

# Polymorphism by Polyinstantiation

Preliminary Implementation Note

Jonathan Shapiro, Ph.D.  
*Systems Research Laboratory*  
Dept. of Computer Science  
Johns Hopkins University

September 6<sup>th</sup>, 2006

## 1 Options for Implementing Polymorphism

Traditionally, there are several ways in which first class polymorphism has been implemented:

- Fully boxed: all data-structures are fully boxed (original ML[2])
- Coercions[3]: data-structures can have a custom unboxed representation, but will be coerced (by conversion at run-time) into a boxed representation when they have to be passed as argument to a function requiring polymorphic argument.
- Dictionaries[4]: data-structures can be unboxed, but we must pass extra type-parameters to functions.
- Hybrid[5]: Using a mix of coercion and dictionaries at various levels.
- Full Polyinstantiation: Functions are customized per-type. (many compilers have used this too various degrees, ex: C++ templates, MLton?)

Depending on which point we pick in the continuum of choices, there are different trade-offs w.r.t. the amount of RTTI support needed, separate compilation, efficiency, code size, dealing with the var-args problem[5], etc. Shao[5] discuss these trade-offs in detail (except for full polyinstantiation).

In the bootstrap compiler for BitC[1], we decided to take implement polymorphism by full polyinstantiation. This will also simplify the implementation of overloading (type-classes), at the cost of requiring whole-program compilation. In this document, we try to describe our approach to implement polymorphism by polyinstantiation, and the associated problems and their solutions.

## 2 Compiler organization

Currently, the compiler performs the following (major) passes:

1. Parse
2. Symbol resolution
3. Type inference
4. Location semantics checking
5. Polyinstantiation

6. Closure conversion and lambda hoisting
7. SSA transformation
8. Backend (currently C) emission

The passes 1-4 shall hereinafter be termed as "frontend passes," and passes 6-8 as "backend-passes."

All passes up to the SSA pass (including intermediate simplification passes) are Bitc AST -> Bitc AST transformations. Even the output of the SSA pass is a mostly-bitc AST, with some extra annotations. This has the advantage that we can re-run the symbol-resolver and type-inference after each pass that does some AST transformation, and be sure that the compiler did type-correct transformations. This will also reduce the burden on developers to build correct type records whenever an AST-change is introduced. This ability to re-run the resolver and type-checker (hereinafter called the R&T step) is also heavily used in the implementation of polyinstantiation.

The front end is done one unit of compilation (one interface or source module, abbreviated as UOC) at a time, or, in an interactive environment, one top level form at a time. The front end also pulls in other dependent interfaces and also processes them. After the frontend is done, the polyinstantiator comes into play. It receives a set of FQNs (*entry points*) that must be instantiated in this pass – for example, `bitc.main.main`, or the current expression in a top-level interactive loop. It starts out from these as roots and constructs an output-UOC in an on-demand basis. Then, it passes this output-UOC to the backend passes.

### 3 What must be Polyinstantiated?

We have three main issues with polyinstantiation:

- Polyinstantiation of top-level definitions. For example, if two instantiations of the identity function:

```
(define (id x) x)
id: (fn ('a) 'a)
```

have types `(fn (bool) bool)` and `(fn (int32) int32)`, then the definition of `id` is polyinstantiated to two definitions as:

```
(define id#fn_bool_bool:(fn (bool) bool)
  (lambda (x) x))
id#fn_bool_bool:(fn (bool) bool)

(define id#fn_bool_bool:(fn (bool) bool)
  (lambda (x) x))
id#fn_bool_bool:(fn (bool) bool)
```

- Polyinstantiation of local definitions (defined at `let`, `letrec`)
- Any user defined types / constructors used in the polyinstantiated value definitions must be transitively polyinstantiated.
- All calls to type-class methods must be replaced with the instantiation of functions defined in the appropriate instantiation of that class.

## 4 The Polyinstantiator

### 4.1 Environment Handling

The polyinstantiator needs an environment in which we have the entries from all interfaces and modules so that newly instantiated definitions can be R&Ted (see Section 4.3). We refer to this environment as Mega-ENV, and it contains

symbol, type and typeclass-instance information from all UOCs. Moreover, since the Mega-ENV contains inter-UOC information, it must always be indexed by the fully qualified names (FQNs) of all variables. In order to polyinstantiate a definition completely, we must construct such an environment and pass it to the specializer.

