Topics in Compilers

John Toman

December 19, 2011

Introduction

In this paper we evaluate a potential curriculum to be used in upcoming semesters of CMSC430 (Compilers). We present the major material covered in chronological order, and the corresponding assignments. Each of the following sections begins with a general overview of the material covered. A subsection is devoted to teaching methods or other sticking points in the material that may require special attention, although when appropriate smaller recommendations may appear mixed with the explanations of the material covered. Another subsection details the assignment chosen to accompany the material. The choice of supporting techonologies is also evaluated.

Parsing Algorithms

The curriculum began with the topic of parsing algorithms. One of the most common parsing algorithms, LR-parsing was presented, along with a more general overview of bottom-up parsing. An explanation of how LR-parsing worked was provided, along with the problems that can arise when using an LR-parser generator, e.g., shift-reduce conflicts. An example of how the state-action tables worked was also given. The algorithm to generate the state-action tables used in LR-parsing was also briefly touched upon.

Project

The project was a unit calculator, similar to the one accessible through Google search, e.g., a utility where one could type "1 foot in meters" and get the corresponding result (0.3048 meters in this case). As mentioned, the a parser generator dypgen was used for this project (and for all subsequent projects). dypgen is not a LR-parser like ocamlyacc but a GLR parser. A GLR parser differs from other LR-parsers in that it does not suffer from shift reduce conflicts. Instead, in the face of ambiguity it will generate all possible productions for the source grammar from the given input. In practice this feature was never helpful, as any languages invented for the purposes of class projects were not complex enough to suffer from ambiguity. I had never used a parser generator before, but the idea of writing grammar productions and associating them with actions was not very difficult to grasp.

The hardest part of the project was not actually writing the calculator itself but working out how to store units and perform unit conversion. My initial solution was at the time of parsing to convert any values encountered into their corresponding SI values, but also keep the original units. In addition it was necessary to store the exponents of every unit. For example, if the expression `3 ft` was encountered it would be stored as $\langle 0.9144, [ft^1] \rangle$ and the expression `4 ft/s` would be stored as $\langle 1.2192, [ft^1; s^{-1}] \rangle$. It was also enforced that a value could only have one of each type of unit, e.g., a value like `4 ft miles` would be an error. The original units were stored so that after computation the resulting value could be reported in the original units. This greatly simplified storing conversions and performing the computations - conversions only had to be defined from any unit to the principle SI type and back. It also meant that when an arithmetic operation was performed no further conversion was necessary. The numerical values could be combined without further processing assuming that the units of the two values matched.

It was suggested that unit conversions be done lazily instead of at the time of parsing, the rationale being that in computing the expression `3 ft + 4 ft` if lazy conversions were used no conversions would be

required. Even in the case of an expression like `3 ft/min + 4 m/min` only one conversion would be required under the lazy conversion scheme instead of the 4 required under the original scheme. However, this scheme ended up being far too complex to implement. It required defining conversions for between all possible units, and while it is certainly possible to compute the conversion coefficients for any two arbitrary units, the code was extremely complex.

Content Recommendations

One of the biggest troubles I had in understanding the state-action table approach used in LR-parsing was why it worked. One explanation given that helped me greatly was that the states allowed us to have an intermediate state where the parsing algorithm was not immediately sure what production rule to use, and so deferred a decision until later input resolved ambiguity. In teaching LR-parsing, I recommend using an explanation such as, "the states tell us that, given that we've seen these symbols on the input, what grammar productions could we be in, and given what symbol we see next, how does that help us decide what production rules to use for this input."

The explanation of the algorithm for generating the state and action tables was very difficult to understand; however that did not affect my theoretical understanding of LR-parsing. This difficulty in understanding the algorithm as presented leads me to suggest that the algorithm to construct such tables be omitted or touched upon only cursorily. Fortunately, the project used a compiler-compiler tool, so my lack of understanding of the algorithm to generate the LR-parsing tables did not affect my ability to complete the project. In teaching about shift-reduce conflicts, I recommend that a grammar with such a conflict be presented, and perhaps an explanation of why that grammar generated such a conflict. However, this explanation may require too deep an understanding of the process used to create the state-action table.

Finally, I also recommend going further into depth about when LR-parsing will *not* work. It was briefly mentioned that there were grammars not parseable by LR-parsing, but it would illustrative to see an example of such a grammar (although, like with the shift-reduce conflicts mentioned above, understanding why LR-parsing fails on a given grammar may require very in depth knowledge of the algorithm used to generate LR-parsing tables).

