Bottom Up Objects - Static Typing Without Types

Benjamin Chung Paley Li Jan Vitek
Northeastern University
bchung@ccs.neu.edu {pa.li,j.vitek}@neu.edu

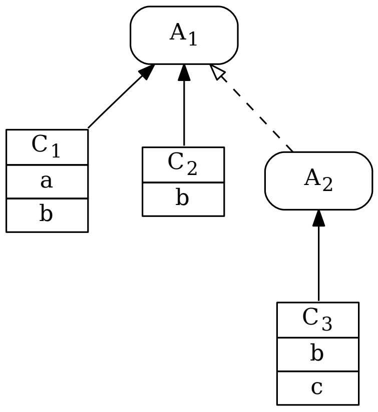

Figure 1. LALALALA

```
1   abstract A1
2   type C1 <: A1 end
3   abstract A2 <: A1
4   type C3 <: A2 end
5
6   function example(arg::C1) print("Hello\n") end
7   function problem(arg::A1)
8     return example(arg)
9   end
10
11  problem(C1()) #prints Hello
12  problem(C3()) #error

ERROR: MethodError: no method matching example(::C3)
Closest candidates are:
  example(::C1)
```

Figure 2. LALALA

1. Introduction

Traditional typed object-oriented languages have a series of common idioms: single dispatch, to figure out which method to use in which situation, interfaces, to abstract over common means of access, and the means to statically ensure that those interfaces are adhered to.

These practices are perfectly suitable for many contexts, as is demonstrated by the success of langauges that have these features, but are not universally applicable, and other languages that do not have any of these features can use the object abstraction while retaining static safety.

In this paper, we will be talking about the Julia programming language. Julia was originally designed as a scientific computing language, in the vein of R or Matlab, and has a number of features designed to support numeric computation and other tasks common in scientific work.

Julia departs from the object oriented norms in all three ways:

- *Multiple dispatch*,using runtime type information to dispatch to the “most specific” method.
- *No explicit classes or interfaces* - but Julia does have subtyping, over structs (which are leaves in the inheritance heirarchy, and act as records) and for abstract types, which are empty labels that allow the construction of heirarchies of types.
- *Dynamic typing*. Julia is dynamically typed, but most Julia programs are heavily type annotated to leverage Julia’s multiple dispatch.

For our purposes, the second deviation is the most interesting, as in Julia abstract types, the only types that can be subtyped from, cannot define an interface that can be safely consumed. An example of a Julia inheritance hierarchy is shown in figure 1, where A_1 and A_2 are abstract types and C_1 through C_3 are structs or concrete types.

Julia programs do have a notion of a functional interface, however, but it does not arise from the language itself. Julia programmers leverage the third property - dynamic typing - to perform operations on an abstract type that cannot be supported by the type itself, following interfaces defined by convention, an idiom that is suggested by the language documentation [1].

The problem with this approach arises in situations like the one shown in figure 2. Here, we expect the function `example` to work on anything of type `A1`, which we then try and call in `problem`. This works fine if we just call it with a `C1`, which has a function `example` defined for it, but if we call it with a `C3` as seen on line 12, then the pictured error message is produced, as `C3` has no method `example`.

We propose that we can check these conventions statically, using the type information that is already in the program. Our system structurally infers the interfaces of abstract types from concrete implementations, then checks that those interfaces are used correctly in client code. Our goal is to be able to detect errors like those encountered in figure 2 before the program is executed, providing programmers with assurance that invocations will succeed.

$$
\begin{aligned}
\tau &::= \texttt{abstract}\, n <: t\,\{\, m(a\ ::\ t,\ \ldots),\ \ldots\ \} \mid \texttt{type}\, n \\
\Gamma &::= \tau,\ \Gamma \mid \cdot \\
t &::= n \mid \texttt{any} \\
d &::= \texttt{abstract}\, n <: t \mid \texttt{type}\, n <: t\,\{f\ ::\ t,\ \ldots\} \\
&\quad \mid m(a\ ::\ t,\ \ldots) = e \\
e &::= x \mid \texttt{new}\, n(e,\ \ldots) \mid m(e,\ \ldots) \mid e.f \mid e.f = e
\end{aligned}
$$

Figure 3. LALALA

$$
\textsc{TAbs}\ \frac{\displaystyle\bigcap_{C<:A} C \bigcap_{A'<:A} A' \cap \top = A}{\texttt{abstract}\, A <: t\ \mathrm{OK}}
\qquad
\textsc{TCon}\ \frac{}{\texttt{type}\, C <: t\,\{f\ ::\ t,\ \ldots\}\ \mathrm{OK}}
$$

References

[1] Julia - interfaces. http://docs.julialang.org/en/release-0.4/manual/interfaces/.