How should the Mega-ENV be built? Of course, for the first time, we must build it by iterating over all the environments of all interfaces and source modules and build a new environment based on their FQNs. However, we don't want to do it every time we enter the instantiator. Since here is that the interface-list and the source-module list are append-only – they are append-only in the case of an interactive interpreter; in the case of static compiler, these lists are frozen – we can note the previously last processed indices into these lists, and only processes additions each time. In the case of a compiler, interfaces and source module environments are immutable once the entire module is processed. However, in the case of an interactive interpreter, there is exactly one unit of compilation that is ever expanding, and the instantiator must cope with this. So, we mark this UOC as “dynamic,” so that the instantiator will re-process it every-time.

Once the Mega-UOC is built, we set it up as the *parent* of the unified-UOC's environment into which new definitions are emitted. Thus, when we try to R&T a newly instantiated definition, not only are all the old definitions and instances available but also all the polyinstantiated definitions. Appropriate name mangling is used to ensure that no polyinstantiated definition will ever collide with an original definitions. Moreover, all typeclass instances are required to be globally distinguishable with respect to the whole program. Thus, we will have no collisions here either.

Once we have updated the Mega-ENV, we call the Specializer(Section 4.3 on each (input) entry-point.

## 4.2 Name mangling

Every identifier *id* with FQN *fqn* and type *typ* is replaced with the mangled name:

$$- + \text{length}(fqn) + fqn + \# + \text{mangled\_string}(type)$$

The '+' operator is meta-syntax for string pasting, and is not a part of the new identifier's name. *length* is the obvious string length function and *mangled\_string* function returns a unique string representation of every type.

For example, the type *list* defined in the interface `bitc.prelude`, if instantiated to the type `(list int32)` will have the name `_17bitc.prelude.list#UR1_4list_5int32`.

## 4.3 The Specializer

The following is the algorithm for the Specializer: It takes as input the AST of a (possibly polymorphic) definition, and a *concrete* type to which this definitions must be instantiated. In the case of polyinstantiating a local definition, it takes in a let-binding. It returns the newly instantiated identifier (not the entire defining form).

The Specializer maintains a *worklist* of definitions currently being processed, to ensure that recursive definitions do not cause the instantiator to go into an infinite loop. All R&Ts in the following algorithm should be understood to be performed with respect to the environment of the Unified-UOC.

1. See if the desired identifier has been instantiated for the desired type by searching the Unified-UOC's environment. If found, just return the identifier for the corresponding definition.
2. Examine the worklist whether we are in the process of instantiating the current definition. If so, emit a declaration if we are instantiating a global, and return the identifier being declared. If we are in the process of instantiating a local definition, we don't need a declaration; so just make-up the right name for the local instantiation and return it.
3. Copy the incoming AST – always copy the definition (rather than the declaration) if one exists.
4. If we are instantiating a type definition to a concrete type *typ*, instantiate the structure to the maximally mutable type that is copy-compatible with *typ* (See Appendix D.)

5. Mangle the name of the identifier being defined depending on the desired type, and also change any recursive usages of the same name within the current definition. For example:

```
(define _8myMod.id#Fn_4bool_4bool (lambda (x) x))
```

6. Clamp the type of this AST to the desired type by introducing a qualifier. For example, to instantiate `id` to `(fn (bool) bool)`, we write:

```
(define _8myMod.id#fn_4bool_4bool:(fn (bool) bool) (lambda (x) x))
```

Here, we must *overwrite* any qualifications already done by the user because we will have strictly more information, and we cannot write the qualification anywhere else (for example, on the RHS) due to copy-compatibility[6].

If we are specializing a type definition (`defstruct` / `defunion`), then replace the usage of type-arguments to that type with concrete types, and erase the argument list of that type.

7. R&T the new definition (or let-binding) so that the types in the body of the definition get clamped aiding further instantiation. After successful R&T, throw away any changes to the environment.<sup>1</sup>

In the case of a let-binding, we must R&T the let-binding with respect to the local environment(which the resolver, type-inference engine will have as part of the AST). However, if we receive a let-binding as input, we must be called from `recInstantiate` (Section 4.4, which means that the containing definition has already been R&Ted in the Unified-UOC environment. So, the local environment will have the Unified-UOC's environment and thus the Mega-ENV as parents.

8. Add the mangled name plus a unique identifier of the AST being instantiated onto the worklist. The unique ID is required because names of local instantiations might collide.
9. Recursively instantiate the body of the newly instantiated AST (Section 4.4).
10. After the current AST is completely instantiated:
  - In the case of a global instantiation, append it to the module in the Unified-UOC. R&T it once more but keep changes to the environments so that further instantiations can use it.
  - In the case of a local instantiation, add the instantiated let-binding to the inner-wrapper of the current `let` expression (Section 4.4).
11. Remove the current definition from the worklist, and return the identifier corresponding to the definition just instantiated.