Project Recommendations

I cannot recommend using dypgen for future projects. The documentation for dypgen focused too heavily on very advanced features while neglecting to mention several important points about basic usage. For example, dypgen comes with a built in lexer generator in addition to the parser generator. However, nowhere in the documentation is it detailed how to use this lexer generator; I learned to use it via a frustrating process of trial and error combined with combing through example usages provided with dypgen. Some difficulties I encountered in trying to use the lexer generator:

- The regular expression language for matching a single character uses single quotes around the character, whereas matching a string of one or more characters required double quotes. When I attempted to write a regular expression like "ab"+ the dypgen program failed with the opaque error message: "Lexing failed with message: lexing: empty token"
- In order for tokens to be used as values in production rules involving those tokens a lexer action was required after the token definition. Failure to do so results in an error in the place the value of the token is accessed. No mention of this requirement was found in the documentation.

In addition, dypgen suffers from several other major undocumented bugs:

- Attempting to use a string as a source for parsing would occasionally fail with the error message: 'Fatal error: exception Failure("lexing: empty token")'. In my experience it was only possible to use an OCaml in_channel type as the source for parsing
- The resulting parse of input was often not complete, and only produced the smallest possible production from the input grammar. For instance, suppose we have the following simple grammar

$$\begin{array}{rcl} e & ::= & n; e \\ & | & n \end{array}$$

That is, a grammar that accepts a semicolon separated list of integers. On the input "1;2;3" dypgen would return a list containing only 1 as the parse of the input, not the complete list of 1, 2, 3 as would be correct. The only way to fix this was to add some sentinel end of input character that would force the parser to consider the entire input.

As previously mentioned, units presented the greatest challenge in the project. Even under the simplified scheme of converting everything into SI units at the time of parsing and then converting back into the original units after evaluation proved to be very difficult to implement; storing these conversions and performing them were both very difficult. Furthermore, units do not lend themselves to easy parsing. Consider the following expression `3 tons`. `tons` here could be metric tons, long tons, or short tons. Allowing disambiguation by allowing `4 metric tons` to be a value in the language would significantly complicate parsing. Considering this is meant to be a project to introduce students to parsing and the complications that arise when dealing with units I suggest the following when using this project:

- Any unit conversion code, including configuration, be provided as part of the starter project code. This will allow students to focus more on using a parser generator and deciding on internal representations of values, rather than on unit conversions.
- The units that the unit calculator should support be a very small subset of actual units. In addition, supporting multi-word unit names should not be required.

Operational Semantics and Lambda Calculus

Operational semantics were covered briefly in CMSC330, and I independently studied it using the book "Types and Programming Languages". A brief introduction to the syntax used in operational semantics and the difference between small step and large step operational semantics was covered. The method for compiling closures was modeled using operational semantics. In teaching operational semantics, I would stress the concept that the syntax and forms used in operational semantics do not necessary have to follow one style, as this did not become clear to me until later in the semester. As was reiterated many times throughout the semester, I would emphasize that operational semantics are "just math" and need not follow some rigorous form.

Projects

In this portion of the semester, we focused more on evaluating the Lua VM as a compilation target for compilers. To that end, two separate projects were assigned: one that compiled simple arithmetic expressions into Lua VM bytecode, and another that added in support for functions and used the hashtables to simulate closures. I had previously written a bytecode compiler/decompiler in OCaml, so the bit manipulation and binary output required for writing this compiler were familiar to me.

Arithmetic Expressions

Implementing this project required two stages: writing the assembler that generated the bytecode for a series of Lua instructions, and translating the AST of an arithmetic expression into a sequence of instructions. I began by writing the assembler first. This did not prove very difficult as Lua only has 3 types of instructions (only 2 of which were actually used in practice), and the instructions themselves are each exactly 32 bits, which meant that any instruction manipulation was done solely on discrete 32-bit aligned chunks. In writing the assembler I consulted the excellent guide "A No Frills Introduction to Lua 5.1 VM Instructions" which was, despite its name, a very in depth reference for the Lua VM bytecode format. The only major difficulty was that Lua bytecode is not cross-platform; the bytecode for a program compiled by the native luac compiler on a 64-bit computer is different than that generated on a 32-bit computer. The difference arose in having to store the size of a `size_t` and an `int` in the bytecode header. If the value stored in the header did not match the current machine's sizes, the Lua VM would refuse to load the bytecode. Therefore, it was necessary to write two functions in C that used the FFI provided by OCaml to get the size of a `size_t` and an `int` on the current machine.

