Refining Objects
(Preliminary Summary)

Robert Harper
Computer Science Department, Carnegie Mellon University

Rowan Davies
School of Computer Science, University of Western Australia

Abstract

Inspired by Cardelli's pioneering work, many type disciplines for object-oriented programming are based on enrichments of structural type theories with constructs such as subtyping and bounded polymorphism. A principal benefit of such a formulation is that the absence of "message not understood" errors is an immediate corollary of the type safety theorem. A principal drawback is that the resulting type systems tend to be rather complex in order to accommodate the methodology of object-oriented programming.

We consider another approach based on a simple structural type theory enriched with a system of type refinements with which we may express behavioral requirements such as the absence of "message not understood" errors. Ensuring this property is viewed as a verification condition on programs that use dynamic dispatch, which we construe as an abstract type of objects supporting instantiation and messaging operations. At the structural level dynamic dispatch may fail, but at the behavioral level this possibility is precluded.

To validate this approach we give an interpretation of Featherweight Java (**FJ**), a widely-used model of object-oriented programming, that comprises a compilation into dynamic dispatch, and an interpretation of the class table as a system of type refinements. We show that well-typed **FJ** programs translate to well-typed and well-refined programs, from which we deduce the same safety guarantees as are provided by **FJ**. More importantly, the behavioral formulation may be scaled to verify the absence of other behaviors, such as down-cast errors, that are not easily handled using only structural types.

1 Introduction

In fairness, designers of object-oriented languages did not simply "forget" to include properties such as good type systems and good *modularity: the issues are intrinsically more complex than in procedural languages.* - Cardelli (1996)

A great deal of effort has gone into the design of type systems for object-oriented programming. A prime objective, formulated by Cardelli in the 1980's, is to devise type systems for object-oriented languages that preclude "message not understood" errors at run-time (see, for example, Cardelli (1988)). Achieving this objective proved quite challenging, stimulating a large body of research on type systems that could account for a rich variety of programming practices while ensuring that such run-time errors are precluded. Numerous new techniques were introduced, ranging from relatively simple concepts such as subtyping to more advanced concepts such as higher-kinded bounded quantification (see, for example, Bruce et al. (1999) and Fisher and Mitchell (1996)).

These type systems are notoriously complex, to the point that their uptake in practice has been more limited than one might have hoped. Negative results, such as the discovery of unsoundness in extant languages such as Eiffel, have had scant influence on their design or use (see Cook (1989)). Positive results, such as the development of comprehensive theories of objects by Abadi and Cardelli (1996), have had only limited influence on the design of new languages. Although languages such as Modula-3 (Cardelli et al. 1989) have benefited from the theories, newer object-oriented languages, such as Scala (Odersky and Rompf 2014), have only weakly developed theoretical foundations. The situation is in sharp contrast to the direct and continuing influence of type theory on the design and implementation of functional languages, including notable examples such as Standard ML (Milner et al. 1997) and Haskell (Jones 2003), and their more recent evolutes such as Agda (Norell 2008) and Idris (Brady 2013).

It is reasonable to ask why this is the case. One response might be to conclude that the complexity of the type theories involved is an indication that the concepts of object-oriented programming are overly complex, perhaps even conceptually and methodologically suspect. Another reaction might be to argue that type systems are simply not up to the task, and should either be made substantially more powerful (and complicated), or be abandoned entirely (by reversion to untyped languages). But, as Scott (1976) made clear decades ago, untyped languages are uni-typed languages, so there is really no possibility of abandoning types; it is only a matter of how they are to be deployed.

In this paper we propose an alternative approach to typing object-oriented languages that exploits the distinction between *structural*, or *intrinsic*, typing from *behavioral*, or *extrinsic*, typing (Reynolds 1985). Briefly, a structural type system is a context-sensitive grammar that determines what are the well-formed programs, and, via Gentzen's inversion principle, how they are executed. A behavioral type system is a system of predicates or relations (propositional functions), called *type refinements*, or just *refinements* for short, that describe the execution properties of well-typed programs (Freeman and Pfenning 1991; Davies and Pfenning 2000; Davies 2005; Dunfield 2007).

Whereas showing that a program is (structurall) well-typed is usually decidable, showing that a program satisfies a refinement is, by Rice's Theorem, a matter of verification requiring proof. In many cases one can derive efficient and effective decision procedures for certain behaviors, such as the ones we shall consider here, but of course one cannot expect to have fully automatic verification of such conditions.

Since Cardelli's orginal work in the area (Cardelli 1988), the structural approach has drawn the most attention for formulating type disciplines for object-oriented programming. One reason is that structural type disciplines induce behavioral properties of programs from general properties of the language in which they are written. Most importantly, a properly formulated structural type discipline enjoys the *type safety* property (Milner 1978; Wright and Felleisen 1994; Harper 2012), which guarantees that certain forms of run-time errors cannot arise. It makes sense, then, to build on this foundation to derive desirable properties of object-oriented programs, such as the absence of "not understood" errors, from the safety theorem for the type discipline. This goal has usually been achieved by regarding objects as analogous to labelled tuples and messages as analogous to projections, so that type safety ensures that no message may be sent to an object that does not recognize it. Achieving this goal, while ensuring that the type system is not too restrictive, requires concepts such as structural subtyping and bounded quantification (see Abadi and Cardelli (1996) for a thorough discussion of the techniques required). The result is an impressive array of typing concepts for relatively little pay-off. Moreover, from a structural point of view, these concepts are, to an extent, questionable. (For example, width subtyping for tuples relies on the assumption that projections are meaningful independently of the tuple type, a property that is not guaranteed by the universal properties of products, but which can often be arranged to hold in specific implementations.)

The difficulty with the structural approach is that it does not scale well to ensure other desirable properties of programs, such as the absence of "downcast errors," or to the enforcement of behavioral subtyping conditions (Liskov and Wing 1994). To better address these issues we propose another approach to typing object-oriented programs that is based on distinguishing the structural concept of dynamic dispatch (Cook 2009; Aldrich 2013) from the behavioral concept of avoidance of run-time errors. According to our view, dynamic dispatch is simply an application of data abstraction in which an abstract type of objects is equipped with introduction operations that *instantiate* a class with instance data and elimination operations that *message* to invoke a method on an instance. Thus, dynamic dispatch amounts to *heterogeneous programming* in which we have a variety of operations (methods) acting on data of a variety of forms (classes). Such a setup can be envisioned as a *dispatch matrix* whose rows are classes, whose columns are methods, and whose entries determine the behavior of each method on each class. The dispatch matrix gives rise to two equivalent implementations of dynamic dispatch that arise from the duality between sums and products in type theory. This implies that there is no inherent reason to prefer a product-based realization of objects; one may just as well use a sum-based representation. (See Section 3 for further discussion of this point.) This description leaves open what we mean by the behavior of a method on an instance of a class. When well-defined, a method determines a result as a function of the instance data of the object on which it acts. But a method may also be undefined on certain classes, and would, if invoked, incur a "not understood" error. Thus, at the structural level, it is possible for dynamic dispatch to fail, even in a well-typed program, just as it is possible to incur an arithmetic fault in a well-typed numeric computation.

