# Wrinkle Update 2016/10/30

I've moved the Method Conversion earlier in the process, and in the consequence of doing this added two extra responsibilities onto the Typechecker.

The first is approximately implemented in the current typechecker, which is that any closure may have the "typecheckMessages" flag set which requires it to assert that the return value is either a Message or a List of Messages (or presumably a Cons of messages or an individual message type).

The second requires it to be able to assert that an expression conforms to a slot type. This is done by using an "AssertTypeExpr".

# HSIE Update 2016/10/08

I made notes about doing this in HSIE in a notebook in B&N.

One additional case I didn't think of at the time:

Also, work very hard at getting the right error messages in the right places. Each PUSH argument should have the location of the start of the thing it refers to and each closure should be associated with the start of the expr (probably the location of the function).

For conflicts, we want to list all the places the variable is used and what the parameter is.

Also, approach this by coming up with specific input cases that show up the various error messages and put them in golden files.

# Update 2016/10/08

Ok, why is this so complicated?

I may be having one of those "oh, yeah, because ..." moments, but I'm thinking that having really done this now, it really doesn't have to be that complicated.

Suppose that you are typechecking "everything" (the "just an expression" case is easier, I claim, and a subset of this).

You have a set of "knowledge" which is recorded, scoped symbols with given types.

You then have an "orchard" of definitions which are somehow mutually recursive to test together. Let's assume they're all functions, because I think that's the only true, hard, recursive case.

So, for each function, you create a new putative type. From it's own definition, you have an explicit arity and a number of cases. The type must be A->B->C->D where the number of types is (arity+1 [the return value]) and each type is (for now) a "Bucket" type which means various different things can go in there.

For each function case, you look at each argument in turn and each pattern in turn. If it's just "x" then you put "FreshPoly" in the bucket. If it's a constant, you put that constant in the bucket. If it's a typed var, put the type in there. If it's a constructor match, you put the derived implied type in there (derived by following the pattern all the way down, since it could have nested typed patterns or constructors).

Now each case will put one entry in each bucket for each parameter. Note that the entries in the bucket are "Or"ed together to make the final type.

Here comes the tricky bit: once you've done this, you then need to look at the code for each case and see what constraints it places on any "FreshPoly" entries (constraints on other vars must either be met or cause an error). So, for example:

f x = x +2

puts a "FreshPoly" in the bucket, but then evaluation of (x+2) makes you realize that x must be a Number.

Unification ONLY (IMHO) comes into the picture when you have mutually recursive functions that pass a var one way or the other and then you have to say that it's not a FreshPoly any more but a FreshPolyPair (so maybe include that in the definition of FreshPoly).

FINALLY, you have to look at what's in each of your buckets and ask: "is this a single type?", "is this a union type?"

Before acting on any of the above, read the rest of what follows. I don't remember what I wrote and it could either be the same thought, or have contradictions.

ONE ADDITIONAL COMPLICATION is the typing of method messages, which we do during method conversion and requires an additional "fake" HSIE conversion, which is before we've dealt with scoped vars. I'm not sure whether we need to do this or if, during typechecker rewrite, we could add the notion of "closure constraints" i.e.:

* There is a new form of expression "TypeExpression" which has the type of the message expression (do we already have CastExpr? What does that do?)
  + For Assign, this is the type of the field to be assigned; the rhs must conform to that type
  + For Send to a contract method, it is the declared type of the contract method
  + For Send to a previously resolved standalone method, it is the previously inferred type of the method
  + For Send to a standalone method in the current orchard, it needs to be the variable assigned to the output of that method
* This is wrapped around an apply expression (the right hand side of the message)
* During HSIE, this is used to attach a type constraint to the closure representing the return var of the closure
* This is then used to force typechecking to identify both the input and return types in one go.
* We can then eliminate the other hack, and also get things to go in about the right order.

# Worked Example

It seems to me that filter covers most of the cases I find interesting:

Consider this definition:

filter f [] = []

filter f (a:l) = g (f a)

g true = a:(filter f l)

g false = (filter f l)

Then this produces this HSIE:

HSIE for test.golden.filter

#Args: 2 #bound: 5

externals: [Cons, Nil] scoped = []

all vars = [v0, v1, v2, v3, v4, v5, v6]

HEAD v1 ??