The compiler portion of the project, that is, the portion that translates arithmetic expressions into instructions, was also very straightforward to implement. I made use of the ability for Lua to use constants directly in certain instructions (such as the add and divide instructions) which could be a potential extra credit opportunity, or a requirement for full credit on the project, as the project itself was not very difficult. I devised a register allocation scheme without any previous experience in the matter. Under this scheme, the parse tree would be walked from the root to the children, at each node reserving a register in which to store the result of evaluating the computation at that node, and then add it to a reserved set. Any register allocated was required to not already be in the reserved set. Terminal nodes, that is constants, required no new register allocations. This was not the optimal solution and would leave registers as allocated after their value was used. A potential data flow project (see below) could be to identify the program points after which the register would no longer be used and de-allocate it.

Hashtable Closures

The second project entailed adding functions to the language supported by the compiler developed in the previous project. The language also supported first class functions and closures and a new syntactic form, the `let` $x = e$ `in` e construct, was added. The Lua VM supports closures natively, but I was specifically instructed to emulate closures by using Lua hashtables to capture the current environment. Conceptually, a closure is a tuple $\langle \lambda(x_1, ..., x_n).e, \Gamma \rangle$, where Γ is the current environment that maps any variables names in scope to their values. Calling a closure is simply invoking the function stored in the first element of the tuple in the environment that is the second element of the tuple. Concretely, this involved adding an environment parameter to every function, which was a hashtable that contained all the mappings of variable names to values. Lua supports first class functions; but any function that appears inside some function must be hoisted out of the body of the function, and stored in the header of the enclosing function. In order to use one of these hoisted functions, a closure object is loaded via a special instruction that identifies the hoisted function via a numeric index. One of the major difficulties encountered in compiling these closures was finding a way to do this hoisting process functionally without using the mutable state provided by OCaml.

When implementing the arithmetic project, one node in the parse tree yielded one instruction in the output. However, with closures this was no longer the case. A function call required:

1. Allocating a register for the Lua closure, and moving the Lua closure out of the closure tuple.
2. Evaluating each of the arguments in order and, if necessary, moving the result of the evaluation into the necessary register for the value to be passed to the function.
3. Moving the stored environment out of the closure tuple and into the final position in the argument list.
4. Calling the Lua closure with the given arguments.

This required a completely different method of compilation from the previous project. By far the most challenging part of the project was the closure allocation. It required hoisting the function, allocating the Lua tuple for the closure, loading the hoisted function into the closure tuple, copying the current environment into a new environment hashtable, adding the current bindings in scope to the environment-capture hashtable, and finally placing the environment has into the closure tuple. Despite this difficulty, providing boilerplate code to the project (for instance, a utility function to generate the instructions necessary for a function call) to let students to focus more on this challenging part may unduly influence a student's code towards some particular implementation which would certainly be undesirable.

Recommendations

Ultimately, the choice of the Lua VM as a target for compilation is very sound. The bytecode format is very simple and extremely well documented. In addition, the tool ChunkSpy, by the same author of the "No Frills" guide is an excellent tool in verifying that the bytecode output by the assembler accurately reflects what the author has in mind. One project implementation point bears mentioning. Creating a closure required copying the current environment and adding the current variables bound in the current scope to this copied environment. In my implementation I used a "standard library" written in Lua that consisted solely of a

function that copied a hashtable. I recommend that in the final assignment this standard library be included with the starter project files. Requiring students to implement the copy function in the compiler would necessitate that students learn to write program loops in Lua assembly, adding extra difficulty to an already difficult project. The method of using this standard library can be found in my reference implementation.

Typing

I had already studied the basics of typing independently from the publication "Types and Programming Languages" so their presentation was not new. Subtyping was also covered, a topic that was unfamiliar to me. The simple type rules developed for simply type checking were modified to allow for subtyping, and the implications of having `int` be a subtype of `float` were demonstrated. However, the pure inference rules for typing a program if subtyping is added do not translate well into an algorithm because such rules are not syntax directed and there is ambiguity at any point as to whether to apply the subtyping rule or to proceed via one of the syntax rules. To remedy this problem, a demonstration of inlining the subtype rules into the function application rule was given.

