6.945 Final Project
Tscheme: Using type inference to automatically complain about the programmer's code

Aaron Graham-Horowitz, Ben Zinberg, Jan Polášek

May 15, 2014

0 The Code

Our code can be found on GitHub at `https://github.com/polasek/tscheme`.

1 Introduction

Tscheme is a static code analyser over a subset of MIT Scheme that aims to reduce the number of bugs in user's code. It uses static type analysis to derive, for each subexpression, a superset of the values that can be taken on by that subexpression. If an empty set is derived, a type error is bound to happen and the error is reported. If all derived supersets are non-empty, the user is allowed to query these supersets to get more information about the code they have written.

There are two common ways of doing static analysis. One approach is static typing, which imposes type rules which are over-approximations of what can happen in correct programs. A good static typing system will provide some sort of type safety - guarantees that will be enforced and that the programmer can rely on. It is for example possible to statically determine that an integer "plus" operator will never be called with a string passed as an argument. The disadvantage of any reasonable type system over a sufficiently powerful language is that there will always be valid programs that do not violate the notion of type safety guaranteed by the type system that nevertheless will not type check. In other words, the type checker will never produce false-positives, but can produce false-negatives (where a "false negative" means that the type checker disapproves of a correct program). As we wanted to have a static analyser that works on an existing Scheme language (or a subset thereof), we decided to go in the other direction, which is a code analyser. Unlike a type checker, a code analyser will never produce false-negatives, but can produce false-positive results.

2 Assumptions

Tscheme works on a functional subset of MIT Scheme. Mutation greatly limits our ability to make inferences about types, as any variable that might be accessible outside the current scope could potentially be changed to any type at almost any time. Rather than work around this complication we require that submitted code be free of side-effects. We make a few other restrictions on the code. We do not allow for procedures with variable number of arguments; this makes it easier to reason about the types of functions. We do not, by default, support all Scheme primitive procedures, however each scheme primitive that we handle is explicitly given a type in our code, and more can easily be added as needed. In a production version of Tscheme we would include all fixed-arity MIT Scheme primitives, but for this early version we include only some common primitives for testing purposes.

Our implementation of type analysis is strict in the requirement of not producing false negatives. To maintain this strictness we do not collect constraints from within the branches of a conditional (we do get constraints from the predicate, however). Because we cannot determine which branches will be accessible, we assume the return value of an `if` or `cond` statement is `*top*`, the type consisting of all possible values.

3 Workflow

Our program is split into two independent parts. First, we have a procedure that takes in code and generates type constraints. These restrict the values that can be taken on by subexpressions of the code that is being analysed. We generate type variables for both variables and expressions in the code and use them to create the right restrictions by using constraints.

Next, we have a procedure that takes a list of constraints and uses them to refine the sets of values that a type variable representing either a subexpression or a code variable can take on. Moreover, it makes sure that no type variable has no value to take on, as

if that is the case, the program is guaranteed to fail (or perhaps, that some procedure within the program can never be called).

4 Types and Constraints

Before elaborating further on our process, we refine our notion of types and constraints.

4.1 Types

A type in Tscheme represents a universe of possible values a variable can take on. Each type variable has an associated Tscheme type, which is represented as a record-type with an attribute for each of the Scheme primitive types (boolean number symbol char string pair list procedure) holding the set of possible values of that type the variable could take on. This set can be `*all*`, meaning all values of the type, `*none*`, meaning the variable cannot have the type, or a `finite-set` containing a list of unique values of the type that the variable can take on. This fine-grained level of detail allows us to precisely type functions like `assq`, which can return a list or the value `#f` but not the other boolean value, `#t`. The type `*top*` has the value `*all*` for every field.

Type variables that can be procedures are handled slightly differently. The procedure field can have the values `*all*` or `*none*`, or it can contain a list of type variables (not types), one for its return value and one for each of its fixed number of arguments. Using type variables for the arguments and return value allows us to concisely represent the potentially recursive nature of procedure types. We use "extended type variables", described below, to reference the type variables contained in a procedure type.