To rule out this possibility we introduce a behavioral type discipline that allows us to express the expectation that certain methods are well-defined on certain classes (or, equivalently, that certain classes admit certain methods as well-defined on their instances). Specifically, we will use a semantic form of *type refinements* of the kind introduced by Freeman and Pfenning (1991) and further developed by Davies and Pfenning (2000); Davies (2005). According to the semantic viewpoint, a type refinement is a predicate (or, more generally, a relation) on a structural type that respects observational equivalence, so that expressions that behave the same way enjoy the same properties. The behavior of dynamic dispatch may be specified by refining the type of the dispatch matrix to express, for example, the expectation that certain methods are well-defined on certain classes. Richer properties of dynamic dispatch may be specified in a similar manner. For example, we may express invariants on the instance data of certain classes (for example, that an integer is always positive) or properties of the results of certain methods (for example, that it return a non-negative number). The critical subsumption property (Cardelli 1988) of type disciplines for object-oriented programming is expressible using logical entailments between refinements, allowing us to support verification in the presence of a class hierarchy.

To assess the viability of our approach, we give an interpretation of Featherweight Java (Igarashi et al. 1999) in terms of the structural formulation of dynamic dispatch to account for its dynamics. We then introduce a system of type refinements derived from the Featherweight Java class table to express the expectation that certain methods are well-defined on certain classes. We then prove that well-typed and well-refined programs cannot incur a "not understood" error, but may still incur a "down-cast error", replicating the guarantees provided by the Featherweight Java type system. Previous work (Davies 2005; Dunfield 2007; Xi and Pfenning 1998) suggests that other conditions, such as absence of down-cast errors or array bounds errors, may be verified in a similar manner. By generalizing from predicates to binary relations it also appears possible to verify equational properties of programs, such as the Liskov-Wing subtyping criterion, in a similar manner. In this respect our approach coheres with the trend to integrate verification of program properties into the development process, allowing us to express a variety of properties of programs that are not easily achievable using purely structural techniques.

2 Background

We will work in a background structural type theory with finite products and sums; function types; general recursive types; predicative polymorphic types; and an error monad with two forms of error. Detailed descriptions of these standard typing constructs may be found in (Harper 2012). We make no use of subtyping, of higher kinds, or of any of the more advanced forms of polymorphism found in the literature (not even impredicativity). Our treatment of the error monad follows the judgmental formulation given by Pfenning and Davies (2001) in which there is a modal separation between *expressions* of a type, which may diverge, but otherwise evaluate to a value of that type, and *commands* of a type, which may incur a run-time error (that is, an uncaught exception) when evaluated. We confine ourselves to functional behavior, and do not consider mutation in this brief account.

The syntactic skeleton of our language, **L**, is given by the following grammar:

Type	τ	$::=$	t	type variable
			$\langle \tau \rangle_{i \in I}$	finite product
			$[\tau_i]_{i \in I}$	finite sum
			$\tau_1 \rightharpoonup \tau_2$	partial function
			$\mu\, t.\tau$	type recursion
			$\forall\, t \,.\, \tau$	type abstraction
			$\tau\, \texttt{cmd}$	encapsulated command
Expression	e	$::=$	x	value variable
			$\texttt{cmd}\, k$	encapsulated command
			$\ldots$	
Command	k	$::=$	$\texttt{ret}\, e$	return a value
			$\texttt{bnd}\, x \leftarrow e \,;\, k$	sequence
			$\texttt{error}$	signal an error
			$\texttt{fail}$	signal a failure

The finite product $\langle \tau \rangle_{i \in I}$ and sum $[\tau_i]_{i \in I}$ types are indexed by a finite set, I, of indices, which may be construed as position numbers or field labels. The finite product and sum types are often written in the display forms $\prod_{i \in I} \tau$ and $\sum_{i \in I} \tau$. Function types, $\tau_1 \rightharpoonup \tau_2$ classify partial (possibly divergent) functions so as to be compatible with general recursive types, $\mu\, t.\tau$. Polymorphic types, $\forall\, t \,.\, \tau$, express (predicative) type abstraction. Existentials are definable from polymorphic types in the usual way, and are sufficient for our purposes. Much of the syntax of expressions is elided for the sake of brevity, but is largely standard.[1]

[1]See, for example, Harper (2012) for more details.

The command type $\tau\,\texttt{cmd}$ represents an error monad formulated in the style of Pfenning and Davies (2001). We consider two forms of error, one that is deemed permissible in a normal execution, and one that is deemed impermissible and should be ruled out by verification. (In Section 4 down-cast errors are considered permissible, and not-understood errors are considered impermissible.) A permissible error is signaled by `error`, and an impermissible error is signaled by `fail`. A non-error return is effected by the command $\texttt{ret}\,e$, where e is a pure expression, rather than another command. The command $\texttt{bnd}\,x \leftarrow e\,;k$ evaluates e to an encapsulated command, evaluates it, possibly incurring a failure or an error, which are propagated, and otherwise passes the return value to the command k. The command type $\tau\,\texttt{cmd}$ is equivalent to the delayed sum type

$$\langle\rangle \rightharpoonup \texttt{[ret} \hookrightarrow \tau, \texttt{error} \hookrightarrow \langle\rangle, \texttt{fail} \hookrightarrow \langle\rangle\texttt{]} \qquad \text{(i.e., equivalent to } 1 \rightharpoonup \tau + 2\text{).}$$

The monadic bind is then an implied three-way case analysis in which the error cases are propagated implicitly, and the return case is handled by the continuation of the bind. This simplifies programming, and is sufficient for our purposes. We note that an error monad does not incur the complications with refinements in the presence of general computational effects considered by Davies and Pfenning (2000) and Dunfield and Pfenning (2003).
The static semantics of **L** is given by three forms of typing judgment:

$\Delta \vdash \tau\ \mathsf{type}$	type formation
$\Gamma \vdash_\Delta e : \tau$	expression typing
$\Gamma \vdash_\Delta k \mathrel{\dot{\sim}} \tau$	command typing

The definitions of these judgments are largely standard, and omitted here. For the sake of clarity, we give the rules for the command types, which are less familiar.

$$\frac{\Gamma \vdash_\Delta k \mathrel{\dot{\sim}} \tau}{\Gamma \vdash_\Delta \texttt{cmd}\,k : \tau\,\texttt{cmd}}$$

$$\frac{}{\Gamma \vdash_\Delta \texttt{error} \mathrel{\dot{\sim}} \tau} \qquad \frac{}{\Gamma \vdash_\Delta \texttt{fail} \mathrel{\dot{\sim}} \tau} \qquad \frac{\Gamma \vdash_\Delta e : \tau}{\Gamma \vdash_\Delta \texttt{ret}\,e \mathrel{\dot{\sim}} \tau}$$

$$\frac{\Gamma \vdash_\Delta e : \tau_1\,\texttt{cmd} \quad \Gamma, x : \tau_1 \vdash_\Delta k \mathrel{\dot{\sim}} \tau_2}{\Gamma \vdash_\Delta \texttt{bnd}\,x \leftarrow e\,;k \mathrel{\dot{\sim}} \tau_2}$$

The dynamic semantics of **L** is given by the following judgments:

$e\ \mathsf{val}$	evaluated expression
$e \mapsto e'$	expression transition
$k\ \mathsf{err}$	run-time error
$k\ \mathsf{fail}$	run-time failure
$k\ \mathsf{final}$	completed computation
$k \mapsto k'$	command transition

The first two define the final states and transition of expression evaluation. The second two define the error states and transition for commands. The expression $\texttt{cmd}\,k$ is a value, regardless of the form of k; it represents a suspended expression that may incur a failure or error when executed. The command $\texttt{ret}\,e$ is fully executed when $e\ \mathsf{val}$; any errors or failures arising within a command are propagated as such.[2] Note that error and failure are observable outcomes of complete programs; these are used in the definition of Kleene equivalence, which states that two pograms either both diverge or have the same observable outcomes.

