Simple Type Inference for Higher-Order Stack-Oriented Languages

Technical Report Cat-TR-2008-001
Draft April 20, 2008

Christopher Diggins

http://www.cdiggins.com

Abstract. Stack based languages are widely used as byte-codes, as intermediate languages, and as programming languages for embedded devices. It is well-known how to check the types of simple stack-based languages such as for the Java virtual machine language, using Hindley-Milner type inference, or using Hoare logic. The simple Hindley-Milner type inference algorithm is insufficient to type-check higher order instructions, such as abstraction operators, in a stack-based language without annotations. This limitation occurs because we require first-class polymorphism.
Our contribution is to describe a simple type-system and type-inference algorithm for higher-order instructions in a single stack stack-based language.

1 Introduction

Many languages compile to intermediate languages that have stack-based semantics [JVML [Lindholm and Yellin(1999)], Python [?], OCaml [?], Ruby [?], CIL [?]]. Several languages used for embedded devices also have stack-based semantics [Forth [Moore(1974)], PostScript [Inc.(1999)], there are even languages designed specifically for general purpose application development with stack-based semantics such as Factor [Pestov(2003)].

It is useful if we can statically verify certain properties of stack-based code: e.g. the stack will never underflow, that an instruction will have the expected number of argument with correct types on the stack when executed.

Some stack-based languages (e.g. the JVML, CIL), by introducing certain restrictions, such as requiring the stack configuration to be the same regardless of which branch the control flow takes. These restrictions can be expressed using a type-system.

It is well-known how to check the types of simple stack-based languages such as for subsets of JVML or CIL, using Hindley-Milner type inference, or Hoare logic. However the simple Hindley-Milner type inference algorithm is insufficient to type-check higher order instructions without annotations. This is because we lack first-class polymorphism

For example we can not push an arbitrary valid subroutine of stack-based instructions on the stack, and apply it to the stack. We show a specific example of this later on. The advantages of having higher-order instructions in a language are well known [?], and are not discussed in this article. We are simply concerned how to introduction higher-order functions to a stack-based language in a type-safe manner.

2 Cat: A Simple Higher-Order Stack-Based Language

For this paper we will consider a statically-typed higher-order stack-based language with only one stack called Cat. By higher-order we mean that functions are first-class values that can be pushed on the stack, as well as taken from the stack and applied to the rest of the stack.

For this article we will be considering a pure subset of Cat that has no effects and does not allow any instruction to have unbounded access to the whole stack. In this subset of Cat, all instructions are, in effect, functions that take a single stack as input and return a single stack as output.

2.1 Abstract Syntax

There are only two operations in the Cat language, abstraction (called quotation in Cat) which is denoted by square brackets ([and]) and composition which is denoted by the sequence of terms.

The abstract syntax of Cat is described thusly:

```
term ::==
  [term]
  term term
  empty
```

It is noteworthy that application is absent from the abstract syntax.

2.2 Type Notations

To express the type-signatures of instructions we use a specific notation based on stack effect diagrams. A function (or instruction) is denoted as follows:

```
(c0 c1 ... cn -> p0 p1 ... pm)
```

Where c0 through c1 are kind expressions describing the types expected on the stack before execution collectively known as the consumption and p0 through pn are the types expected on the stack after execution, collectively known as the production.

The abstract syntax of types of Cat that are considered in this paper is summarized by the following grammar:

```
kind ::==
  type
  stack

type ::==
  a..z     // type variable
  (S0->S1) // function from stack to stack

stack ::==
  A..Z     // stack variable
  S t      // stack containing a type on top
```

Fig. 1. Type system abstract syntax

Note that the production rules explicitly forbid the empty stack, and from putting stacks on stacks. This is key to keeping the Cat type system manageable. We should point out that in the actual Cat implementation type variables and stack variable are preceded by apostrophes and can have multiple characters.

It follows from the productions rules that all functions in Cat are polymorphic, because a stack variable has to be at the bottom of each stack. All functions can be applied to any stack as long as the top items in the stack conform to the consumption.

2.3 Well-typed Expressions

Well-typed expressions in Cat have the following characteristics:

Polymorphism over the rest of the stack All expressions in Cat are polymorphic over the rest of the stack:

```
expr : (R c0 c1 ... cn -> R p0 p1 ... pm)
```

In plain English this means that no instruction in Cat can have unlimited access to the rest of the stack. This restricts certain kinds of instructions such as clearing the stack or reversing the stack. In other words every instruction has to have an effect on a number of items on the stack that can be established statically through type inference. This is related to Wand's notion of row polymorphism [Wand(1987)].

Stacks can not be placed on stacks It follows from the syntax rules of types that a stack variables may not occur above another stack variable.

Types can not occur below stack It follows from the syntax rules of types that a stack variables may not occur above a type variable.