4.2 Constraints

Constraints are also encoded using record types. A constraint has a left side, a right side, and a relation between the left and right side. Not all relations are symmetric, so the left and right sides have different interpretations. The left side is what we call an "extended type variable", which can be a type variable or a reference to the return value or a numbered argument of an extended type variable. Extended type variables allow us to easily make statements about properties of procedures. The right side can be an extended type variable, or it can be a full Tscheme type.

A constraint represents one of three relations between types (or type variables). The simplest relation is "equals", which means the left side and the right side must have exactly the same type. This is a symmetric relation and allows us to reduce our number of type variables through substitution when the right side is an extended type variable. A similar relation is "requires", which tells us the left side must be a subset of the right side. This is not symmetric, and does not add information about the right side, only the left. Lastly the "permits" relation tells us that the variable on the left side is capable of taking on at least one value within type represented on the right side. In other words, the intersection of the left and right sides must be nonempty. "Permits" constraints are not useful for refining types, but can tell us when a type variable has no valid types and report a type error.

The constraint record type also contains metadata about the code that generated the constraint, which helps us give useful error reporting when a constraint is violated. As we refine type variables we track the constraints that were used to make inferences about the type, so that when a user queries a code variable we can give them information about how the type associated with that variable was determined.

5 Constraint generation

The constraint generation component of Tscheme uses the user's code as a source of hints as to what the type of each subexpression should be. If a variable `x` is used as an operand to `+`, we can infer that `x` must be a number, assuming the program has no bugs. Other usages of `x` later in the code (within the same scope) may give additional information;

e.g., if we later find out that `x` is either `5` or `#t`, we can infer that `x` must be `5`. Thus, as more code is read in, information about existing variables accumulates monotonically. (This of course relies heavily on the assumption that there is no mutation.)

5.1 Constraints are deduced from local information

These usage-based hints also have the extremely useful property that the rationale for a hint comes from exactly one statement in the code. This allowed us to separate the constraint generation program almost completely from the unifier. Much more importantly, it allowed us to build a *conceptually* simple specification for the program. The constraint generator makes one pass through the code, accumulating constraints in a database. After the accumulation is complete, its interface with the unifier is simply to hand over the database. Conversely, the unifier's logic has nothing to do with the structure of the user's program; it simply makes deductions using the constraints it is given.

There was a slight lie in the above paragraph, in that some non-local information is used. To see why, consider a code fragment of the form

```
(+ x 7)
```

From this we can deduce that `x` must be a number; but because the constraints concern type variables, the constraint gathering component must know to which type variable `x` is bound in the current scope. Type variables have unique names, so the unifier does not need to worry about scoping, but the user code may have shadowing. Thus the constraint gatherer essentially must pass through the code linearly, keeping track of a scope-aware "environment" data structure (the current mapping of code variables to type variables).

5.2 Implementation

The basic implementation strategy for constraint gathering is simple. A generic operation, `tscheme:process-expr`, takes in an expression and the current mapping of code variables to type variables. It then "processes" the expression: it adds any deduced constraints to a global constraint table, and outputs two things: a type variable representing the value produced by the expression, and an updated mapping of code variables to type variables. Naturally, many of the handlers for `tscheme:process-expr` make recursive calls to `tscheme:process-expr`. The logic as to which constraints are deduced is covered in section 5.3.

It turns out that the most deceptively complicated aspect of constraint generation is keeping track of the mapping of code variables to type variables. Not only does scope have to be properly accounted for (that's easy), but also we must account for the deferred evaluation of bodies of lambda expressions. Thus, for example, when analyzing the expression

```
(define (f) x)
(define (g x) 7)
(define x 3)
```

our system must know that the `x` inside the body of `f` is the same `x` that gets defined to 3, but is not the same `x` as in the body of `g`. This tricky task is accomplished by the `cvmap` data structure, defined in `cvmap.scm`. It is a bit of a hack: a `cvmap` is basically an association list in which lookups of undefined code variables automatically generate a new type variable (rather than producing an error). When the code variable does get defined, its associated type variable is automatically identified with the placeholder type variable from earlier. A detailed explanation of this strategy, as well as how it interacts with scoping, is contained in `cvmap.scm`.