SWITCH v1 Cons filter.fl: 3.13

BIND v2 v1.head ??

BIND v3 v1.tail ??

RETURN v6:clos6 [v4:test.golden.filter\_1.g, v5:clos5] filter.fl: 3.17

SWITCH v1 Nil filter.fl: 2.9

RETURN Nil filter.fl: 2.15

ERROR

CLOSURE v4 ??

PUSH test.golden.filter\_1.g filter.fl: 4.0

PUSH v2:a filter.fl: 3.10

PUSH v0:f filter.fl: 3.7

PUSH v3:l filter.fl: 3.12

CLOSURE v5 ??

PUSH v0:f filter.fl: 3.7

PUSH v2:a filter.fl: 3.10

CLOSURE v6 ??

PUSH v4:test.golden.filter\_1.g filter.fl: 4.0

PUSH v5:clos5 filter.fl: 3.20

-------

HSIE for test.golden.filter\_1.g

#Args: 1 #bound: 3

externals: [Boolean, Cons, test.golden.filter] scoped = [a, f, l]

all vars = [v0, v1, v2, v3]

HEAD v0 ??

SWITCH v0 Boolean filter.fl: 5.2

IF v0:ev0 false

RETURN v1:clos1 [] filter.fl: 5.11

IF v0:ev0 true

RETURN v3:clos3 [v2:clos2] filter.fl: 4.10

ERROR

CLOSURE v1 ??

PUSH test.golden.filter filter.fl: 5.11

PUSH Scoped[f] filter.fl: 3.7

PUSH Scoped[l] filter.fl: 3.12

CLOSURE v2 ??

PUSH test.golden.filter filter.fl: 4.12

PUSH Scoped[f] filter.fl: 3.7

PUSH Scoped[l] filter.fl: 3.12

CLOSURE v3 ??

PUSH Cons filter.fl: 4.10

PUSH Scoped[a] filter.fl: 3.10

PUSH v2:clos2 filter.fl: 4.12

=======

# Typechecking revisited

While the "canonical" Peter-Hancock method of typecheck is very general and applies to all functional languages, it fails to meet the needs I have for a number of critical reasons:

* it doesn't allow for good error checking;
* it doesn't support union types directly;
* it uses the lambda calculus as its base rather than HSIE;
* it assumes a "clean" substrate rather than an existing set of symbols;
* it does not make it easy to "assert" believed facts about the type of an expression;
* the code I've written feels unwieldy because of the transition between programming models (functional to imperative).

I propose to remedy all of these in a single sweep of completely rewriting the typechecker.

## What we've got

When we start out to do some typechecking, we have an established set of knowledge:

* a set of known (and named) builtin, compound and union types;
* a set of symbols we have previously checked, are builtin, or are somehow defined in a containing scope, with known types;
* a block of code in HSIE, although it may just be an "E" (as when checking individual method cases);
* a set of previously-bound, named variables, which possibly have "actual" types, but possibly have "to-be-figured-out types" (including mutually recursive symbols, externally scoped symbols and the arguments to a method if we're just looking at an "E");
* a putative "top" type for the function or expression, even if that is just something like "Any->Any" (at the very least it tells us something about the believed arity of the function).

## Error-checking and reporting

Given all the problems I've had with error-checking and reporting, I want to be crystal clear as we go through this process what is "known" and what is not "known" and what the provenance is for any claim that we are making about the types of things.

Specifically, this means I want to clearly distinguish between things that have been established, and how they have been established (e.g. a function has its type declared and recorded from where the first case of the first function is) and things that are not known (such as a variable in an expression).

There is a problem that we have run into any number of times that we need to be really careful of, which is that we tend to think of "arguments to the function" (or more specifically, the variables in the argument patterns) as being a thing, whereas, once we have moved to HSIE, what really matters is the variables we have substituted.

Then we have the issue of how to deal with variable-variable contention, which is what happens if you happen to end up with two variables with unknown types which get restricted to what types they can hold in an incompatible way.

I think what we need to solve all of this is a "chain-of-custody" of each type statement that we make which either says "this is just axiomatic in the current type checking regime" or "this is true because you made this statement in the type checking regime".