Variable can not occur only in productions A type variable or stack variable must be contained with the consumption of the function, or the consumption of a function appearing in the production.

The following are examples of well-typed expressions that don't violate this rule:

```
(A (A -> B) -> A)
(A -> A (C (C -> D) -> D))
```

The following are examples of ill-typed expressions that violate this rule:

```
(A -> B)
(A -> (A -> B))
```

2.4 Primitives

The core Cat language consist of the following primitive instructions.

- dup - for any stack with at least one item, duplicate the top item
- pop - for any stack with at least one item, remove the top item
- swap - for any stack with at least two items, swap the top two items
- apply - for any stack with a function on top, apply the function to the rest the stack
- quote - for any stack with at least one item, replace the top item with a function that returns that value. Alternatively this could be considered a thunk generating function.
- compose - for any stack with two functions on the top, replace these functions with a new function that is the result of composing the top function with the second function.

The types of these primitives are expressed in our notation as:

- dup : (A b → A b b)
- pop : (A b → A)
- swap : (A b c → A c b)
- apply : (A (A → B) → B)
- quote : (A b → A (C → C b))
- compose : (A (B → C) (C → D) → A (B → D)))

A simple evaluator for the Cat primitives, can be expressed using the following series of term-rewriting rules:

- rule { $a dup } ⇒ { $a $a }
- rule { $a pop } ⇒ { }
- rule { $a $b swap } ⇒ { $b $a }
- rule { [$A] apply } ⇒ { $A }
- rule { $a quote } ⇒ { [$a] }
- compose { [$A] [$B] compose } ⇒ { [$A $B] }

In these rules single value expressions are expressed as lower-case variables preceded by a $ character, whereas arbitrary length expressions bounded by quotation delimiters are expressed as upper-case variables names preceded by a $ character.

A smaller basis (i.e. set of primitive instructions) is of course possible, in fact we can reduce these down to two instructions [Kerby(2002)] and even flatten all code [?] (i.e. remove the explicit quotation operator), however this basis was chosen for its combination of simplicity and clarity.

2.5 Composition versus Application

The sequence of terms in Cat denotes composition rather than application. This means that the following terms are equivalent:

```
f g = [f] [g] compose apply
f g h = [f] [g] compose apply [h] compose apply
```

The property of distributivity of function application over function composition, leads to an interesting result for programs of three or more terms:

```
f g h = [f] [g] compose [h] compose apply
      = [f] [g] [h] compose compose apply
```

This means that there are many different yet equally valid ways to evaluate a Cat program, and equivalently many different ways to type-check a Cat program.

3 Type System

Given that there are only two operations in Cat, there are only two typing judgements in the Cat language: quotation (a.k.a abstraction), and composition. Cat also lacks identifiers, thus we have no need for a type environment, to map from identifiers to types.

3.1 Universal Quantification

In Cat type variables are all universally quantified. The forall quantifying operator is omitted from type annotation since it can be deduced from the following rule:

> Any type or stack variable is assumed to be universally quantified over the inner-most function that includes all occurences of that type variable.

The usage of such a rule simplifies type inference greatly. To demonstrate this rule , consider the types of the primitives with explicit universal quantification:

- dup : forall.A.b.(A b → A b b)

- pop : forall.A.b.(A b → A)
- swap : forall.A.b.c.(A b c → A c b)
- apply : forall.A.B.(A (A → B) → B)
- quote : forall.A.b.(A b → A forall.C.(C → C b))
- compose : forall.A.B.C.D.(A (B → C) (C → D) → A (B → D)))

Including explicit forall qualifiers would make expressing typing judgements much more complicated than necessary, because we would have to explicitly manage forall qualifiers. By omitting them we can construct a type reconstruction algorithm that only has to deal with type variables and stack variables. However, an explicit renaming step has to be added.

3.2 Quotation

Quotation is the Cat equivalent of an abstraction operation. It pushes a Cat expression on the stack without evaluation. Quotation is denoted by square brackets, e.g. [t0 t1 ... tn]. The type of a term that is quoted is defined by the following typing rule:

```
f : (A -> B)
----------------------- T-QUOTE
[f] : (C -> C (A -> B))
```

Note that this is a short form of the following more complete rule:

```
f : forall.R.(R c0 ... cn -> R p0 ... pm)
------------------------------------------------------------------- T-Q
[f] : forall.S.(S -> S forall.R.(R c0 ... cn -> R p0 ... pm))
```

However, as we stated previously the forall qualifiers are implied.

3.3 Composition

Composition is indicated in Cat by the sequencing of two terms together.

```
f : (A -> B) g : (B -> C)
------------------------- T-COMPOSE
f g : (A -> C)
```

4 Type Inference

Type inference, also called type reconstruction, is the process of finding the best type that satisfies an expression. Failure to do so, indicates a type error. By far the most widely used and understood type inference algorithm is the Hindley-Milner (HM) type inference algorithm. The type inference algorithm used by Cat is essentially the same as HM with the introduction of a new rule for renaming generic variables.