5.3 Rules for constraint generation

We now detail the logic for generating constraints from the user's code. Because each constraint is generated by a single piece of code, the logic is quite easy to explain.

- An expression of the form `(query qname expr)` (which is not itself valid Scheme) represents a query. It behaves exactly like the expression `expr`, except that the type variable bound to the output value is recorded in a global query table, and can be looked up by the name `qname` afterwards.
- A quoted expression or a self-quoting expression (number, string, character, boolean) is bound to a fresh type variable. An "equals" constraint is generated saying that the new type variable's type is a singleton consisting of the literal value.
- A `define` statement consists of a left-hand side (the name) and a right-hand side (the value being named). The right-hand side is a subexpression, which we process by recursively calling `tscheme:process-expr`. The recursive call hands back to us a type variable representing the subexpression; we then bind the left-hand side to this type variable in our `cvmap`. No constraint is generated.
- When we encounter a variable, we simply look up its corresponding type variable in our `cvmap`; no constraint is generated. The return value of a `define` is unspecified, so it doesn't matter what type variable we pass back.
- When we encounter a `begin` expression, we simply process each of the constituent subexpressions in sequence, and hand back the type variable and `cvmap` produced by the last subexpression.
- When we encounter a `lambda` expression, we must do several things at once. A fresh type variable `v` is allocated for the resulting procedure itself, and fresh type variables are allocated for the return value (say `r`) of the procedure as well as each of its arguments (say $\mathtt{a}_1, \ldots, \mathtt{a}_n$). A constraint is generated to link these type variables back to `v` itself, namely,

  $$\mathtt{v\ EQUALS\ (procedure\ r\ a}_1\ \ldots\ \mathtt{a}_n\mathtt{)}$$

  We then recurse into the body. Afterwards, we add a constraint that `r` must be a subtype of the return type of the last expression within the body. Note that, since the body of a `lambda` has its own lexical scope, the `cvmap` returned after processing a `lambda` should be the one we had *before* entering the body (except that the accumulated waiting markers are remembered—see `cvmap.scm`). This way, shadowing is correctly handled: a separate type variable is created for each scope in which a name appears.
- For reasons which we discussed above and will continue to discuss below, `if` expressions are handled very conservatively. We allocate a fresh type variable `v` for the `if` expression and say nothing about `v` except its existence (i.e., `v REQUIRES *top*`). We do not accumulate constraints from the consequent or alternative of the `if`; we also assume that no `define` statements appear within the consequent or alternative, and thus do not modify the `cvmap` based on them. (We do, however, accumulate constraints and updates to the `cvmap` which appear in the predicate.)
- Applications of procedures, i.e. combinations

  $$\mathtt{(operator\ arg}_1\ \ldots\ \mathtt{arg}_n\mathtt{)},$$

  are the most complex part of constraint generation (in concept, but not in code). The actual value of $\mathtt{arg}_i$ must lie within the set of permissible values of the `i`th argument of `operator`, but this can mean different things depending on the syntactic form of $\mathtt{arg}_i$ (which is itself a subexpression). If $\mathtt{arg}_i$ is the subexpression `(f 1 2)`, then the return type of `f` had better contain at least one value which is permissible as the ith operand to `operator`; thus we get the constraint

  `(arg operator i) PERMITS (ret f)`.

  However, if $\mathtt{arg}_i$ is a variable `x`, then every possible value of `x` must be permissible as the `i`th operand to `operator`; thus we get the constraint

  `x REQUIRES (arg operator i)`.

It turns out to be sensible to generate `REQUIRES` constraints for operands which are variables, and `PERMITS` constraints for all other operands. This is a very conservative strategy, but it would take considerable redesign (or a tolerance for false negatives) to open the door for more aggressive behavior.

6 Refining types by using constraints

This section discusses how we process the generated constraints, i.e., how we get as much information as possible about the types of our type variables and how we enforce that every type variable has at least one value it can take on along with enforcing that our "permits" constraints hold. We will discuss the high-level implementation, omitting some implementation details and technicalities.