## 4.4 The recursive Instantiator

The recursive instantiation propagator (`recInst`, for short), will walk through *newly instantiated* definitions and look for usages of other definitions and call the `specializer` on these definitions. There are a couple of special cases that `recInst` must handle:

1. *Case Lambda/ switch<sup>2</sup>*: These expressions bind new variables (argument/local name) but not polymorphically. Therefore, just replace the names of all bound identifiers with their mangled-names. Also, replace all use-cases of these identifiers. Later, `recInstantiate` the body (rest of the AST).
2. *Case Type-qualified expression*: Replace the type-qualification with an explicitly concrete type from the type-obtained. Otherwise, type-variable scoping will play havoc, especially while dealing with let-polyinstantiation. R&T the (AST for) type-qualification, and `recInstantiate` it.

---

<sup>1</sup> It might seem that this R&T can be done with respect to the original UOC in which the current definition is defined, and there is no need to construct a Mega-ENV, but this is not true (Appendix B).

<sup>2</sup> To be specific, each leg of a switch expression, except the `otherwise` clause.

3. *Case Int/Float Literal*: Since we no longer have type classes after polyinstantiation, *every* int / float literal must be qualified with the precise type.
4. *Case Identifier usage*: We should now fetch the definition and (poly)instantiate it. However, this has some sub-cases:
  - If the identifier is a value constructor, instantiate its container type fully.
  - If the identifier is a type-class method, fetch the implementation from the *appropriate* instance. Note that there must exist an instance that will satisfy this usage as the program was declared type-correct. An instance may provide, as implementation for a method of this typeclass, a method from another class. Therefore, this process must be repeated until we find a “real” function to instantiate.
  - Otherwise, this is a “normal” identifier usage, just fetch the definition of this identifier. In the case of a local definition, the definition is the corresponding let-binding.

After fetching the right definition to instantiate, call `specializer` with this definition, and the type required by the use of the definition. In case the type of the use case of the definition is non-concrete, fix it to some concrete type (say `unit`<sup>3</sup>) before calling the `specializer`. Replace the identifier usage with the identifier returned by the `specializer`.

5. *Case let/ letrec*:

- Wrap the body of the current definition with another (respective) `let/ letrec` container with empty bindings<sup>4</sup> (See Appendix C). For example:

```
(let ((id (lambda (x) x))
      (id ()))
      (id #t))
```

is changed to

```
(let ((id (lambda (x) x))
      (let ()
        (id ())
        (id #t))))
```

- Now `reInst` the body of the `let`-expression (not the bindings). This will (probably) trigger instantiations of the local bindings, while the `specializer` will add to the inner `let` expression.
- After the body is completely instantiated, carry forward (by `reInstantiating`) all non-polymorphic bindings in the outer definition onto the inner definition. This must be done in order to preserve their side-effects. Note that polymorphic expressions cannot have side-effects due to the value-restriction[1].
- Finally, discard the outer wrapper, and declare the inner `let` expression as “the” real AST. In case no bindings are carried forward (all the bindings were polymorphic, and unused in the body), just drop the `let` and declare the body as the instantiated expression.

## 5 Comments

We have explained the mechanism of implementing polymorphism by polyinstantiation used in BitC. It should be noted that since the polyinstantiator instantiates ASTs on demand, it performs the “tree shaking” optimization by construction. That is, only those definitions (including library code) that are ultimately required for the correct execution of the program are processed and emitted by the backend of the compiler.

<sup>3</sup> We may want to instantiate it to some well-known dummy type so that we can properly pretty-print it for the user.

<sup>4</sup> This is an invalid AST, but we will fix it before R&Ting it.

## A The Big-AST approach to polyinstantiation

This section describes how polyinstantiation was originally implemented in the compiler, and identifies some associated problems with it. Here, we only deal with instantiation of top-level definitions.

1. After all the front-end passes have been run on all interfaces and source modules (successfully), perform a ‘link’ step. Here, build a new source module (called big-AST) by appending all top-level forms from all interfaces and source modules (in that order). While building this big-AST, the names of all identifiers being defined is replaced with their fully-qualified names (FQNs) to ensure non-collision.
2. R&T the big-AST. If there are any link-time errors, abort. Else, we have a properly typed big-AST. We also have an `_environment_` where all the definitions in the entire program exist. This can be used in further resolution / typing of new definitions introduced during polyinstantiation. We have now also seen all (typeclass) instance in the entire program. The instances are required to be globally non-interfering wrt the entire program.
3. We maintain a ‘todo’ list of things that we want to emit into the backend. We start from this list and must emit these definitions as well as those required by them – transitively – into the backend. Initially, this list is something like `bitc.main.main`, any external definitions and exceptions, etc. All of the things in initially in the todo list must have concrete type. Call the procedure ‘Specialize’ (see below) on each item in the todo list (on its own type) till it becomes empty.  
After all specializations is done, remove all `_polymorphic_` ASTs that are unreachable. (Polymorphic definitions cannot induce state changes).
4. Rearrange the big-AST in dependency order so that the use of a type or definition comes after its definition. Optionally, redundant declarations may be removed at this step.
5. R&T the big-AST once again to ensure validity and we are done.