We now formulate a system of type refinements in the style of Freeman and Pfenning (1991) and Davies (2005). A refinement, ρ, of a type, τ, is, in general, a relation on the elements of τ. Davies and Pfenning considered only unary relations, which is all that are required here, but is useful to consider binary relations to express deeper properties of programs, as in Denney (1998). We depart from Davies and Pfenning, however, in treating refinements semantically, rather than syntactically. In their work refinements are formulated as a syntactic type discipline, with emphasis on decidability of refinement checking. Here we stress the semantics of refinements, leaving mechanical verification as a separate, albeit but important, practical matter.

The syntax of refinements is given as follows:

$$\begin{array}{lllll}
\rho ::= & r & \text{variable} & \rho_1 \rightharpoonup \rho_2 & \text{partial function} \\
 & \top & \text{truth} & \forall\,(t \sqsupseteq \vec{r} : \theta)\,.\,\rho & \text{generic family} \\
 & \bot & \text{falsity} & i \cdot \rho & \text{summand} \\
 & \rho_1 \wedge \rho_2 & \text{conjunction} & \mu\,\vec{r}.\vec{\rho}\ \texttt{in}\ r & \text{recursive} \\
 & \rho_1 \vee \rho_2 & \text{disjunction} & \texttt{ret}\,\rho & \text{normal return} \\
 & \langle \rho \rangle_{i \in I} & \text{product} & \texttt{error} & \text{error} \\
 & & & \texttt{fail} & \text{failure}
\end{array}$$

The logical refinements represent finite conjunctions and disjunctions of properties of any fixed type. The product, function, and command refinements represent the action of their corresponding types on predicates. The summand refinements specify, for a finite sum type, a summand and refinement of its underlying value. The finite sum refinement may be defined by the equation

$$\sum_{i \in I} \rho_i \triangleq \bigvee_{i \in I} (i \cdot \rho_i),$$

the disjunction of all of its summand refinements. The recursive refinement $\mu\,\vec{r}.\vec{\rho}\ \texttt{in}\ r$ specifies one of a set of mutually recursive properties of the recursive unrolling of a value of a recursive type. The command refinements, $\texttt{error}$, $\texttt{fail}$, and $\texttt{ret}\,\rho$, are just summand refinements for the sum type underlying the command refinement, as discussed earlier.

[2]The full definition of the static and dynamic semantics of **L**, and the proof of its type safety, may be found in Harper (2012).

The generic refinement requires further explanation. Following the treatment of abstract refinements for Standard ML modules by Davies (2005), the refinement $\forall\,(t \sqsupseteq \vec{r} : \theta)\ .\ \rho$ refines the polymorphic type $\forall\, t\ .\ \tau$ by introducing a type variable, t, a finite set, $\vec{r}$, of variables refining t, and a finite set of *entailment assumptions*, θ, involving the variables $\vec{r} \sqsubseteq t$. The generic refinement may be seen as the behavioral analogue of bounded quantification (Cardelli and Wegner 1985), but with the freedom to introduce a finite set of abstract refinements satisfying a specified set of entailments.

The *entailment judgment* $\rho_1 \leq_\tau \rho_2$ between two refinements of τ states any closed expression of type τ that satisfies ρ_1 also satisfies ρ_2. Entailment may be seen as the behavioral analogue of structural subtyping. If θ is a finite set of refinement assumptions $\rho_i \leq_\tau \rho_i'$, then the hypothetical judgment $\theta \vdash_\tau \rho \leq_\tau \rho'$ states that whenever the entailments in θ are valid, then so is $\rho \leq_\tau \rho'$. We write Θ for a family of refinement assumptions θ_τ indexed over types τ. This notation often arises when the types τ range over a given set of type variables Δ.

The *expression refinement* judgment has the form

$$x_1 \in_{\tau_1} \rho_1, \ldots, x_n \in_{\tau_n} \rho_n \vdash_\Theta e \in_\tau \rho,$$

where Θ is a family of refinement assumptions for Δ, $\Theta \vdash \rho_i \sqsubseteq \tau_i$ (for each i), $\Theta \vdash_\Delta \rho \sqsubseteq \tau$, and $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_\Delta e : \tau$.

The semantics of the basic refinement judgments is given by assigning to each refinement $\rho \sqsubseteq \tau$ a subset of the closed expressions, modulo observational equivalence, of the type τ, and similarly for closed commands, which are expressions of a distinguished sum type. The details of the construction of such an interpretation are too involved to present here, but the required techniques are well-understood.[3] The semantics of refinements enjoys these properties:

$$\begin{array}{rll}
e \in_\tau \top & \text{iff} & \text{always} \\
e \in_\tau \bot & \text{iff} & \text{never} \\
e \in_\tau \rho_1 \wedge \rho_2 & \text{iff} & e \in_\tau \rho_1 \text{ and } e \in_\tau \rho_2 \\
e \in_\tau \rho_1 \vee \rho_2 & \text{iff} & e \in_\tau \rho_1 \text{ or } e \in_\tau \rho_2 \\
e \in_{\langle \tau \rangle_{i \in I}} \langle \rho \rangle_{i \in I} & \text{iff} & e \cdot i \in_{\tau_i} \rho_i \ (\forall i \in I) \\
e \in_{\tau_1 \rightharpoonup \tau_2} \rho_1 \rightharpoonup \rho_2 & \text{iff} & e_1 \in_{\tau_1} \rho_1 \text{ implies } e(e_1) \in_{\tau_2} \rho_2 \\
i \cdot e_i \in_{[\tau_i]_{i \in I}} i \cdot \rho_i & \text{iff} & e_i \in_{\tau_i} \rho_i \\
\mathtt{fold}(e) \in_{\mu t.\tau} \mu \vec{r}.\vec{\rho} \ \mathtt{in}\ r_i & \text{iff} & e \in_{[\mu t.\tau/t]\tau} [\mu \vec{r}.\vec{\rho} \ \mathtt{in}\ r_1/r_1, \ldots, \mu \vec{r}.\vec{\rho} \ \mathtt{in}\ r_n/r_n]\rho_i \\
e \in_{\forall t.\tau} \forall (t \sqsupseteq \vec{r} : \theta) \,.\, \rho & \text{iff} & e[\sigma] \in_{[\sigma/t]\tau} [\vec{\rho}/\vec{r}]\rho \quad \text{for all} \quad \sigma, \vec{\rho} \sqsubseteq \sigma \textit{ sat. } \theta \\
\mathtt{cmd}\, k \in_{\tau\, \mathtt{cmd}} \rho & \text{iff} & k \in_\tau \rho \\
\mathtt{ret}\, e \in_\tau \mathtt{ret}\, \rho & \text{iff} & e \in_\tau \rho \\
\mathtt{error} \in_\tau \mathtt{error} & \text{iff} & \text{always} \\
\mathtt{fail} \in_\tau \mathtt{fail} & \text{iff} & \text{always}
\end{array}$$

In the clause for refinements of $\forall t \,.\, \tau$ we quantify over refinements $\vec{\rho} = \{\rho_1, \ldots, \rho_n\}$ of the (monomorphic) type σ such that the entailments $[\vec{\rho}/\vec{r}]\theta$ over σ are all valid. Generally, an entailment $\rho_1 \leq_\tau \rho_2$ is valid iff whenever $e \in_\tau \rho_1$, then $e \in_\tau \rho_2$, and this extends to sets of entailments conjunctively.