A special case of subtyping was presented for references which involved side effects (a topic not yet covered during the semester) and the complications they produce. An example to motivate why an `int ref` should not be a subtype of a `float ref` was given, and the theoretical justification was also presented. There was no project associated with this portion of the curriculum.

Recommendations

When presenting types I recommend that the types involved in type checking "by hand", that is with inference rules, are human supplied and need not necessarily be inferrable or annotated into the program. The most unintuitive for me was the subtyping rule for functions. Subtyping in the most general sense allows one to use a value of one type safely where another type was expected. The function subtyping rule reflects this goal of subtyping, but when it was presented in the abstract I had a hard time understanding why the rule resulted in that property. Understanding of the rule only came when I framed the function subtyping rule in terms of objects. One such example that helped me understand this typing rule is produced below.

Function Subtype Example Suppose we defined the subtyping relation for some classes as follows: `Volvo` $\leq$ `Car` $\leq$ `Machine` and the `Car` class had some field called `mpg`, and we had some function with the signature `Car foo(Car)`. If we wanted to replace `foo` with some other function in the following expression `foo(carVariable).mpg` what functions would be safe to swap in? If `bar` had the prototype `Volvo bar(Machine)` then such a replacement would be safe; a `Car` is safe to pass to a function expecting a `Machine` because of subtyping, and because `Volvo` is a subtype of `Car` it is guaranteed to have an `mpg` field; its usage as a return type safe. What we've now illustrated is that `bar` is safe to use where `foo` was used (that is, it is a subtype of `foo`) because of subtyping of the argument and return type.

Type Inference

I began exploring type inference independently during a break in the curriculum when there was no material being covered. Previous treatment of typing either had type annotations or the types were supplied during inferencing. I attempted to derive my own scheme for type inference without consulting any resources. This was an excellent exercise, and gave me experience in deriving type systems. A possible homework assignment would be to require students to give a best effort to derive rigorous rules for type inference. First, non-polymorphic type inference was covered using constraint solving. Then the Hindley-Milner type inference method and the Union-Find data structure were also presented. The motivation for allowing only let-polymorphism was also covered, and the undecidability of System-F style polymorphism was introduced.

Project

The project was to add type inference to the "arithmetic expressions with functions" language developed in the previous project. The most difficult part was implementing the Union-Find data structure required for type inference. Unlike the previous project, a purely functional approach was in feasible in order to implement the optimizations that give the Union-Find data structure an almost linear run time. The implementation itself was only 190 lines, including whitespace and comments. Although difficult to implement, this was the most straightforward projects of the semester.

Recommendations

I can make no recommendation either way on whether or not an implementation of Union-Find should be provided as the starter code. Certainly there is more to the type-inferencing than the Union-Find data structure, but in my implementation the two were very intertwined, and implementing Union-Find gave me a better appreciation of how type-inferencing *really* works from a technical standpoint. In learning type inference methods, there were two major points that I felt needed clarification. It was not immediately obvious why in the following program:
`let` $a = \lambda().\lambda x.x$ `in`
`let` $b = a()$ `in` ...
the identifier b is still a polymorphic function. It should be emphasized that any free type-identifiers at the time of binding in a let construct, whether they come from a function definition *or* application of a polymorphic function, will make the bound function polymorphic. I would also stress that polymorphism is only created via let binding. That is, in the following program:
$(\lambda x.x)$ 4
the function $(\lambda x.x)$, despite being the identity function, is not polymorphic because it is not bound via let.

Data Flow Analysis

The next portion of the curriculum covered data flow analysis. Control flow graphs and the modeling of program flow via these graphs was presented. The idea of progam points and proving facts about various program points motivated data flow analysis. Four kinds of facts were covered: available expressions, liveness, very busy expressions, and reaching information. The Gen-Kill framework for data flow analysis and the worklist algorithm that uses the gen-kill framework was explained. In addition, several auxiliary topics were covered to aid in the teaching of data flow analysis, including the greatest lower bound and the least upper bound operations, lattices, and partial orders. Several examples of the workflow algorithm were walked through to how it worked. There was no project associated with this section of the curriculum, although performing some sort of data flow analysis on Lua bytecode (the very simple nature of the bytecode lends itself to easy transformation into CFGs) would make a good project.