6.1 Interface specification

The procedure gets a list of constraints as an input. It maintains a map from type variables to types. Initially, all type variables are mapped to `*top*`, meaning that they can take on any value. By processing the constraints, we refine the values the type variables can take on, while keeping track of what constraints have restricted the possible values of each type variable. This information is then used for error reporting and dependency tracking and could be further used by the user.

6.2 Processing a constraint

As discussed in section 4, each constraint contains the left part, which is either a type variable or a reference to a return value or argument of a procedure that can be taken on by the type variable passed. In the latter case, we have to recursively walk the type variable mapping to obtain a type variable, if possible. Note that this can fail if the referenced type variable cannot be a procedure or can only be a procedure that doesn't have enough arguments, in which case a type error is reported. It can also fail because the type variable can be a procedure but we do not have any more information about it at the moment, i.e. it is `*any*`. In that case, we do not have any way of extracting information from this constraint at the given point of computation, therefore it is skipped. The right part of a constraint can also take on a type variable or a procedure reference. If that is the case, we proceed in the same way.

Now we have reduced the problem to the case where we have two type variables, or a type variable and a type. We obtain the current type for type variables from the mapping, and have two types, corresponding to the left and right part of the constraint. By the semantics of the three constraints, we perform the following:

- In the case of equals, we compute the intersection of the two types. We change the mapping of both type variables (or one if only one is present) to this new type.
- In the case of requires, we compute the intersection of the two types and update the mapping from the left type variable to map to this new type.
- In the case of a permits, we compute the intersection of the two types and verify that it is not empty. State of the map does not change in this case, as permits only requires that the intersection of the two types is non-empty, i.e. that there is at least one common value.

6.3 Processing order

The ordering in which constraints are processed can have an effect on the result. For example, if a permits comes before any requires and equals constraints, it will always be true, however if it was run last, it might have found an empty intersection of types.

If all constraints are run repeatedly until the mapping from type variables to types converges or an empty type is derived, the order in which constraints are processed does not matter anymore. This also ensures that all information that can be gained from the constraints is obtained and reflected in the refined types, therefore our code repeatedly enforces all constraints until convergence is reached.

Our algorithm is guaranteed to terminate, as constraints can only contain finite sets of specific values or infinite sets of classes of types, such as numbers or strings, and the way constraints are generated, there won't be any recursive references between type variables. As we only have intersection, it is obvious that there is only a finite number of steps that can be made before an empty type is derived.

6.4 Computing intersection

Computing an intersection of two types tA and tB is trivial for the most part. The only tricky case occurs when we are intersecting two types that could take on a procedure value, where both of them have a specified list of type variables that represent the procedure's arguments and return types. If these two types are not pairwise equal, we might need to enforce the given constraint on the respective pairs of type variables. However, we do this if and only if the intersection of types of tA and tB is otherwise empty.

It is trivial to see why this is necessary if the intersection of the types is otherwise empty, as we do need to know whether the types all give non-empty intersections or not. To see why we cannot do it in any case, consider for example a requires constraint. Say that tA can be either a number or a thunk that returns a string, and tB can be either a number or a thunk that returns a number. Clearly, the intersection of the two types is a number. However, if we proceeded with enforcing the constraint on the procedures, we would derive a contradiction, although it is possible that the set of constraints we were given does not necessarily derive an empty type.

The reason the procedure types behave differently than other type classes is that they are represented with type variables rather than pure types. This allows for a nice flat representation and type variable sharing, however introduces some complexity as described above.

6.5 Substitution

When an equals constraint boils down to setting equality between two different type variables, we compute the intersection of the types, and set the mapping from the left type variable to the new type in our mapping. Rather than doing the same for the second type variable, we substitute the left type variable for the right. The substitutions have to be done everywhere, that is both in mapping and in all constraints. The advantage of doing this is that it essentially removes the current constraint or makes it trivial, and all constraints referencing one of the type variables will now affect both. Making these substitutions thus speeds up convergence, as the number of constraints is effectively reduced by doing so.