3 Dynamic Dispatch

3.1 Structural Typing

Consider a system defining a finite set, M, of methods acting on data objects classified by a finite set, C, of classes. Associated to each class $c \in C$ is a type τ^c, called the *instance type* of c, the type of c, that classifies the instance data of that class. Associated to each method $m \in M$ is a type τ_m, called the *result type* of m, that classifies the result of that method when applied to some data object. Such a system may be concisely described as an element of the type

$$\tau_{\mathsf{het}} \triangleq (\sum_{c \in C} \tau^c) \rightharpoonup (\prod_{m \in M} \tau_m)$$

parameterized by the choice of classes and methods and their associated instance and result types. It describes a collection of methods each acting on data of one of a collection of classes, which is an instance of the general concept of heterogeneous programming available in any language with products and sums.

[3]The main difficulty is with recursive types, for which see Pitts (1996); Crary and Harper (2007); Harper (2012)). Using only predicative polymorphism considerably simplifies the construction. The refinements of the concrete sum monad for errors are interpreted the same as refinements for functions returning sums, similar to a simple, unary version of the PER semantics for monadic refinements for exceptions given by Benton and Buchlovsky (2007).

By a de Morgan-type duality there is an isomorphism between τ_{het} and the type τ_{dm} defined by the equation

$$\tau_{\mathsf{dm}} \triangleq \prod_{c \in C} \prod_{m \in M} (\tau^c \rightharpoonup \tau_m).$$

The type τ_{dm} describes a *dispatch matrix* of dimension $|C| \times |M|$, with rows indexed by classes and columns indexed by methods. The entry, e^c_m, of the dispatch matrix defines the *behavior* of method m on instances of c as a function of type $\tau^c_m \triangleq \tau^c \rightharpoonup \tau_m$ mapping τ^c, the instance type of c, to τ_m, the result type of m. The class c and method m may be thought of as the *coordinates* of the behavior of method m on instances of class c.

Any matrix may be seen as a row of columns or a column of rows. In the case each row $c \in C$ of the dispatch matrix determines the behavior of methods M on instances of the class c. Thus the dispatch matrix may be seen as C-indexed column of methods acting on the instance data of c:

$$\tau_{\mathsf{dm}} \cong \prod_{c \in C} (\tau^c \rightharpoonup (\prod_{m \in M} \tau_m)).$$

Dually, each column $m \in M$ of the dispatch matrix determines the behavior of m on the instances of each of the classes C. Thus the dispatch matrix may also be seen as an M-indexed column of results for each possible instance:

$$\tau_{\mathsf{dm}} \cong \prod_{m \in M} (\sum_{c \in C} \tau^c) \rightharpoonup \tau_m.$$

In view of these isomorphisms neither organization can be seen as more significant than the other. They are, rather, equivalent descriptions of the information encoded in the dispatch matrix.

Dynamic dispatch is an implementation of the abstract type

$$\tau_{\mathsf{dd}} \triangleq \exists(t_{\mathsf{obj}} . \langle \mathtt{new} \hookrightarrow \prod_{c \in C} \tau^c \rightharpoonup t_{\mathsf{obj}}, \mathtt{snd} \hookrightarrow \prod_{m \in M} t_{\mathsf{obj}} \rightharpoonup \tau_m \rangle),$$

which specifies a type, t_{obj}, of objects on which are defined two families of operations, *instantiation* and *messaging*, which are, respectively, the introductory and eliminatory forms of the object type. The intended behavior is that sending a message m to an instance of class c engenders the behavior given by the dispatch matrix with coordinates c and m. Clients of this package are equipped with instantiation and messaging operations

$$\frac{\Gamma \vdash_\Delta e : \tau^c}{\Gamma \vdash_\Delta \mathtt{new}[c](e) : t_{\mathsf{obj}}} \qquad \frac{\Gamma \vdash_\Delta e : t_{\mathsf{obj}}}{\Gamma \vdash_\Delta \mathtt{snd}[m](e) : \tau_m}$$

Given a dispatch matrix, e_{dm}, we may implement dynamic dispatch τ_{dd} in two equivalent ways, by defining a representation type, τ_{obj}, and an associated *class*, or *constructor, vector*, e_{cv}, of type

$$\tau_{\mathsf{cv}}(\tau_{\mathsf{obj}}) \triangleq \prod_{c \in C} (\tau^c \rightharpoonup \tau_{\mathsf{obj}}),$$

and a *method*, or *message, vector*, e_{mv}, of type

$$\tau_{\mathsf{mv}}(\tau_{\mathsf{obj}}) \triangleq \prod_{m \in M} (\tau_{\mathsf{obj}} \rightharpoonup \tau_m).$$

We will consider two equivalent implementations of the dynamic dispatch abstraction. The *method-based*, or *sum*, form of dynamic dispatch is given by the following definitions:

$$\begin{aligned}
\tau^{\Sigma}_{\mathsf{obj}} &\triangleq \sum_{c \in C} \tau^c \\
e_{\mathsf{cv}} &\triangleq \langle c \hookrightarrow \lambda\,(x : \tau^c)\ [c \hookrightarrow x]\,\rangle_{c \in C} \\
e_{\mathsf{mv}} &\triangleq \langle m \hookrightarrow \lambda\,(\mathit{this} : \tau^{\Sigma}_{\mathsf{obj}})\ \mathtt{case}\ \mathit{this}\,\{\,[c \hookrightarrow x] \Rightarrow e_{\mathsf{dm}} \cdot c \cdot m(x)\,\}_{c \in C}\rangle_{m \in M}.
\end{aligned}$$

The *class-based*, or *product*, form of dynamic dispatch is given by the following definitions:

$$\begin{aligned}
\tau^{\Pi}_{\mathsf{obj}} &\triangleq \prod_{m \in M} \tau_m \\
e_{\mathsf{cv}} &\triangleq \langle c \hookrightarrow \lambda\,(x : \tau^c)\ \langle m \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot m(x)\rangle_{m \in M}\rangle_{c \in C} \\
e_{\mathsf{mv}} &\triangleq \langle m \hookrightarrow \lambda\,(x : \tau^{\Pi}_{\mathsf{obj}})\ x \cdot m\rangle_{m \in M}.
\end{aligned}$$

For either choice of implementation the instantiation and messaging operations behave by deferral to the constructor and messaging vectors, respectively:

$$\begin{aligned}
\mathtt{new}[c](e) &\mapsto^* (e_{\mathsf{cv}} \cdot c)(e) \\
\mathtt{snd}[m](e) &\mapsto^* (e_{\mathsf{mv}} \cdot m)(e),
\end{aligned}$$

whenever e is a value of appropriate type. Then, by construction, we have in either case that

$$\mathtt{snd}[m](\mathtt{new}[c](e)) \mapsto^* (e_{\mathsf{dm}} \cdot m \cdot c)(e),$$

again under the condition that e is a value. This property may be seen as characterizing dynamic dispatch (Igarashi et al. 1999) in that sending a message m to an instance of class c engenders the behavior assigned to m on c by the dispatch matrix.

This basic model of dynamic dispatch may be elaborated to account for several forms of self-reference found in object-oriented languages:

1. Any method may call any other, including itself.
2. Any class may create an instance of any other, including itself.
3. The instance type of a class may involve any object.
4. The result type of a method may involve any object.

Scaling up to allow for these behaviors is largely a matter of generalizing the type τ_{dm}, choosing τ_{obj} to be a recursive type, and making corresponding changes to the class and method vectors, based on the choice of τ_{obj}. The details of the construction can be found in Harper (2014), but may be briefly summarized as follows.