Recommendations

One of the most confusing point for me in learning data flow analysis was actually one of naming. The categories of problems, e.g., "forward must", "backwards may", etc. seemed to be reversed from what one would intuitively expect. For example, backwards must problems used data about what comes after a program point to prove data flow facts at some point, so one would expect that it would be a "forward" problem. I would be sure to stress that the "backwards" and "forwards" refer to in which direction the assumptions flow along the edges in the control flow graph. That is, in a backwards must/may problem, the facts used to reason about some program point flow *backwards* along the edges leading from the current program point, and in forward may/must problem the facts flow *forwards* along the edges leading into the node representing a program point.

I also had trouble in understanding the least upper bound operator and it's lower bound counterpart. The use of the $\leq$ operator in it's expression, i.e. $x \leq x \sqcap y, y \leq x \sqcap y$ (and similarly for the greatest lower bound), implied that this operator only worked on integers or other numbers. It is important to reiterate that the $\leq$ is not necessarily the "less than or equal to" operator that's used to relate numbers, but any

partial order. One such illustration of this concept would be an example involving sets, which does not involve numbers at all. In the original slides to teach data flow analysis, the intersection operation was used as the greatest lower bound operator but its presentation was separated from the initial introduction of the bounds operators that it did not help to illustrate that $\leq$ can be any partial order.

Finally, when presenting the Gen-Kill framework, it is important to stress that Gen and Kill mean different things depending on the type of analysis being done. There is not one single Gen or Kill function, rather the meaning of Gen and Kill changes based on the facts being proven.

Final Project

The curriculum ended with a final project chosen by me. Originally, I planned to develop a type-inferencer that could type-check a program that used arbitrary program strings to invoke methods on objects (such dynamic property lookups are possible in JavaScript, Python, Java, among others). Use of such a feature is not so uncommon; I used it to write a general layout algorithm in JavaScript (which proved to be the inspiration for this project) that worked for both vertical and horizontal layouts. To develop this type-inferencer, a very simple functional object language was developed that allowed property lookups only through constant strings. This language was classless, properties could be added to any object at any time; this approach was chosen because it is similar to how JavaScript works (JavaScript has something like classes called prototypes, but their use is not required to build objects). The idea was once a type inferencer for this language was complete, it would be a matter of extending the inferencer to support property lookups using arbitrary expressions. However, it turned out to be extremely difficult to develop even a type system for this language, much less a type-inferencer. Due to this difficulty, the focus of the final project turned away from it's original goal and instead focused on developing a type system for the language. As the type-system was developed, the pros and cons of any given type system were weighed. For instance, some type systems were created that would have type checked some programs available in the original source language but these systems were ultimately rejected as being too limited.

Project

A simple interpreter for the object language developed to type-check was developed, but it was so simple to implement it does not deserve much comment. Implementation of this interpreter took less than an hour, and despite being an introduction to the use of polymorphic variants in OCaml, had no real education merit. The development of the type system was the real focus of this final project (although not complete, the current version of this type system is included in the appendix).

This development of a type system for an invented language, and the trade offs one must make for type-safety under a given scheme proved to be extremely educational. I strongly recommend that one of the projects (not homeworks) in the class be that students must develop a type system for some given language, and provide an argument as to why it's correct and what trade offs were made in creating the type system.

A Type System for the Object Language

A.1 Language Definition

A.1.1 Language Grammar

The following is the grammar for the language.

e	::=	x	id
	|	n	integers
	|	s	strings
	|	$\{\}$	empty object
	|	`this`	this object
	|	$e[s] = \lambda(x_1, ..., x_n).e$	object bind
	|	$e[s] = e$	object set
	|	$e[s]$	object get
	|	$e[s](e_1, ..., e_n)$	object call

A.1.2 Operational Semantics

We define the following for values in our language:

$$\begin{array}{rcl} v & ::= & \texttt{int} \mid \texttt{str} \mid obj \\ obj & ::= & s \mapsto v_o; obj \mid \emptyset \\ v_o & ::= & v \mid (\lambda(x_1, ..., x_n).e, \Gamma) \end{array}$$

Where Γ is defined as the variable environment, defined as

$$\Gamma ::= x \mapsto v; \Gamma \mid \texttt{this} \mapsto obj; \Gamma \mid \cdot$$

