Homework 4: Dynamic Dispatch and Dynamic Types

15-814: Types and Programming Languages
Fall 2015
TA: Evan Cavallo (ecavallo@cs.cmu.edu)

Out: 10/22/15
Due: 11/5/15, 10:30am

1 Representation Independence and the Dispatch Matrix

In class, we discussed viewing a system of methods and classes as a *dispatch matrix*, a term of type

$$\tau_{\mathsf{dm}} \triangleq \prod_{c \in C} \prod_{d \in D} (\tau^c \to \rho_d)$$

where c is the set of classes, d the set of methods, τ^c the instance type for class c, and ρ_d the result type of method d. This matrix admits two alternate, isomorphic representations, as a method vector τ_{mv} or class vector τ_{cv}:

$$\tau_{\mathsf{mv}} \triangleq \prod_{d \in D} \left(\left(\sum_{c \in C} \tau^c \right) \to \rho_d \right) \qquad \tau_{\mathsf{cv}} \triangleq \prod_{c \in C} \left(\tau^c \to \left(\prod_{d \in D} \rho_d \right) \right)$$

If we are working with a system of classes and methods in this way, we may want to introduce an abstract type of objects, which are constructed from instance data and respond to method calls. We can define such a thing using existential types:

$$\exists t_{\mathsf{obj}}. \left\langle \mathtt{new} \hookrightarrow \prod_{c \in C} (\tau^c \to t_{\mathsf{obj}}),\ \mathtt{send} \hookrightarrow \prod_{d \in D} (t_{\mathsf{obj}} \to \rho_d) \right\rangle$$

We'll write τ_{abs} for the type inside the existential. The function `new` takes a class name and instance data for that class and constructs an object, while the function `send` takes an object and a method name and produces an element of the method's result type. Given a dispatch matrix $e_{\mathsf{dm}} : \tau_{\mathsf{dm}}$, there are two natural ways of implementing $\exists t_{\mathsf{obj}}.\tau_{\mathsf{abs}}$. The first, which is analogous to the traditional "abstract data type"-based organization, implements t_{obj} as a sum over class names of instance data.

$$\tau^1_{\mathsf{obj}} \triangleq \sum_{c \in C} \tau^c$$

In this case, `new` simply packages instance data as an element of this sum, while `send` performs a method call by looking up the class and method names in the dispatch matrix.

$$\begin{array}{rcl} \mathtt{new}_1 & \triangleq & \langle \lambda x{:}\tau_c.c \cdot x \rangle_{c \in C} \\ \mathtt{send}_1 & \triangleq & \langle \lambda u{:}\tau^1_{\mathsf{obj}}.\mathtt{case}\ u\ \{x.(e_{\mathsf{dm}} \cdot c \cdot d)(x)\}_{c \in C} \rangle_{d \in D} \end{array}$$

Observe that, in this organization, the type of `send` is the same as the method vector type τ_{mv}. The second implementation, which corresponds to an "object-oriented" system, defines an object as a product of methods.

$$\tau^2_{\mathsf{obj}} \triangleq \prod_{d \in D} \rho_d$$

Here, `new` looks up a class name in the dispatch matrix and returns the tuple of associated methods, while `send` simply extracts the relevant method.

$$\begin{array}{rcl} \mathtt{new}_2 & \triangleq & \langle \lambda x{:}\tau_c.\langle (e_{\mathsf{dm}} \cdot c \cdot d)(x) \rangle_{d \in D} \rangle_{c \in C} \\ \mathtt{send}_2 & \triangleq & \langle \lambda u{:}\tau^2_{\mathsf{obj}}.u \cdot d \rangle_{d \in D} \end{array}$$

In this case, we can observe that `new` has the class vector type τ_{cv}. In sum, we have the following two implementations:

$$\begin{array}{rcl} m_1 & \triangleq & \texttt{pack}\ \tau^1_{\mathsf{obj}}\ \texttt{with}\ \langle \texttt{new} \hookrightarrow \texttt{new}_1, \texttt{send} \hookrightarrow \texttt{send}_1 \rangle\ \texttt{as}\ \exists t_{\mathsf{obj}}.\tau_{\mathsf{abs}} \\ m_2 & \triangleq & \texttt{pack}\ \tau^2_{\mathsf{obj}}\ \texttt{with}\ \langle \texttt{new} \hookrightarrow \texttt{new}_2, \texttt{send} \hookrightarrow \texttt{send}_2 \rangle\ \texttt{as}\ \exists t_{\mathsf{obj}}.\tau_{\mathsf{abs}} \end{array}$$

In this section, we'll start by making precise the correspondence between the class-based and method-based presentations. (For now, we'll assume that the language is terminating, since the parametricity theorem as we've presented it fails to hold in the presence of non-termination.)