The types of the components of the dispatch matrix must be changed so that they have access to the class vector (for creating new instances) and the method vector (for sending messages to instances). Moreover, the instance type of each class and the result type of each method may involve instances created in this manner. Thus, the components of the dispatch matrix are given the (predicative) polymorphic type

$$\tau_m^c \triangleq \forall\, t_{\mathsf{obj}}\,.\, \tau_{\mathsf{cv}}(t_{\mathsf{obj}}) \rightharpoonup \tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \rightharpoonup \tau^c(t_{\mathsf{obj}}) \rightharpoonup \tau_m(t_{\mathsf{obj}}).$$

The type variable, t_{obj}, is the abstract type of objects with which the behaviors interact via the class- and method vectors.behaviors provided as arguments.

In the method-based (sum) form the type τ_{obj} of objects is defined by the equation

$$\tau_{\mathsf{obj}}^{\Sigma} \triangleq \mu\, t_{\mathsf{obj}}. \sum_{c \in C} \tau^c(t_{\mathsf{obj}}),$$

whereas in the class-based (product) form the type τ_{obj} is defined by the equation

$$\tau_{\mathsf{obj}}^{\Pi} \triangleq \mu\, t_{\mathsf{obj}}. \prod_{m \in M} \tau_m(t_{\mathsf{obj}}).$$

The implementations of the method and class vectors in terms of the dispatch matrix are slightly more involved than before, because the object types are recursive (requiring folding and unfolding operations), and either the method vector (in the sum form) or the class vector (in the product form) must be self-referential using standard fixed-point operations.

Finally, we observe that it is not necessary for every method to be meaningfully defined on every class of object. More precisely, an ill-defined situation may be defined as one that signals a run-time error corresponding to the "message not understood" error described in the introduction to this paper. This amounts to choosing τ_m, the result type of method m, to admit the possibility of a run-time fault, which may be accomplished using the error monad described in Section 2. Once this possiblity is allowed, it becomes important to specify and verify that certain method and class combinations are sensible, which we view as a behavioral, rather than structural, property of a program.

3.2 Behavioral Typing

As we have seen in the preceding section, dynamic dispatch is a form of heterogeneous programming in which the behavior of a collection of methods is defined on the instances of a collection of classes. In some cases the behavior is to give rise to a "not understood" error, reflecting that the particular combination is ill-defined. The expectation that a method m be defined on every instance of a class c is not inherent in the idea of dynamic dispatch, but is rather a *methodological consideration* imposed from the outside, much as one might insist as a matter of methodology that other forms of run-time fault are to be precluded. Indeed, following Cardelli's principle, one might say that what makes dynamic dispatch, a mode of use of recursive products and sums, be "object-oriented" is just that such expectations are stated and and enforced for each program (for example, by decalarations that form a "class table" for a program). More generally, one may wish to enforce many other methodological conditions, such as absence of "down-cast" errors, or avoidance of "bound check" errors, not all of which can be anticipated in a particular structural type system.

In Section 4 we will carry out a full-scale verification of the absence of "not understood" messages for an interpretation of **FJ** as an application of dynamic dispatch. Here we outline the general approach to verification of properties of dynamic dispatch using type refinements. For the sake of clarity , we first consider the non-self-referential case of dynamic dispatch; this makes it easier to explain the generalization to admit self-reference. To carry out a verification of the properties of dynamic dispatch involves the following ingredients:

1. A family of refinements $\rho^c_m \sqsubseteq \tau^c_m$, which constrains the behavior of the entries of the dispatch matrix. This family determines a refinement $\rho_{\mathsf{dm}} \sqsubseteq \tau_{\mathsf{dm}}$ given by
$$\rho_{\mathsf{dm}} \triangleq \prod_{c \in C} \prod_{m \in M} \rho^c_m \sqsubseteq \prod_{c \in C} \prod_{m \in M} \tau^c_m.$$

2. A family of refinements $\rho^c_{\mathsf{obj}} \sqsubseteq \tau_{\mathsf{obj}}$, for each $c \in C$, and $\rho^m_{\mathsf{obj}} \sqsubseteq \tau_{\mathsf{obj}}$, for each $m \in M$. In Section 4 we will choose the refinement ρ^c_{obj} to express that an object is an instance of class c, and the refinement ρ^m_{obj} to express that an object understands method m.

3. A refinement $\rho^c \sqsubseteq \tau^c$ characterizing the instance data of class c. The instance refinement determines a refinement of the class vector type given by
$$\rho_{\mathsf{cv}} \triangleq \prod_{c \in C} (\rho^c \rightharpoonup \rho^c_{\mathsf{obj}}) \sqsubseteq \prod_{c \in C} (\tau^c \rightharpoonup \tau_{\mathsf{obj}}).$$
The refinement ρ_{cv} states that if the instance data satisfies ρ^c, then the resulting instance will be an object that satisfies ρ^c_{obj}.

4. A refinement $\rho_m \sqsubseteq \tau_m$ characterizing the result of method m. The result refinement determines a refinement of the method vector type given by
$$\rho_{\mathsf{mv}} \triangleq \prod_{m \in M} (\rho^m_{\mathsf{obj}} \rightharpoonup \rho_m) \sqsubseteq \prod_{m \in M} (\tau_{\mathsf{obj}} \rightharpoonup \tau_m).$$
The refinement ρ_{mv} states that if an object satisfies ρ^m_{obj}, then the result of method m will satisfy ρ_m.

5. Because τ^c_m is $\tau^c \rightharpoonup \tau_m$, the refinement ρ^c_m must satisfy the entailment $\rho^c_m \leq \rho^c \rightharpoonup \rho_m$ so that if ρ^c_m holds for matrix entry e^c_m, instances satisfying ρ^c are mapped to results satisfying ρ_m.

These choices determine verification conditions that ensure that dynamic dispatch is well-behaved. We must ensure that $e_{\mathsf{dm}} \in \rho_{\mathsf{dm}}$, which is to say that $e^c_m \in \rho^c_m$ for each behavior e^c_m, and then we must show $e_{\mathsf{cv}} \in \rho_{\mathsf{cv}}$ and that $e_{\mathsf{mv}} \in \rho_{\mathsf{mv}}$, making use of this fact. In sum form the method vector condition follows directly from the fact that $e_{\mathsf{dm}} \in \rho_{\mathsf{dm}}$, but the class vector condition must be checked for the choice of ρ^c and ρ^c_{obj}. In product form the dual situation obtains: the class vector condition follows from the verification of the dispatch matrix, and the method vector condition must be verified for the choice of ρ^m_{obj} and ρ_m.

These conditions ensure that dynamic dispatch satisfies the following properties:

1. if $e \in \rho^c$, then $\texttt{new[}c\texttt{](}e\texttt{)} \in \rho^c_{\mathsf{obj}}$, and
2. if $e \in \rho^m_{\mathsf{obj}}$, then $\texttt{snd[}m\texttt{](}e\texttt{)} \in \rho_m$.

In the case that τ_m is a command type $\tau'_m\ \texttt{cmd}$, indicating that method m may fail when invoked, then some additional conditions are required to ensure that "message not understood" errors are avoided. Specifically, if instances of c are to admit method m, then we require the following conditions:

1. Failure is not an option: $\rho^c_m \leq \rho^c \rightharpoonup \texttt{ret}\,\rho'_m \vee \texttt{error}$, for some ρ'_m such that $\rho'_m \sqsubseteq \tau'_m$.
2. Any object satisfying ρ^c_{obj} must satisfy ρ^m_{obj}: $\rho^c_{\mathsf{obj}} \leq \rho^m_{\mathsf{obj}}$.