We further define the operation $\Gamma(x)$ to mean the most recent binding of x in Γ iff such a binding exists. The case for $\Gamma(\texttt{this})$ is defined similarly. Our operational semantics are defined thus:

$$\frac{}{\Gamma \vdash n \to \texttt{int}}\ (\textsc{Int}) \qquad \frac{}{\Gamma \vdash s \to \texttt{str}}\ (\textsc{String}) \qquad \frac{}{\Gamma \vdash \{\} \to \emptyset}\ (\textsc{Empty-Obj}) \qquad \frac{\Gamma(x) = v}{\Gamma \vdash x \to v}\ (\textsc{Id}) \qquad \frac{\Gamma(\texttt{this}) = obj}{\Gamma \vdash \texttt{this} \to obj}\ (\textsc{This})$$

$$\frac{e_1 \neq \texttt{this} \qquad \Gamma \vdash e_1 \to obj \qquad \Gamma \vdash e_2 \to v}{\Gamma \vdash e_1[s] = e_2 \to s \mapsto v; obj}\ (\textsc{Obj-Set}) \qquad \frac{\Gamma \vdash e_1 \to obj \qquad obj(s) = v}{\Gamma \vdash e_1[s] \to v}\ (\textsc{Obj-Get})$$

$$\frac{e_1 \neq \texttt{this} \qquad \Gamma \vdash e_1 \to obj}{\Gamma \vdash e_1[s] = \lambda(x_1, ...x_n).e \to s \mapsto (\lambda(x_1, ..., x_n).e, \Gamma); obj}\ (\textsc{Obj-Bind})$$

$$\frac{\begin{array}{c} \Gamma \vdash e_0 \to obj \\ obj(s) = (\lambda(x_1, ..., x_n).e_{body}, \Gamma') \\ \Gamma \vdash e_1 \to v_1 \cdots \Gamma \vdash e_n \to v_n \\ \texttt{this} \mapsto obj; x_1 \mapsto v_1; \cdots; x_n \mapsto v_n; \Gamma' \vdash e_{body} \to v_2 \end{array}}{\Gamma \vdash e_0[s](e_1, ..., e_n) \to v_2}\ (\textsc{Obj-Call})$$

For consistency, evaluation begins with $\Gamma = \texttt{this} \mapsto \emptyset$. Note also that we disallow binding directly on the this object. To allow such a form would allow functions like the following $\lambda().\texttt{this}[foo] = 4$ which is very difficult to type under the type system presented here. In addition we disallow functions of the form $\lambda().\texttt{this}$ which is polymorphic over the type of `this`, which introduces complexity into our type system. We omit this restriction from our inference rules to avoid cluttering the already complex (Obj-Call) rule.

A.2 Types

A.2.1 Type Grammar

We define the following as our type grammar:

$$\begin{array}{rcl} t & ::= & \texttt{int} \mid \texttt{string} \mid m \\ m & ::= & \langle s : t_o \rangle \circ m \mid \emptyset \mid \alpha \\ t_o & ::= & t \mid \ell \\ p & ::= & \beta \mid t \\ \ell & ::= & \forall \alpha.\ell \mid f_p \\ f_p & ::= & \forall \beta.f_p \mid f \\ f & ::= & (p \times \cdots \times p) \xrightarrow{m} p \end{array}$$

We adopt the convention that α are row-variables that range over m, and β are type variables that range over t. The notation $\xrightarrow{m}$ means that a function makes the assumption that `this` has the type m (a more formal explanation is given later). Note that as written, the grammar would allow the type

$$\forall \alpha.() \xrightarrow{\alpha} \texttt{int}$$

Which would allow for a function that was polymorphic over the type of `this` which has no immediately clear correspondence to some expression in our source language. Therefore we adopt the convention that a function may not have any row-variables in it's assumptions about `this`.

A.2.2 Subtyping

We must now define the subtyping relation. In order to subtype objects, we define out subtyping rules to be a relation on t_λ not t. However, the correspondence between t_λ and t should be readily obvious. We define t_λ as follows:

$$\begin{array}{rcl} t_\lambda & ::= & \texttt{int} \mid \texttt{string} \mid m \mid \tau \mid \ell \\ & \mid & (t \times \cdots \times t) \xrightarrow{m} t \\ m & ::= & \langle s : t \rangle \circ m \mid \emptyset \mid \alpha \end{array}$$

where ℓ, and any forms on which is relies, is defined as it was before. Notice the addition of τ, which is some globally unique id. Notice also that we now differentiate between polymorphic functions and non-polymorphic functions in our types, and that functions, both polymorphic and non-polymorphic are first class types. We omit the standard width and depth based subtyping rules here, as well as the subtype rules for `int` and `string`, as they are self-evident. However, the subtyping rules for functinos and τ requires special attention.

