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

Hannes Mehnert
Department of compiler construction and programming languages
TU Berlin
Overview

• Dylan overview
• SSA conversion
• Gradual typing
• Huet based type inference
• Dylan types and type inference
• Conclusion and further work
Motivation

- More performance
- Feasibility
- Formal type system
Compiler

- Input: intermediate language
- Output: intermediate language
Dylan overview

- Object-oriented (class-based, mutable state)
- Algol style syntax
- 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
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 c = phi(a1, a2);
let a3 = phi(a1, a2);
let c = a3 + a3;
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**

\[
\begin{aligned}
\gamma & \sim \gamma & \tau & \sim ? & ? & \sim \tau \\
\tau_1 & \sim \tau_3 & \tau_2 & \sim \tau_4 \\
\tau_1 & \rightarrow \tau_2 & \sim \tau_3 & \rightarrow \tau_4
\end{aligned}
\]
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

\[ \Gamma(x) = \tau \]
\[ \Gamma \vdash_g x : \tau \mid \{\} \quad \text{C-VAR} \]

\[ \Gamma \vdash_g c : \text{typeof}(c) \mid \{\} \quad \text{C-CNST} \]

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

\[ \Gamma(x \mapsto \tau) \vdash_g e : \rho \mid C \]
\[ \Gamma \vdash_g \lambda x : \tau.e : \tau \rightarrow \rho \mid C \quad \text{C-ABS} \]
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:

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

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

$\tau \simeq \text{var} | \tau \simeq ? | \gamma \simeq \gamma \Rightarrow \text{pass}$

$_ \Rightarrow \text{error: inconsistent types}$

$G = \text{quotient graph by equivalence class}$

if $G$ is acyclic

return \{ $u \mapsto stype(u) \mid u$ 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; \( \text{identity}(A)(x :: A) => (x :: A) \))

- Bounded quantification (functions)
  - foo(x :: <integer> => <string>) => (result :: <string>)

- Occurrence typing:
  - method returns a boolean, if true, the argument was of a given type
  - instance? (O <: <type>)(x, type :: O) = O => (result :: <boolean>)
Subtyping

• **XXX**: subtyping relation for tuple, etc.

\[
S \vdash t : S \\
\Gamma \vdash t : T \\
\Gamma \vdash t : T
\]

\[
T_1 \vdash S_1 \quad S_2 \vdash T_2 \\
S_1 \rightarrow S_2 \vdash T_1 \rightarrow T_2
\]

\[
S <: U \\
U <: T \\
S <: T
\]

\[
S <: S \quad S <: U \\
S <: U \\
S <: T
\]

\[
S <: \top \\
S <: \top
\]

\[
S<:S \quad S<:U \\
S<:T \\
S<:T
\]

\[
? <: ? \\
S<:S
\]
Dylan - Extensions for Type Inference

- Extensions for solver
  - Subtyping
  - Tuple types
  - Record types
  - Type variables

- Control flow nodes
  - If
  - Loop
  - Phi-node
  - Multiple value
Solver Extensions

XXX: write as algorithm!

- 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

XXX: formalize!

• 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

XXX: formalize!
Optimization: Call upgrade
(generic function)

- choose direct method instead of generic function

- GF dispatch expensive at runtime (iterate through all methods of GF and compute distance, find most specific)

```dylan
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;
```
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))
```
Performance

- XXX: evaluate Dylan library/compiler!
Conclusion

• Implemented formal type system into dynamic programming language

• Gap between static and dynamic typing is getting smaller
Further Work

• 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


- 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](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)


  - 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