These further conditions ensure that if $e \in \rho^c$, then

$$\texttt{snd[}m\texttt{](new[}c\texttt{](}e\texttt{))} \in \texttt{ret}\,\rho'_m \vee \texttt{error},$$

which is to say that sending m to an instance of c cannot fail.

The self-referential case is handled similarly, with some additional complications arising because the entries in the dispatch matrix are polymorphic in the object type and abstracted with respect to the class and method vectors. The ingredients are as follows:

1. As before, a family of refinements $\rho^c_m \sqsubseteq \tau^c_m$ characterizing the behavior of method m on instances of class c as specified by the dispatch matrix.
2. As before, a family of refinements, $\vec{\rho}$, consisting of refinements $\rho^c_{\mathsf{obj}} \sqsubseteq \tau_{\mathsf{obj}}$, for each $c \in C$, and $\rho^m_{\mathsf{obj}} \sqsubseteq \tau_{\mathsf{obj}}$, for each $m \in M$.

3. Variable refinements $\vec{r}$ consisting of refinements r^c and r_m of the abstract object type t_{obj} for each c and m. These are to be thought of as abstract correlates of the refinements ρ^c and ρ_m of τ_{obj} that will instantiate them when the dispatch implementation is chosen. The refinement variables $\vec{r}$ are governed by a finite set of entailment assumptions, θ, that must be true when τ_{obj} instantiates t_{obj} and $\vec{\rho}$ instantiates $\vec{r}$.

4. As before, instance and result refinements, stated parametrically in t_{obj} and $\vec{r} \sqsubseteq t_{\mathsf{obj}}$, and object refinements for each class and method, also parametrically in the same variables.

5. We require that

$$\rho^c_m \leq \forall\, (t_{\mathsf{obj}} \sqsupseteq \vec{r} : \theta)\, .\, \rho_{\mathsf{cv}}(\vec{r}) \rightharpoonup \rho_{\mathsf{mv}}(\vec{r}) \rightharpoonup \rho^c(\vec{r}) \rightharpoonup \rho_m(\vec{r}),$$

   where $\vec{r}$ and θ are the refinement variables and their governing entailments described above.

The last requirement ensures that e^c_m satisfies the instantiation of the polymorphic refinement

$$\rho_{\mathsf{cv}}(\vec{\rho}) \rightharpoonup \rho_{\mathsf{mv}}(\vec{\rho}) \rightharpoonup \rho^c(\vec{\rho}) \rightharpoonup \rho_m(\vec{\rho}),$$

where τ_{obj} is the object type refined by the refinements $\vec{\rho}$ specified above.

A detailed example is given in the next section in which we give an interpretation of **FJ** into **L**, and use refinements to state and prove that "not understood" errors are precluded in well-refined programs.

4 Refining Featherweight Java

4.1 Overview

To demonstrate the suggested separation of structural from behavioral typing, we give a relatively straightforward translation of Featherweight Java (Igarashi et al. 1999) into **L**, then equip it with a system of refinements that ensures that "message not understood" failures cannot arise in a well-refined program. End-to-end we achieve the same safety guarantees as were ensured by the original formulation; our goal here is to show that the proposed reorganization is adequate to achieve the same ends. But, as we shall outline in Section 5, the separation permits consideration of significantly more elaborate verifications than are feasible by increasing the complexity of the structural type system that determines the operational semantics of the language.

The main idea is very simple, particularly if we (temporarily) ignore the self-referential aspects of **FJ** (to which we shall return momentarily). The key step is to translate the **FJ** class table into a dispatch matrix whose entries are commands that either return the behavior of method m on class c when it is defined by the class table, or signal a failure to indicate that it is not defined.

The main idea of the verification hinges on the definition of two forms of refinement particular to the problem at hand, $\mathtt{inst}[c]$ and $\mathtt{recog}[m]$, which refine the type τ_{obj}, regardless of whether it is chosen to be of sum or product form. Informally, an object $o : \tau_{\mathsf{obj}}$ is an instance of class c is defined *semantically* to mean that for every method m associated to class c in the **FJ** class table, the object o does not fail when method m is invoked on it. Notice that o is not required to have arisen from the constructor of class c, but could be any object that behaves in the way that such an object would (that is, it could be an instance of a subclass of c). This semantic instance property is certainly not decidable, but this is not relevant to our purposes. Similarly, an object $o : \tau_{\mathsf{obj}}$ recognizes the method m if any instance of any class declared to have m in the class table does not fail when sent message m. This, too, is not decidable, but this is not relevant for our purposes. What is relevant is that the semantic definition of $\mathtt{inst}[c]$ is not defined by declaration, and does not reflect the history of how the object was created, but is instead a description of its behavior when executed. This ensures that the subsumption principle of **FJ** is validated under our interpretation.

Following the methodology outlined in Section 3, we will set up a system of refinements that ensures that no "message not understood" errors can arise. The instance refinement, $\rho^c \sqsubseteq \tau^c$, is chosen as the product of the class types declared for the instance variables $\prod_{c f \in F^c} \rho^c_{\mathsf{obj}}$. The result refinement ρ_m is chosen to be

$$\mathtt{ret}\,((\prod_{c\, i \in \overline{\mathsf{arg}}_m} \rho^c_{\mathsf{obj}}) \rightharpoonup (\mathtt{ret}\,\rho^{\mathsf{res}_m}_{\mathsf{obj}} \vee \mathtt{error}))$$

expressing the typing conditions augmented with the possibility of a "downcast" error as outcome of the method body. In the case that the class table does not associate m with c, we instead choose ${\rho^c_m}'$ as

$$\mathtt{fail} \sqsubseteq \tau_m$$

reflecting that it is a "not understood" message. In either case we have

$$\rho^c \rightharpoonup \rho_m \sqsubseteq \tau^c \rightharpoonup \tau_m.$$

The refinements ρ^c_{obj} are chosen to be $\mathtt{inst}[c]$ for each $c \in C$, and the refinements ρ^m_{obj} are chosen to be $\mathtt{recog}[m]$ for each m in M. These choices ensure that the class vector entry at c creates objects that are, semantically, instances of c, and that the method vector entry at m delivers a non-error result when applied to the instance data of a class for which it is defined. We note that if m occurs in the class table entry for c, then $\mathtt{inst}[c] \leq \mathtt{recog}[m]$, which states that an instance of c admits the message m, as would be expected. Similarly, **FJ** has the property that if $c <: c'$ then every method m is the class table entry for c if it is in the class table entry for c', thus $\mathtt{inst}[c] \leq \mathtt{inst}[c']$. These choices determine the refinements of the class and method vectors, as described in Section 3.

To complete the verification we need only check that the dispatch matrix derived from the **FJ** class table, and the associated class and method vectors satisfy the stated refinements, which they shall do by construction. This guarantees the well-behavior of dynamic dispatch in the sense described in Section 3, which ensures that "message not understood" cannot arise.

The main additional complication to account for self-reference is that we must, as outlined in Section 3, choose abstract refinement variables r^c and r_m that refine the abstract type t_{obj} of objects, together with a set of entailments, θ, that will be true for ρ^c_{obj} and ρ^m_{obj} when t_{obj} is instantiated to τ_{obj}. Within the dispatch matrix the code makes use of these assumptions, just as it would have made use of the refinement entailments that are true of $\texttt{inst[}c\texttt{]}$ and $\texttt{recog[}m\texttt{]}$ in the non-self-referential case. The result proceeds along similar lines to those outlined above.