The algorithm for Specializer: Specializer takes 2 arguments a (possibly polymorphic) definition, and a type to which desired instantiation is desired. [Note that the definition that is being specialized is already in the big-AST]

1. If the desired definition has been instantiated for the desired type, just return.
2. If specialization is requested on a declaration, and if we have a definition, first specialize that, then emit a declaration based on the name of the specialized definition.
3. If the requested specialization is on a union constructor, specialize the entire union.
4. If the specialization is on a type-class method, look up the list of available instances. get the instance and specialize that instead. Note that this step handles the case where the instance of one method is satisfied by another method as well.
5. Now that we have a definition to specialize, make a copy of the definition’s AST. Mangle its name based on the name plus the target type desired (there is a unique mangle for every type).
6. Clamp the type of this AST to the desired type by introducing a qualifier. In cases of specializing a type, replace all type arguments with concrete types.
7. R&T this definition in the big-AST environment. This will build new type records for the body of the definition which we can later use for further instantiations.
8. Recursively walk the body of this definition and add any other definitions needed by this body to the todo list. Note that this must add all definitions used regardless of whether this involves specialization or not, because they may contain other concrete uses of other polymorphic definitions that are not exposed by the outer type.
9. Finally after the entire AST is specialized again R&T it to make sure we are fine and add it to the big-AST. After the definition is processed, fix the use to of this definition (because of which we started specialization) to refer to the new definition.

However, the above algorithm has the following drawbacks:

- It forms the bigAST from all the definitions known in the entire program, rather than doing it ‘on-demand’ one-output point at a time. In this sense, it is one-shot-compiler friendly and may not scale to an interactive interpreter.
- It mutates the big-AST in place, and is thus not “functional” in style – something that is much desired keeping the next version of the compiler in mind.
- It needs a post-instantiation dependency analysis and re-ordering as the definitions are not emitted on a demand-driven basis.

## B Why do we need the Mega-ENV?

Suppose that we do not have a Mega-ENV, but instead R/T every new definition in the environment of the UOC in which the original definition was present. Now, Consider the following UOCs:

```
(interface IF1
  (proclaim UID:(fn ('a) 'a))
)

(module SM1
  (provide if1 IF1)
  (defunion (un 'a) (Ctr c:'a))
  (define (UID x) (Ctr x) x)
)

(module SM2
  (defstruct St st:bool)
  (define p (UID (St)))
)
```

If we want to instantiate p:

1. We start with SM2’s environment, need to instantiate p. So, we copy the definition of p as:

```
(define _5SM2.p#UR_6SM2.St:SM2.St (UID (St)))
```

and type it in SM2’s environment, everything is OK.

2. Now, we need to recurse over this definition, we encounter UID, get its definition, see that it is a proclamation, and get its real definition. Then, we make a copy of this definition, and modify it:

```
(define UID#FN_1_6SM2.St_6SM2.St:(fn (SM2.St) SM2.St)
  (lambda (x) (SM1.Ctr x) x))
```

Now, we must type it. Which environment should we type it in? Not in SM1 because we don’t know ‘St’ there, and not in SM2 because we don’t know ‘un’ there. Not in the big-AST environment as there is nothing relevant there.

There is no single environment which has enough information to type this new definition, unlike the original (take-1) scheme, where we were forming one big AST comprising of all definitions. So, the conclusion is that even though the unified-AST can be formed on demand, R&Ting the new definitions must be done in a totally-rich environment similar to the take-1-big-AST’s environment.