(Sub-Poly)

$$\frac{\begin{array}{c} \iota = f[\alpha_1 \mapsto \emptyset] \cdots [\alpha_n \mapsto \emptyset][\beta_1 \mapsto \tau_1] \cdots [\beta_n \mapsto \tau_n] \\ \iota' = f'[\alpha'_1 \mapsto \emptyset] \cdots [\alpha'_n \mapsto \emptyset][\beta'_1 \mapsto \tau_1] \cdots [\beta'_n \mapsto \tau_n] \qquad \tau_1 \cdots \tau_n \ fresh \qquad \iota \leq \iota' \end{array}}{\forall \alpha_1 \cdots \alpha_n . \forall \beta_1 \cdots \beta_n . f \leq \forall \alpha'_1 \cdots \alpha'_n . \forall \beta'_1 \cdots \beta'_n . f'}$$

(Sub-Tau)

$$\frac{\tau' = \tau}{\tau' \leq \tau}$$

(Sub Function)

$$\frac{\begin{array}{c} t'_1 \leq t_1 \cdots t'_n \leq t_n \qquad t_{ret} \leq t'_{ret} \\ m' \leq m \end{array}}{(t_1 \times \cdots \times t_n) \xrightarrow{m} t_{ret} \leq (t'_1 \times \cdots \times t'_n) \xrightarrow{m'} t'_{ret}}$$

Stated informally, a non-polymorphic function a is a subtype of another function a' iff the covariance and contravariance is preserved for the argument types and return types respectively, and if the function a makes assumptions about `this` compatible with those of a'. The polymorphic subtyping simply transforms a polymorphic function type into a regular function type, replacing any type-variables with fresh τ. This replacement of type-variables with τ ensures that two polymorphic functions are apolymorphic in the same way; when two previously polymorphic arguments are related in the subtyping rule (Sub-Poly) we can then

be sure that they were from the same type-variable in the original polymorphic function. Before moving onto typing, we define the following instantiation rule for our polymorphic functions.

$$\text{(Inst)}\quad \frac{(t_1 \times \cdots \times t_n) \xrightarrow{m} t_{ret} \leq f[\alpha_1 \mapsto m_1] \cdots [\alpha_n \mapsto m_n][\beta_1 \mapsto t] \cdots [\beta_n \mapsto t^n]}{\forall \alpha_1 \cdots \alpha_n . \forall \beta_1 \cdots \beta_n . f \preceq (t_1 \times \cdots \times t_n) \xrightarrow{m} t_{ret}}$$

(where $\mapsto$ is capture avoiding substitution). Finally, we adopt the notation $\ell(\texttt{this})$ to denote the `this` assumptions for a lambda type.

A.2.3 Typing Rules

We now move onto our typing relation. The typing relation has a context Γ defined similar to the one above, but it instead maps identifiers to types instead of values.

$$\text{(Int)}\quad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \text{(String)}\quad \frac{}{\Gamma \vdash s : \texttt{string}} \qquad \text{(Empty Obj)}\quad \frac{}{\Gamma \vdash \{\} : \emptyset}$$

$$\text{(This)}\quad \frac{\Gamma(\texttt{this}) : m}{\Gamma \vdash \texttt{this} : m} \qquad \text{(Object Set)}\quad \frac{\Gamma \vdash e_1 : m \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1[s] = e_2 : \langle s : t \rangle \circ m} \qquad \text{(Object Get)}\quad \frac{\Gamma \vdash e_1 : m \qquad m(s) : t}{\Gamma \vdash e_1[s] : t}$$

$$\text{(Object Bind)}\quad \frac{\Gamma \vdash e_1 : m \qquad \lambda(x_1, ..., x_n).e_2 : \ell \qquad m \leq \ell(\texttt{this})}{\Gamma \vdash e_1[s] = \lambda(x_1, ..., x_n).e_2 : \langle s : \ell \rangle \circ m} \qquad \text{(Object Call)}\quad \frac{\Gamma \vdash e_0 : m \qquad m(s) = \ell \qquad \Gamma \vdash e_1 : t_1 \cdots \Gamma \vdash e_n : t_n \qquad \ell \preceq (t_1 \times \cdots \times t_n) \xrightarrow{m} t_{ret}}{\Gamma \vdash e_0[s](e_1, ..., e_n) : t_{ret}}$$