Task 1 *Define an isomorphism between the types τ_{mv} and τ_{cv} by giving terms $f : \tau_{\mathsf{mv}} \to \tau_{\mathsf{cv}}$ and $g : \tau_{\mathsf{cv}} \to \tau_{\mathsf{mv}}$. You are not required to prove that the two are mutually inverse (yet).*

To prove that these functions form an isomorphism, we need to show that $g(f((e)) \cong e$ for any $e : \tau_{\mathsf{mv}}$ and that $f(g(e)) \cong e$ for any $e : \tau_{\mathsf{cv}}$. We can do this using logical equivalence, but first we have to define logical equivalence at product and sum types.

- $e_1 \sim_{\prod_{a \in A} \tau_a} e_2$ holds iff $(e_1 \cdot a) \sim_{\tau_a} (e_2 \cdot a)$ holds for all $a \in A$.
- $e_1 \sim_{\sum_{a \in A} \tau_a} e_2$ holds iff there exists $a \in A$ such that $e_1 \mapsto^* a \cdot e_1'$, $e_2 \mapsto^* a \cdot e_2'$, and $e_1' \sim_{\tau_a} e_2'$.

Task 2 *For the functions f and g you defined in Task 1, show that $g(f(e)) \sim_{\tau_{\mathsf{mv}}} e$ for any $e : \tau_{\mathsf{mv}}$. Assume an eager dynamics. You may use parametricity and the fact that $\sim$ is closed under forward and converse evaluation.*

In addition to this isomorphism, we can show that the two implementations m_1 and m_2 of $\exists t_{\mathsf{obj}}.\tau_{\mathsf{abs}}$ are equivalent. For this, we need a definition of logical equivalence at an existential type. As you might expect, this is dual to the definition for universal types.

- $e_1 \sim_{\exists t.\tau} e_2$ iff there exist e_1' and e_2' such that
  - $e_1 \mapsto^* \texttt{pack}\ \tau_1\ \texttt{with}\ e_1'\ \texttt{as}\ \exists t.\tau$,
  - $e_2 \mapsto^* \texttt{pack}\ \tau_2\ \texttt{with}\ e_2'\ \texttt{as}\ \exists t.\tau$,
  - there exists an admissible relation $R : \tau_1 \leftrightarrow \tau_2$ such that $e_1' \sim_\tau e_2'$ when we use R as the definition of $\sim_t$.

(One way to see that this definition is plausible is to unroll the definition of $\sim$ at the System **F** encoding $\exists t.\tau \equiv \forall u.(\forall t.\tau \to u) \to u$.) At this particular type, parametricity and its consequences are often referred to as *representation independence*. The consequences are quite remarkable: in order to show that two elements of an abstract type have the same behavior in any context, one need only give a single admissible relation which serves to relate them!

Task 3 *Show that $m_1 \sim_{\exists t_{\mathsf{obj}}.\tau_{\mathsf{abs}}} m_2$. As part of this proof, you will have to show that a particular relationship holds between (a) $\texttt{new}_1$ and $\texttt{new}_2$ and (b) $\texttt{send}_1$ and $\texttt{send}_2$; you need only give the proof for (a). Also, you are not required to show that any relations you define are admissible, and you may use the fact that the language is terminating without proof.*

2 Self-Reference and Dynamic Dispatch

In this section, we will address a common feature in object-oriented programming languages: self-reference. (In order to do this, we will be returning to a language with recursive types and giving up termination.) It is common in such languages for method definitions to take a "`this`" or "`self`" argument which refers to the object on which the method was called. Our previous

definition of the dispatch matrix fails to account for this, but we can easily rectify it with a small change:

$$\tau'_{\mathsf{dm}} \triangleq \prod_{c \in C} \prod_{d \in D} \forall t_{\mathsf{obj}}.(\tau_{\mathsf{abs}} \rightharpoonup \tau^c \rightharpoonup \rho_d)$$

Here τ_{abs}, which depends on t_{obj}, is as defined in the previous section, and t_{obj} may also appear in the types τ^c and ρ_d. Each entry in the dispatch matrix now takes two additional arguments: first, a type variable t_{obj} which represents the type of objects, and second, the interface to the type t_{obj}, which has type τ_{abs}.