Metaphor

Is it possible to replace the metaphor of "types" with the metaphor of some kind of processing engine? In other words, to replace the "unification" of Prolog with something more mechanical that is tree-building? This just occurred to me in the context of what I was just writing about "chain-of-custody". Could we see the operation of type-checking not so much as checking the types, but of building up a "proof of work" of how you could prove the types to be what you claim them to be? Obviously the "can you prove it" bit is the most important output, but we do also want (in the majority of cases) to get back the actual type of the function (to store back in our universe of knowledge), but the set of deductions would also be very interesting.

## Statements

So what kind of statements can we have in our proof-checking environment?

The most obvious ones are axiomatic. I can think of the following examples:

* constants - such as strings and integers, which have builtin types;
* type constructors - such as "Cons", which are defined to have specific function types mapping possible arguments to specific types;
* previously defined functions - in this case, there is a specific name and it has a specific, completely defined type.

There are then derived expressions. I think all of these *must* involve at least one variable. It seems these come in three different varieties. The first variety is "this is asserted to be the case, i.e. already proved":

* variable is pre-bound as part of an external context - for example, if it is the name of another method being recursively defined here;
* variable appears as an argument to the function under test - in this case it will be restricted to the type of that argument (which may well be Any);
* variable used as an argument in a pattern - in this case, it must be restricted to the type accepted by the relevant position in the constructor type.

The second variety is "this is a constraint which must be checked":

* using a variable as an argument to a constructor - it must be of an appropriate (sub) type
* using a variable as an argument to a function - it must be of an appropriate (sub) type

The third variety is "something about closures":

* each closure represents a value resulting from the application of a function to zero or more arguments;
* it therefore ends up with a type;
* this places a requirement on the first argument to be a function of the given arity (or more - it can end up being curried);
* The closure itself is used in other places and is bound to a variable name.

## Method of proceeding

It is to be assumed that we have already done dependency analysis and that all the symbols referenced in the statements to be checked are either previously defined in the existing pool of knowledge or are part of the set of (mutually dependent) statements currently being checked. The set of mutually dependent statements includes all methods which are mutually recursive, together with all the statements which are nested inside other statements and share some of their argument values (I think - the key being that it may require all of these to operate together to deduce the type of one of the shared arguments, and, more importantly, I'm fairly sure this is what I ended up doing last time).

The following description mainly applies to full function analysis using HSIE, including deducing the resulting type of the entire function/method. If you are just asserting that an expression has a given type, you can skip a lot of these steps and just move on to expression asserting.

Each variable we consider (arguments, introduced in patterns and introduced in closures) is considered to have a type, and thus there is at the outset a 1:1 correpsondence of "variables" to type variables. We create a list of knowledge about the type variables we're creating.

The first step is to collect all the functions in the mutual dependency group and rename the variables so that the ones that are common across the functions (i.e. declared in an outer scope that is shared with some functions in an inner scope) are the same while all the others are different. We already do this.

Then we need to go through all the HSI steps and assert that any function arguments are considered "at least" as big as what matches them. For simple arguments, this means "Any", but for constants it is just the type of the constant, and for constructor arguments (in patterns) it is a freshly-instantiated copy of the type (including new generic type variables as appropriate).

Each closure also represents a variable of a given type. We need to look at each closure in turn. Each variable used as an argument needs a constraint that it is "at most" whatever the relevant argument is. Variables used as functions must be observed to be "functions", but I'm not sure exactly what we need to say about their arguments: I think we want to introduce a set of new type arguments (one for each parameter) and apply these rules recursively. We can also assert that the closure variable itself is "at least" the result of applying the function to the set of arguments we know about.

This will lead to a set of variables, each of which has a set of statements either of the form "at least this big" or "no bigger than this" which either form a consistent set which enables us to define the resulting type or contain inconsistencies which we can report as errors.

I think this final step is the same basic unification process that Hancock outlines, it's just that I think it's manageable and I can produce decent error messages of the "the expression X" (or "the variable X") "at location L has inconsistent type usages. It requires that ...". And then list the statements that appear to be in conflict.

The resulting type is the union of the accepted types at each parameter position producing the union of all the result types. In the theoretical realm we can also make the statement that for all other argument inputs it will produce an "Error" output.