Type reconstruction consists of the following steps:

– applying the type judgement rules T-QUOTE or T-COMPOSE
– generate constraints
– solve constraints using unification

When composing two functions, a constraint is generated because the production of the first term must be equal to the consumption of the second term.

When generating constraints we follow a recursive process:

– when constraining two vectors if both vectors have a top type, create a new constraint by equating the two tops, then generate constraint by equating the rest of the vector
– when constraining two functions, generating constraints by equating the consumption then the production

Given a set of constraints we apply a unification algorithm. This is explained very well in Programming Languages: Application and Interpretation by Shriram Krishnamurthi [Krishnamurthi(2006)], and in many other sources. Our only modification is that during substitution, when substituting a function for a type variable we rename all generic variables in the function.

4.1 Type Reconstruction Example

A simple example that is illustrative of the Cat type inference algorithm are deriving the type of the expression "‘[dup] apply"’:

Starting with "‘dup"’ we apply the quotation rule as follows:

```
dup : (A0 b0 -> A0 b0 b0)
------------------------------------ T-QUOTE
[dup] : (A1 -> A1 (A0 b0 -> A0 b0 b0)
```

Next we compose with apply as follows:

```
apply : (A2 (A2 -> B2) -> B2)
---------------------------- T-COMPOSE
[dup] apply : (A1 -> B2)
```

Of course this is not the end of the story, the following constraint is generated:

```
A1 (A0 b0 -> A0 b0 b0) = A2 (A2 -> B2)
```

Unifying these two generate the following additional constraints:

```
(A0 b0 -> A0 b0 b0) = (A2 -> B2)
A1 = A2
```

Next we unify the two functions:

```
A0 b0 = A2
A0 b0 b0 = B2
A1 = A2
```

No more unifications are left to do, so we start with the substitution. We replace all instances of "'A2"' with "'A0 b0"' because it is longer.

```
A0 b0 b0 = B2
A1 = A0 b0
```

Now we replace "'B2"' with "'A0 b0 b0"' and "'A1"' with "'A0 b0"' in our final result giving us:

```
[dup] apply : (A0 b0 -> A0 b0 b0)
```

This type is now immediately recognizable as being the precise type we expect.

4.2 Generic and Non-Generic Type Variables

In [Cardelli(1987)] Luca Cardelli the notion of generic and non-generic type variables. In order to infer types in Cat we have to consider the genericity of type and stack variables. Type and stack variables are considered either generic or non-generic relative to a particular function. A generic type variable does not occur outside of the particular function. This is not a property that we manage, but rather something that we query during unification.

There is a correlation between forall quantifiers and generic variables: The forall quantifier expresses that a function is the outer-most enclosing function for which the variable is generic.

4.3 Renaming Variables

In the Cat type inference algorithm an important rule which is absent from simple ML type-inference is to assign unique names to all generic variable in a function when unifying a function with a type variable.

Consider the case of composing the expression "'[1] : (A -¿ A (B -¿ B int))"' with the expression "'dup : (A b -¿ A b b)"'. Following the normal type unification process we would arrive at the following rules:

```
[1] dup : (A -> A (B -> B int) (B -> B int))
```

The problem in the above code is that B is non-generic relative to each of the two functions on the stack. In other words the types of both functions are polymorphic, but dependent.

If we make the universal quantifier operator explicit, it may help to illustrate the problem. The following is the erroneous type:

```
[1] dup : forall.A.B.(A -> A (B -> B int) (B -> B int))
```

The following is the correct type:

```
[1] dup : forall.A.(A -> A forall.B.(B -> B int) forall.C.(C -> C int))
```

4.4 Decidability

We do not know whether or not the type reconstruction algorithm we have described is decidable. Nonetheless we have yet to see it fail in practice, and are optimistic that practical application of the algorithm in compilers will not be impeded even if decidability is not proven. To quote Luca Cardelli [Cardelli(1987)]

> Type checking is usually restricted to decidable type systems, for which type checking algorithms can be found. However in some cases undecidable systems could be treated by incomplete type checking heuristics (this has never been done in practice, so far), which only attempt to prove theorems in that system, but may at some point give up. This could be acceptable in practice because there are limits to the complexity of a program: its meaning could get out of hand long before the limits of the type checking heuristics are reached.

Proving the decidability of the algorithm, would be an interesting area of research, out of the scope of this technical report.