Task 4 *For this task, you'll implement an (admittedly contrived) system where objects are used to represent booleans. Your definition should support the following two classes:*

- *The class* `one` *with* $\tau^{\mathtt{one}} = \mathtt{unit} + \mathtt{unit}$*, which represents true as* $\mathtt{L} \cdot \langle\rangle$ *and false as* $\mathtt{R} \cdot \langle\rangle$*.*
- *The class* `two` *with* $\tau^{\mathtt{two}} = \mathtt{nat}$*, which represents false as* `z` *and true as any successor* $\mathtt{s}(e)$*.*

and the following two methods:

- *The method* `if` *with* $\rho_{\mathtt{if}} = \forall t.t \rightharpoonup t \rightharpoonup t$*, which, when given a type and two arguments of that type, returns its first argument if the boolean is true and its second if the boolean is false.*
- *The method* `not` *with* $\rho_{\mathtt{not}} = t_{\mathsf{obj}}$*, which returns a boolean's negation.*

Task 5 ***Correction 11/2/15:*** *Changed* $\exists t_{\mathsf{obj}}.\tau_{\mathsf{obj}}$ *to* $\exists t_{\mathsf{obj}}.\tau_{\mathsf{abs}}$*.*

Suppose you are given a self-referential dispatch matrix $e_{\mathsf{dm}} : \tau'_{\mathsf{dm}}$*. Give an implementation of the abstract dispatch matrix type* $\exists t_{\mathsf{obj}}.\tau_{\mathsf{abs}}$*. (You will probably want to separate your definition into a few helper terms.) In order to deal with self-reference, use the type* τ `self` *we described in class (and which is covered in PFPL 20.3), which is definable using recursive types. Do not use* `fix`*.*

3 Dynamic Types with Refinements

In this section, we will investigate *type refinements*, which provide one way of reasoning statically about classes in a language with dynamic types (specifically, Hybrid **PCF**, although we will omit sums and products for sake of simplicity). While reminiscent of subtyping, this approach makes a crucial distinction between *types*, which determine the structure of the language, and *refinements*, which serve as static guarantees on already-well-typed terms. Put another way, we distinguish between *structure* (types) and *behavior* (refinements). PFPL 25 describes refinments in detail. We'll begin with a grammar of refinements:

$$\phi ::= \top_\tau \mid \phi \wedge_\tau \phi \mid \mathtt{num}!\phi \mid \mathtt{fun}!\phi \mid \phi \rightharpoonup \phi$$

$\top_\tau$ is the "largest refinement:" it applies to any error-free term of type τ. The refinement $\phi_1 \wedge_\tau \phi_2$ applies to any term which satsfies both ϕ_1 and ϕ_2. The refinement $\mathtt{num}!\phi$ applies to an term of type `dyn` which is statically known to be have class `num` and contains a `nat` satisfying ϕ – proving that terms satisfy this sort of condition is our motivation for introducing refinements. Likewise, $\mathtt{fun}!\phi$ will only apply to classified terms statically known to have class `fun`. The remaining refinement $\phi_1 \rightharpoonup \phi_2$ applies to elements of a function type and will hold if any argument satisfying ϕ_1 produces a result satisfying ϕ_2. We begin making this precise with a judgment $\phi \sqsubseteq \tau$, pronounced "ϕ refines τ", which specifies to which type each refinement

applies:

$$\frac{}{\top_\tau \sqsubseteq \tau} \qquad \frac{\phi_1 \sqsubseteq \tau \quad \phi_2 \sqsubseteq \tau}{\phi_1 \wedge \phi_2 \sqsubseteq \tau} \qquad \frac{\phi \sqsubseteq \texttt{nat}}{\texttt{num!}\phi \sqsubseteq \texttt{dyn}} \qquad \frac{\phi \sqsubseteq \texttt{dyn} \rightharpoonup \texttt{dyn}}{\texttt{fun!}\phi \sqsubseteq \texttt{dyn}} \qquad \frac{\phi_1 \sqsubseteq \tau_1 \quad \phi_2 \sqsubseteq \tau_2}{\phi_1 \rightharpoonup \phi_2 \sqsubseteq \tau_1 \rightharpoonup \tau_2}$$