4.2 Compiling FJ to L

The following presentation of **FJ** follows the CBV version by Pierce (2002). One difference is that we use $\boldsymbol{m}$ for **FJ** method names, because we need to distinguish these from the method names m in the interpretation in **L**. For classes c the two coincide.

The syntax of **FJ** is a subset of Java, aside from top-level programs, which consist of a class table and an expression to evaluate.

class declarations	CL	$::=$	$\texttt{class}\,c\,\texttt{extends}\,c\ \{\overline{c}\ \overline{f};K\ \overline{ME}\}$
constructor declarations	K	$::=$	$c(\overline{c}\ \overline{f})\ \{\texttt{super}(\overline{f});\ \textit{this}.\overline{f}=\overline{f};\}$
method declarations	ME	$::=$	$c\ \boldsymbol{m}(\overline{c}\ \overline{x})\ \{\texttt{return}\ T;\}$
terms	T	$::=$	$x \mid T.f \mid \texttt{new}\,c(\overline{T}) \mid T\cdot\boldsymbol{m}(\overline{T}) \mid (c)\ T$
values	V	$::=$	$\texttt{new}\,c(\overline{V})$
class table	CT	$::=$	$\overline{CL}$
program	P	$::=$	(CT, T)

We now instantiate our framework to show a relatively straightforward compilation of **FJ** programs to **L** types and expressions. To ease the presentation and aid comparison we adopt some similar notation to **FJ**, including having a single implicit global class table CT.

We rely on the following auxiliary definitions from the presentation of **FJ** by Pierce (2002):

$$\texttt{mtype}(\boldsymbol{m}, c)\quad \texttt{mbody}(\boldsymbol{m}, c)\quad \texttt{fields}(c)\quad c <: d$$

We also require the following definitions derived from these, which depend on some standard properties of **FJ** like unique fields. For notational convenience we treat method arguments like records with integers as the field names. Also, the Π record type below with $c\,i \in \overline{\mathsf{arg}}_{\boldsymbol{m}}$ constructs a record type with indices i, and in what follows similarly the index excludes the c type declaration.

$$\left.\begin{aligned} \overline{\mathsf{arg}}_{\boldsymbol{m}} &\triangleq (c_1\,1), \ldots, (c_n\,n) \\ \mathsf{res}_{\boldsymbol{m}} &\triangleq c \end{aligned}\right\} \begin{aligned} &\text{where } \mathtt{mtype}(\boldsymbol{m}, c') = \overline{c} \to c \text{ for some } c' \\ &\qquad\qquad\text{(this maps each } \boldsymbol{m} \text{ uniquely)} \end{aligned}$$

$$\begin{aligned} F &\triangleq \{cf \mid cf \in F^{c'} \text{ for some } c'\} \quad \text{(this maps each } f \text{ uniquely)} \\ F^{c} &\triangleq \mathtt{fields}(c) \\ \tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}) &\triangleq \prod_{c\,i \in \overline{\mathsf{arg}}_{\boldsymbol{m}}} t_{\mathsf{obj}} \end{aligned}$$

Each **FJ** method $\boldsymbol{m}$ gives rise to a method $\boldsymbol{m}$ in the interpretation. Fields and casting are implemented by adding extra methods: for each field we have a method $\mathtt{get}[f]$ and for each class we have a method $\mathtt{cast}[c]$. We write m to indicate a method in the interpretation, which may be any of the three forms $\boldsymbol{m}$, $\mathtt{get}[f]$ or $\mathtt{cast}[c]$.

In Figure 1 we instantiate the framework in **L** in the previous sections by defining the sets of classes and methods, C and M, and the associated types $\tau^{c}(t_{\mathsf{obj}})$ and $\tau_{m}(t_{\mathsf{obj}})$. This instantiates the types of the dispatch entries and the types of the method and class vectors from Section 3. Following the convention for **FJ**, these definitions implicitly depend on the **FJ** class table CT for a particular program, and much of the remainder of this section assumes similarly that there is a fixed "global" class table.

Class Types	**Method Types**
$C \triangleq \{c \mid c \text{ is declared in } CT\}$	$M \triangleq \{\boldsymbol{m} \mid \boldsymbol{m} \text{ is declared in } CT\} \cup \{\mathtt{get}[f] \mid f \in F\} \cup \{\mathtt{cast}[c] \mid c \in C\}$
$\tau^{c}(t_{\mathsf{obj}}) \triangleq \prod_{c'\,f \in F^{c}} t_{\mathsf{obj}}$	$\tau_{m}(t_{\mathsf{obj}}) \triangleq \tau'_{m}(t_{\mathsf{obj}})\ \mathtt{cmd}$ where $\tau'_{\boldsymbol{m}}(t_{\mathsf{obj}}) \triangleq \tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}) \rightharpoonup (t_{\mathsf{obj}}\ \mathtt{cmd})$; $\tau'_{\mathtt{get}[f]}(t_{\mathsf{obj}}) \triangleq t_{\mathsf{obj}}$; $\tau'_{\mathtt{cast}[c']}(t_{\mathsf{obj}}) \triangleq t_{\mathsf{obj}}$
$\tau_{\mathsf{cv}}(t_{\mathsf{obj}}) \triangleq \prod_{c \in C} (\tau^{c}(t_{\mathsf{obj}}) \rightharpoonup t_{\mathsf{obj}})$	$\tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \triangleq \prod_{m \in M} (t_{\mathsf{obj}} \rightharpoonup \tau_{m}(t_{\mathsf{obj}}))$
$\tau^{c}_{m} \triangleq \forall\, t_{\mathsf{obj}}\,.\,\tau_{\mathsf{cv}}(t_{\mathsf{obj}}) \rightharpoonup \tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \rightharpoonup \tau^{c}(t_{\mathsf{obj}}) \rightharpoonup \tau_{m}(t_{\mathsf{obj}})$	

Figure 1: Compiling **FJ** syntax to **L** types

$$
\begin{array}{rcll}
e^c_m & \triangleq & \Lambda(t_{\mathsf{obj}})\,\lambda\,(cv\!:\!\tau_{\mathsf{cv}}(t_{\mathsf{obj}}))\ \lambda\,(mv\!:\!\tau_{\mathsf{mv}}(t_{\mathsf{obj}})) & \\
 & & \qquad\qquad \lambda\,(u\!:\!\tau^c(t_{\mathsf{obj}}))\ \mathtt{cmd}\,\Psi(k^{c,\Gamma}_m) & \\
 & & & \\
\text{where } \Gamma & \triangleq & \left(\begin{array}{l} t_{\mathsf{obj}}, cv : \tau_{\mathsf{cv}}(t_{\mathsf{obj}}), mv : \tau_{\mathsf{mv}}(t_{\mathsf{obj}}), \\ u : \tau^c(t_{\mathsf{obj}}), \mathit{this} : t_{\mathsf{obj}} \end{array}\right) & \\
\Psi & \triangleq & \mathit{this} \mapsto cv \cdot c(u) & \\
 & & & \\
\text{and } k^{c,\Gamma}_{\boldsymbol{m}} & \triangleq & \mathtt{ret}\ (\lambda\,(\overline{x}\!:\!\tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}))\ \|T\|^{\Gamma,\overline{x}:t_{\mathsf{obj}}}) & \text{if } \mathtt{mbody}(\boldsymbol{m}, c) = (\overline{x}, T) \\
 & \triangleq & \mathtt{fail} & \text{if } \mathtt{mbody}(\boldsymbol{m}, c) \text{ is undefined} \\
k^{c,\Gamma}_{\mathtt{get}[f]} & \triangleq & \mathtt{ret}\ (u \cdot f) & \text{if } f \in F^c \\
 & \triangleq & \mathtt{fail} & \text{otherwise} \\
 & & & \\
k^{c,\Gamma}_{\mathtt{cast}[c']} & \triangleq & \mathtt{ret}\ \mathit{this} & \text{if } c <: c' \\
 & \triangleq & \mathtt{error} & \text{otherwise}
\end{array}
$$

Figure 2: Dispatch entries for **FJ** methods $\boldsymbol{m}$, plus $\mathtt{get}[f]$ and $\mathtt{cast}[c']$

The dispatch matrix entries e^c_m are defined in Figure 2 via an auxiliary command $k^{c,\Gamma}_m$, with the context Γ explicitly indicating which type variables and typed expressions variables are allowed to be free relative to the command, in this case t_{obj}, cv, mv, u and *this*. We adopt the convention that each **FJ** variable has a corresponding **L** variable with the same name. This is particularly convenient in the translation of terms $|T|^{\Gamma}$. Also in what follows we often use substitutions Ψ, to replace free variables with types and expressions as appropriate, such as the substitution for *this* here.