## C More about Local Instantiations

Why do we need an extra wrapper (as described in Section 4.4) for instantiating `let` or `letrec` bound definitions? For example, in the following expression,

```
(define top
  (let ((id (lambda (x) x))
        (id ())
        (id #t)))
    ()))
```

`id` must be instantiated locally to two functions with types `(fn (())) ()` and `(fn (bool) bool)`. It might seem, that the instantiator can add extra bindings to the current `let` expression, and instantiate it as:

```
(define top
  (let ((id (lambda (x) x))
        (id#FN_4unit_4unit: (fn (())) ()) (lambda (x) x))
        (id#FN_4bool_4bool: (fn (bool) bool) (lambda (x) x))
        (id#FN_4unit_4unit ())
        (id#FN_4bool_4bool #t))
    ()))
```

And later drop the polymorphic definition of `id`. While this works fine for `id`, it will fail in more complicated cases. For example:

```
(define top
  (letrec ((odd (lambda (x) (even x)))
           (even (lambda (x) (odd x))))
    (even 0:int32):int32))
```

The original type for both `odd` and `even` is `(fn ('a) 'b)`. When we process the body of the `letrec`, we see that `even` must be specialized for `(fn (int32) int32)`, so we get:

```
(define top
  (letrec ((odd (lambda (x) (even x)))
           (even (lambda (x) (odd x) ))
           (even#FN_5int32_5int32:(fn (int32) int32)
                                   (lambda (x) (odd x) )))
    (even#fn_int32_int32 0:int32):int32))
```

Now if we R&T this form, we get:

```
odd: (fn (int32) int32)
even: (fn (int32) int32) and
even#FN_5int32_5int32:(fn (int32) int32)
```

since all `let`-bindings are typed and generalized together. So, any further usages of `(even 200:int64)` or `(odd #t)` will fail. Therefore, we should not add new `let`-bindings to the same `let` expression, but introduce an inner `let`-binding, that will encapsulate the body. After one step of instantiation, the above expression will now get modified as:

```

(define top
  (letrec ((odd (lambda (x) (even x)))
           (even (lambda (x) (odd x) )))

    (letrec ((even#FN_5int32_5int32:(fn (int32) int32)
              (lambda (x) (odd x))))
      (even#FN_5int32_5int32 0:int32):int32)))

```

Now the types at the original let-binding are left unchanged, and the symbols in the inner `letrec` are built on a per-use basis. Further specialization must be careful to emit things to the inner `letrec` and not create a new let form for every binding – especially in the case of a `letrec`. Of course, in the end, we will have to carry forward all non-polymorphic definitions in the original let-binding that were not already specialized/referenced, and drop the original letbinding.

```

(define top
  (letrec ((even#FN_5int32_5int32:(fn (int32) int32)
              (lambda (x) (odd#FN_5int32_5int32 x))))
    ((odd#FN_5int32_5int32:(fn (int32) int32)
      (lambda (x) (even#FN_5int32_5int32 x))))
    (even#FN_5int32_5int32 0:int32):int32)))

```

## D Copy-compatible and Polyinstantiation

Consider the following definitions:

```

(define mb:(mutable bool) #t)
(define p1:(pair bool (mutable bool)) (pair #t mb))
(define p2:(pair (mutable bool) bool) p1)

mb: (mutable bool)
p1: (pair bool (mutable bool))
p2: (pair (mutable bool) bool)

```

Now, if we want to instantiate `p1` and `p2`, the type `pair` will consequently be poly-instantiated to the following structures:

```

(defstruct _17bitc.prelude.pair#SV2_20_17bitc.prelude.pairM_4bool_4bool:val
  _3fst:(mutable bool)
  _3snd:bool)

(defstruct _17bitc.prelude.pair#SV2_20_17bitc.prelude.pair_4boolM_4bool:val
  _3fst:bool
  _3snd:(mutable bool))

```

Now, the instantiation of `p1` has the type `_17bitc.prelude.pair#SV2_20_17bitc.prelude.pairM_4bool_4bool` and the instantiation of `p2` has the type `_17bitc.prelude.pair#SV2_20_17bitc.prelude.pair_4boolM_4bool`. Therefore, the initialization of `p1` from `p2` will not type-check, as they now have different types, and not different instantiations of the same type.

## References

- [1] J. S. Shapiro, S. Sridhar, M. S. Doerrie, “BitC Language Specification”  
<http://coyotos.org/docs/bitc/spec.html>

- [2] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised* The MIT Press, May 1997.
- [3] X. Leroy, “Unboxed objects and polymorphic typing.” *ACM SIGPLAN Symposium on Principles of Programming Languages* pages 177–188, January 1992. 8(4):343–355, 1995.
- [4] R. Harper, G. Morrisett, “Compiling polymorphism using intentional type analysis.” *ACM SIGPLAN Symposium on Principles of Programming Languages* pages 130-141, January 1995.
- [5] Zhong Shao, “Flexible representation analysis.” *ACM SIGPLAN international conference on Functional programming* Pages: 85-98, 1997.
- [6] “Issues with Type Inference in BitC” <http://www.coyotos.org/docs/bitc/mutinfer.html>