We can now introduce the judgment $e \in_\tau \phi$, which presupposes $e : \tau$ and $\phi \sqsubseteq \tau$ and expresses that e satisfies the refinement ϕ. We'll begin with the rules that can apply at any type:

$$\frac{}{\Phi, e \in_\tau \phi \vdash e \in_\tau \phi} \qquad \frac{\Phi \vdash e \in_\tau \phi_1 \quad \Phi \vdash e \in_\tau \phi_2}{\Phi \vdash e \in_\tau \phi_1 \wedge \phi_2} \qquad \frac{\Phi, x \in_\tau \phi \vdash e \in_\tau \phi}{\Phi \vdash \texttt{fix}[\tau](x.e) \in_\tau \phi} \qquad \frac{\Phi \vdash e \in_\tau \phi' \quad \phi' \leq_\tau \phi}{\Phi \vdash e \in_\tau \phi}$$

The first is the standard reflexivity rule, and the second is the natural definition of conjunction. The third prescribes that, in order to prove a refinement holds of a fixed point, we assume it holds and show that it is preserved. Finally, the last rule requires some explanation. We must define an additional judgment $\phi' \leq_\tau \phi$ which states that ϕ' is a subrefinement of ϕ. In order to keep this assignment's exposition from becoming interminable, we won't go through the rules for this judgment, which are defined in PFPL 25.1 – it suffices for our purposes to note the following rule:

$$\frac{\phi \sqsubseteq \tau}{\phi \leq_\tau \top_\tau}$$

This just ensures that $\top_\tau$ is the largest refinement. The refinement satisfaction rule above states that any e which satisfies a refinement ϕ also satisfies any refinement which is larger (i.e. weaker) than ϕ. Now, we'll move on to the set of rules for refining classified terms:

$$\frac{\Phi \vdash e \in_{\texttt{nat}} \phi}{\Phi \vdash \texttt{num!}e \in_{\texttt{dyn}} \texttt{num!}\phi} \qquad \frac{\Phi \vdash e \in_{\texttt{dyn}\rightharpoonup\texttt{dyn}} \phi}{\Phi \vdash \texttt{fun!}e \in_{\texttt{dyn}} \texttt{fun!}\phi} \qquad \frac{\Phi \vdash e \in_{\texttt{dyn}} \texttt{num!}\phi}{\Phi \vdash e@\texttt{num} \in_{\texttt{nat}} \phi} \qquad \frac{\Phi \vdash e \in_{\texttt{dyn}} \texttt{fun!}\phi}{\Phi \vdash e@\texttt{fun} \in_{\texttt{dyn}\rightharpoonup\texttt{dyn}} \phi}$$

$$\frac{\Phi \vdash e \in_{\texttt{dyn}} \top_{\texttt{dyn}}}{\Phi \vdash \texttt{num?}e \in_{\texttt{bool}} \top_{\texttt{bool}}} \qquad \frac{\Phi \vdash e \in_{\texttt{dyn}} \top_{\texttt{dyn}}}{\Phi \vdash \texttt{fun?}e \in_{\texttt{bool}} \top_{\texttt{bool}}}$$

