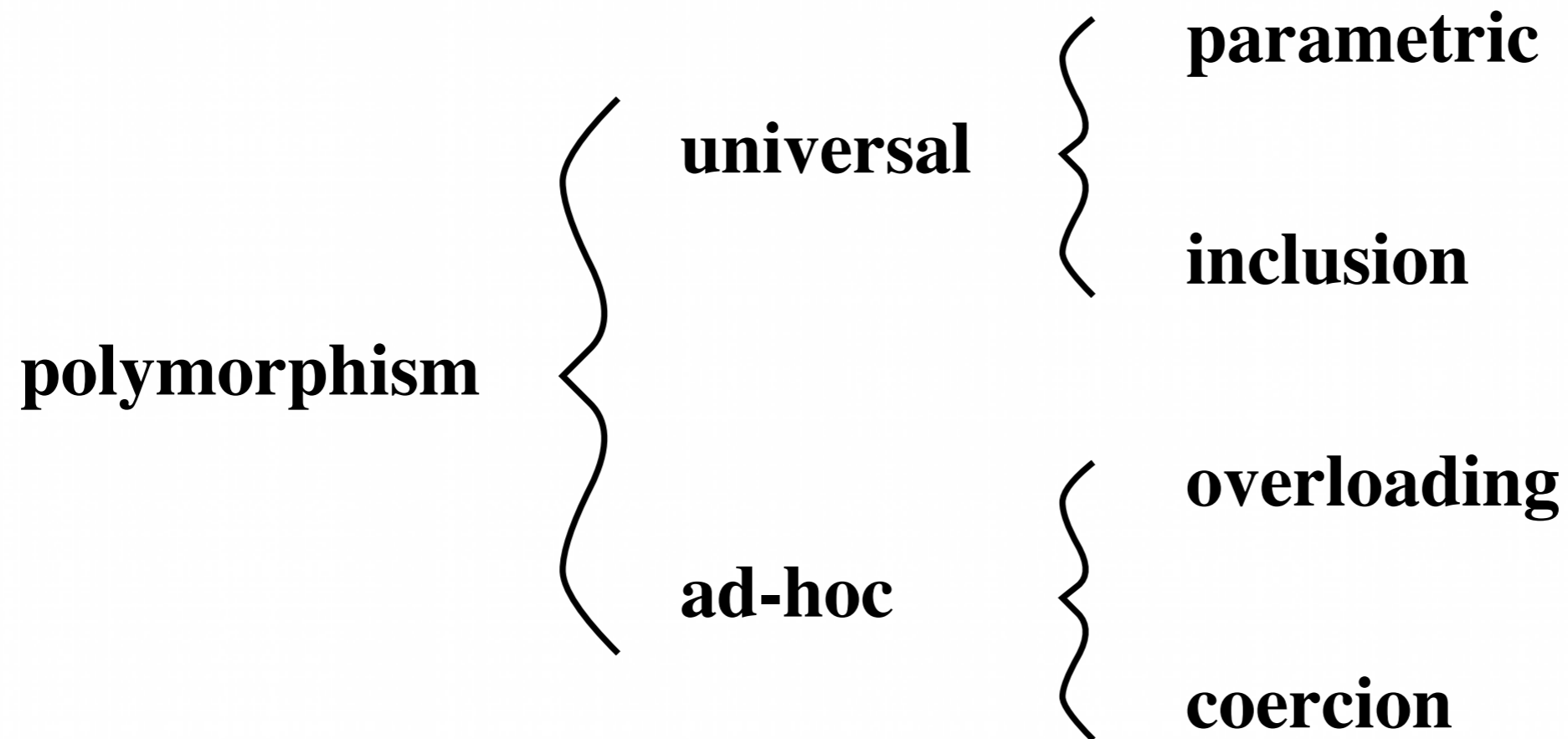


# Thank you! Questions?

- Visualization: <http://visualization.dylan-user.org/>
- Graph library (yFiles): <http://www.yworks.com/>
- Dylan: <http://www.opendylan.org>



# Polymorphism



Cardelli, Wegner "On Understanding Types, Data Abstraction and Polymorphism"