5 Related Work

5.1 Type Systems

The following description of Hindley-Milner type systems by Daan Leijen [Leijen(2008)] is a very lucid and accurate summary of the limitations of HM types as they apply to languages that require firs-class polymorphic values such as higher-order stack-based languages like Cat:

> The Hindley-Milner type system restricts polymorphism where function arguments and elements of structures to be monomorphic. Formally, this means that universal quantifiers can only appear at the outermost level (i.e. higher-ranked types are not allowed), and quantified variables can only be instantiated with monomorphic types (i.e. impredicative instantiation is not allowed).

These two limitations of the HM type system are precisely what we have had to address in this paper. We allow impredicative instantiation of types, and higher ranked types.

The type system which appears to have the most in common with the one presented in this paper is the HMF [?] type system, which is the Hindley-Milner type system extended with regular System F [?] types. System F allows polymorphic values as first-class citizens.

5.2 Languages

[to do]

6 Extending Cat with Self Types

It appears to be possible to extend the typing discipline with a limited form of equirecursive [?] function type to enable the typing of a wider number of phrases.

A motivating example of the need for recursive types is the m-combinator defined by the expression ”‘dup apply”’ which is an important component of the y-combinator.

The approach we propose is for the recursive relation checker to return a special indicator, to indicate a recursive type has occurred. We identify these as ”‘self”’ types.

During the unification algorithm recursive types can be identified when a constraint is generated in which a type variable is required to be equivalent to a function within which the type variable occurs. This requires an additional check to see that a type variable is contained within a particular function. This is a simple scan of the type variables in a function, which are finite, thus the algorithm is guaranteed to terminate and should not affect the overall decidability of the program.

For example, when computing the type of ”‘dup apply”’ we have to compose the types:

```
dup:(A0 b0 -> A0 b0 b0) apply:(A1 (A1 -> B1) -> B1)
```

This gives us:

```
dup apply : (A0 b0 -> B1)
```

With the following initial constraints:

```
(A1 -> B1) = b0
A1 = A0 b0
```

Note however that as we unify constraints we end up with a constraint to a function that refers to itself:

```
A1 = A0 (A1 -> B1)
```

When we expand the variable A1 we get:

```
A1 = A0 (A0 (A1 -> B1) -> B1)
```

Because we see that (A1 → B1) is equal to (A0 (A1 → B1) → B1) which contains (A1 → B1) we can replace (A1 → B1) with a ”‘self”’ type.

```
A1 = A0 (A0 self -> B1)
```

Now we perform the substitutions with the original result:

```
A0 = A0
b0 = (A1 -> B1)
A1 = A0 (A0 self -> B1)
b0 = (A0 (A0 self -> B1) -> B1)
dup apply : (A0 (A0 (A0 self -> B1) -> B1) -> B1)
```

There is a final step called a roll-up, where any function type containing a function with a self type is checked for equivalency. If all aspect are equal, we replace it with the child. This is demonstrated in the following steps:

```
dup apply : (A0 (A0 (A0 self -> B1) -> B1) -> B1)
dup apply : (A0 (A0 self -> B1) -> B1)
dup apply : (A0 self -> B1)
```

Self types are restricted to functions which refer to themselves. If a function f contains a function g that refers to f, then this is considered a type error and can not be handled by the type inference algorithm.

When unifying a self type with a type variable, the type variable takes precedence. It is considered a better match for a type than a self because it is more general.

7 Acknowledgments

John Cowan, Daniel Ehrenberg, Marc Feeley, Frank Krueger, David Haguenauer, Colin Hirsch, Stefan Monnier, John Nowak, Andreas Rossberg, Anton von Straaten, William Tanksley Jr., Manfred von Thun, and Kris Unger.

Special thanks to Melanie Charbonneau for supporting me through the hard times.

References

Cardelli(1987). Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987. URL `citeseer.ist.psu.edu/cardelli88basic.html`.

Inc.(1999). Adobe Systems Inc. *Postscript Language Reference*. Addison-Wesley, 3rd edition, 1999. URL `http://www.adobe.com/products/postscript/pdfs/PLRM.pdf`.

Kerby(2002). Brent Kerby. The theory of concatenative combinators. http://tunes.org/ iepos/joy.html, 2002.

Krishnamurthi(2006). Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. self-published, 2006. URL `http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-`

Leijen(2008). Daan Leijen. Hmf: Simple type inference for first-class polymorphism, 2008. URL `http://research.microsoft.com/users/daan/download/papers/hmf-tr.pdf`.

Lindholm and Yellin(1999). Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems Inc., 2nd edition, 1999. URL `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

Moore(1974). Charles Moore. Forth: a new way to program a mini-computer. *Astronomy and Astrophysics Supplement*, (15), 1974.

Pestov(2003). Slava Pestov. Factor programming language. URL `http://www.factorcode.org/`. Programming language implementation and documentation., 2003.

Wand(1987). M. Wand. Complete type inference for simple objects. In *In Symposium on Logic in Computer Science*, June 1987. A Corrigenda appeared in LICS 88.