The first pair of rules states that a tagged term satisfies the refinement corresponding to its tag, while the second pair states that a coercion is well-refined if the term is known to have the correct class. The third simply states that an instance check is error-free if its argument is. Last of all, we have straightforward rules for nats, booleans, and functions:

$$\frac{}{\Phi \vdash \texttt{z} \in_{\texttt{nat}} \top_{\texttt{nat}}} \qquad \frac{\Phi \vdash e \in_{\texttt{nat}} \top_{\texttt{nat}}}{\Phi \vdash \texttt{s}(e) \in_{\texttt{nat}} \top_{\texttt{nat}}} \qquad \frac{\Phi \vdash e \in_{\texttt{nat}} \top_{\texttt{nat}} \quad \Phi \vdash e_0 \in_\tau \phi \quad \Phi, x \in_{\texttt{nat}} \top_{\texttt{nat}} \vdash e_1 \in_\tau \phi}{\Phi \vdash \texttt{ifz}(e; e_0; x.e_1) \in_\tau \phi}$$

$$\frac{}{\Phi \vdash \texttt{true} \in_{\texttt{bool}} \top_{\texttt{bool}}} \qquad \frac{}{\Phi \vdash \texttt{false} \in_{\texttt{bool}} \top_{\texttt{bool}}} \qquad \frac{\Phi \vdash e \in_{\texttt{bool}} \top_{\texttt{bool}} \quad \Phi \vdash e_t \in_\tau \phi \quad \Phi \vdash e_f \in_\tau \phi}{\Phi \vdash \texttt{if}(e; e_t; e_f) \in_\tau \phi}$$

$$\frac{\Phi, x \in_{\tau_1} \phi_1 \vdash e \in_{\tau_2} \phi_2}{\Phi \vdash \lambda x{:}\tau_1.e \in_{\tau_1\rightharpoonup\tau_2} \phi_1 \rightharpoonup \phi_2} \qquad \frac{\Phi \vdash e \in_{\tau_1\rightharpoonup\tau_2} \phi_1 \rightharpoonup \phi_2 \quad \Phi \vdash e' \in_{\tau_1} \phi_1}{\Phi \vdash e(e') \in_{\tau_2} \phi_2}$$

This completes the set of refinement satisfaction rules.

Task 6 *For the following terms, give the strongest possible refinement derivable with the rules above or determine that none exists. If there is a derivation, describe it, and if not, explain why.*

1. $(\texttt{ifz}(e; \texttt{num!z}; x.\texttt{num!}x))@\texttt{num}$*, assuming* $e \in_{\texttt{nat}} \top_{\texttt{nat}}$*,*
2. $\texttt{ifz}(e; \texttt{num!z}; \_.\texttt{fun!}(\lambda x{:}\texttt{dyn}.x))$*, assuming* $e \in_{\texttt{nat}} \top_{\texttt{nat}}$*,*
3. $(\texttt{ifz}(e; \texttt{num!z}; \_.\texttt{fun!}(\lambda x{:}\texttt{dyn}.x)))@\texttt{num}$*, assuming* $e \in_{\texttt{nat}} \top_{\texttt{nat}}$*,*
4. $\texttt{fix}[x](\texttt{nat}.x)$*.*

Task 7 *Although the expression* $\texttt{ifz}(\texttt{s}(\texttt{z}); \texttt{fun!}(\lambda x{:}\texttt{dyn}.x); x.\texttt{num!}x)$ *clearly evaluates to* $\texttt{num!z}$*, it is not possible to give it the refinement* $\texttt{num!}\top_{\texttt{nat}}$*, because the refinement system fails to notice that the first argument to* ifz *is a successor. We can solve this by introducing new refinements* $\texttt{z} \sqsubseteq \texttt{nat}$ *and* $\texttt{s}(\phi) \sqsubseteq \texttt{nat}$ *(where* $\phi \sqsubseteq \texttt{nat}$*), which are satisfied by zero and successors respectively. (This is quite analogous to the case of* dyn *when we consider that both* $\texttt{dyn} \approx \texttt{nat} + (\texttt{dyn} \rightharpoonup \texttt{dyn})$ *and* $\texttt{nat} \approx \texttt{unit} + \texttt{nat}$ *are essentially sum types!) Define refinement satisfaction rules for* z *and* $\texttt{s}(\phi)$ *such that*

$$\texttt{ifz}(\texttt{s}(\texttt{z}); \texttt{fun!}(\lambda x{:}\texttt{dyn}.x); x.\texttt{num!}x) \in_{\texttt{dyn}} \texttt{num!z}$$

is derivable.

Correction 11/2/15: *Note: by "refinement satisfaction rules for* z *and* $\texttt{s}(\phi)$*" we mean not just rules involving* z *and* $\texttt{s}(e)$ *but also rules for* ifz*.*