Note that substitution can only be done in the case the procedure types of the two types being mapped from the type variables are not both procedure specifications with different type variables for the same reason as above.

7 Development process

We spent several weeks at the beginning of the project simply discussing the nature of the problem and working through code samples by hand. We decided early on to take a code-analysis approach over any sort of strict typing, but it was only after much experimentation that we decided to make a constraint-based implementation. We favored this

method because of its modularity—the process of generating constraints is completely separate from the process of unifying constraints and refining types. We worked together to create the shared interface for generating types and constraints (see `types.scm` and `constraints.scm`), and then were able to divide up the rest of the work. Ben wrote most of `get-constraints.scm` and `rewrite.scm`, while Jan and Aaron worked together on `analyze-constraints.scm`. Considerable supporting code was needed for some of the data structures: Jan and Aaron covered types and Ben covered `cvmaps`. Since finishing the basic implementation we have been continually refining the code and writing tests to find cases it has trouble handling (most of which we've been able to accommodate).

8 Discussion and future work

Tscheme in its current form is best thought of as a framework for type-analysis in Scheme. There are several directions we could expand to make a production-ready system, all of which have their own benefits and costs.

8.1 Branching

One of the limitations of our design is its ability to analyse branches of conditionals. The conservative approach we've taken allows us to say very little about the branches and the value of a conditional. An alternative approach that might be more appropriate for a production system would be to assume any branch could be taken. This would however require an extensive overhaul of the design:

- Each possible computation path would carry a separate set of constraints, and there is a guaranteed error if and only if all branches have contradictory constraint sets. The unifier would therefore have to consider many different sets of constraints (exponential in the number of branching instructions in the user code).
- Our data structures would have to be redesigned. The data structure holding the consraints would have to be hierarchical, rather than a flat table.
- Dynamic bindings could become very difficult to compute.

8.2 Reporting

At present, `tscheme:analyze` reports (upon success) the deduced type of each type variable, along with the constraints that were used to deduce each type. Also returned is a mapping which gives the type variables corresponding to each of the user's queries. In this way, the user can look up the type of each queried expression. (Upon failure, the type variable that produced the contradiction is returned, along with a set of constraints from which the contradiction was derived.)

Each constraint contains the user code that produced it, so that the user can easily find the parts of his code that may be problematic. While this is often quite useful, more sophisticated reporting would be desirable:

- Suppose the user analyzes the code `(define x 4)`. The reported type of `x` is, correctly, the singleton {4}. But the user code used to justify this constraint is simply `4`. We would like the user code to be `(define x 4)`. What we're really saying is that we'd like to see not only reasons that deductions about type variables were made, but also reasons that code variables were bound to type variables. This would require significant reworking of the constraint generator, if not a breaking-down of the divide between constraint generation and unification.

8.3 User Interface

Further work on this project could include a more user-friendly interface. Because we track the locations in user code of the constraints that generated an error, it would be possible in principle to print out the code with problem areas clearly highlighted. An

appealing approach would be to integrate Tscheme with emacs, with a macro allowing users to run a section of code through Tscheme's analyzer and then use the results to highlight the code in place. Emacs integration could also allow the user to select variables to query using emacs markers, rather than writing `query` into the code. Conveniences such as these would be an important part of transitioning Tscheme as an academic project to more consumer-oriented software.

9 Conclusions

Tscheme exemplifies a modular approach to code analysis. The user's code (like all Scheme code) consists of a list of subexpressions, and constraints are generated by iterating linearly through that list (perhaps with recursion into subexpressions). The constraints are then unified by a separate component. Thus far, neither component has had to "worry about" the other's job, but we have seen a few ways in which the system might become more powerful if they did (e.g., justifying the bindings of code variables to type variables). It would be interesting to explore whether the system can be extended while keeping its modular design. In any case, our hope is that a hint-based code analysis system like Tscheme could one day save programmers a significant amount of time and frustration without forcing them to write their code in a different way.