More generally, a context Γ can specify allowed free type variables t and allowed free expression variables along with their type which can depend on type variables earlier in the context. This means that t_{obj} is an abstract type in the rest of Γ. Indeed, we consider the initial part of Γ with $t_{\mathsf{obj}}, cv : \tau_{\mathsf{cv}}(t_{\mathsf{obj}}), mv : \tau_{\mathsf{mv}}(t_{\mathsf{obj}})$ to correspond to a client's view of an existential package with type τ_{dd} from Section 3, with cv and mv being the `new` and `snd` components. This is appropriate here because with self-reference the bodies of the dispatch entries are themselves clients of dynamic dispatch abstract type. Γ is augmented with $u : \tau^c(t_{\mathsf{obj}})$ so that the instance data of the object is available and $\mathit{this} : t_{\mathsf{obj}}$ to appropriately allow *this* to appear in the **FJ** method bodies.

In the definition of $k^{c,\Gamma}_m$ in Figure 2, for each class c the we interpret each of three kinds of methods using the `fail` command appropriately for undefined method bodies and undefined fields and `error` for casts to non-superclasses.

Defined method bodies are translated via an inductive translation $\|T\|^{\Gamma}$ which we will see shortly. We use a little syntactic sugar here, following **FJ**, writing $\lambda\,(\overline{x}\!:\!\tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}))\ T$ to abbreviate a function binding the variables in $\overline{x}$ to the corresponding components of the argument.

To interpret **FJ** we only need to use the instance data u in two ways:

- Each get method $\texttt{get}[f]$ for class c is interpreted as a command returning the f field of $u : \tau^c(t_{\mathsf{obj}})$. f must be present in $\tau^c(t_{\mathsf{obj}})$ if $f \in F^c$, assuming that we've correctly dispatched to $e^c_{\texttt{get}[f]}$ due to sending method $\texttt{get}[f]$ to an instance of (exactly) class c. (See the definition of e'_{mv} below.)
- The **FJ** special variable *this* is substituted by the expression $cv \cdot c(u)$ with type t_{obj} which is equivalent to the object on which the method was called, assuming that we've correctly dispatched to e^c_m due to sending method m to an instance of c. This is then available for the translation of recursive method calls in method bodies (via $\| T \|^{\Gamma, \overline{x}:t_{\mathsf{obj}}}$) and is also used for successful casts.

It's important here that the instance data for a class c is only directly accessed from the $\texttt{get}[f]$ method implementation for exactly the class c. All other uses of the instance data are via method calls to *this* which dispatch appropriately, hence no subtyping or similar constraints are ever required between the types of instance data of different classes.[4]

We compile **FJ** expressions T as corresponding **L** commands, in a relatively direct way, aside from making the propagation of errors explicit via the monadic bind. The compilation is parameterized by a context Γ that includes **FJ** expression variables in scope to corresponding **L** expression variables, including *this* which is always in scope in **FJ** method bodies. Γ also maps cv and mv to corresponding **L** expressions for the class and method vectors (renaming as necessary to avoid clashes with **FJ** variable names).

$$\| T \|^{\Gamma} \triangleq \texttt{cmd}\, | T |^{\Gamma}$$

$$\begin{array}{rcl}
|x|^{\Gamma} & \triangleq & \texttt{ret}\, x \qquad \text{if } x \text{ is in } \Gamma \text{ (including when } x \text{ is } \textit{this}\text{)} \\
|\texttt{new}\, c\texttt{(}\overline{T}\texttt{)}|^{\Gamma} & \triangleq & \texttt{bnd}\, \overline{x} \leftarrow \| \overline{T} \|^{\Gamma} ; \texttt{ret}\; \texttt{(}cv \cdot c\texttt{)}\,\texttt{(}\overline{x}\texttt{)} \\
| T \cdot m\texttt{(}\overline{T}\texttt{)} |^{\Gamma} & \triangleq & \texttt{bnd}\, x \leftarrow \| T \|^{\Gamma} ; \texttt{bnd}\, y \leftarrow \texttt{(}mv \cdot m\texttt{)}\,\texttt{(}x\texttt{)} ; \\
 & & \texttt{bnd}\, \overline{x} \leftarrow \| \overline{T} \|^{\Gamma} ; \texttt{bnd}\, w \leftarrow y\texttt{(}\overline{x}\texttt{)} ; \texttt{ret}\, w \\
| T \cdot f |^{\Gamma} & \triangleq & \texttt{bnd}\, x \leftarrow \| T \|^{\Gamma} ; \texttt{bnd}\, y \leftarrow \texttt{(}mv \cdot \texttt{get}[f]\texttt{)}\,\texttt{(}x\texttt{)} ; \texttt{ret}\, y \\
| \texttt{(}c\texttt{)}\; T |^{\Gamma} & \triangleq & \texttt{bnd}\, x \leftarrow \| T \|^{\Gamma} ; \texttt{bnd}\, y \leftarrow \texttt{(}mv \cdot \texttt{cast}[c]\texttt{)}\,\texttt{(}x\texttt{)} ; \texttt{ret}\, y
\end{array}$$

At this point we can use the expressions e^c_m to interpret the class table CT of an **FJ** program as a typed **L** expression for the dispatch matrix:

$$e_{\mathsf{dm}} = \langle \langle e^c_m \rangle_{m \in M} \rangle_{c \in C}$$

We sketch here two parts of the type correctness theorem for the compilation to **L** of the **FJ** syntax. Because every **FJ** class/type is interpreted as t_{obj}, type correctness in **L** corresponds to **FJ** syntactic correctness, including scoping of

[4]An alternative approach to *this* is to pass it as a separate argument to e^c_m, with an invariant that u and *this* must correspond. This makes little difference here when interpreting **FJ**, but appears to scale better to certain kinds of extensions like run-time inheritance.

variables and consistency of argument counts. We omit some syntactic lemmas such as that subclasses have all the fields of their superclasses with the same type. Note that the **FJ** object types of the fields are not yet relevant, and the presence of specific object types in the lemma statement is simply because **FJ** lacks a judgment witnessing syntactic correctness without also requiring specific object types.

We write $\vdash_{\textsf{FJ}} ME\,\texttt{OK}\ \texttt{in}\,c$ for the **FJ** judgment "ME is a valid method declaration for class c", which implicitly depends on the types declared in CT; see Pierce (2002).

Lemma 1.

1. *If* $\overline{x} : \overline{c} \vdash_{\textsf{FJ}} T : c$ *and* $\Gamma = t_{\textsf{obj}}, cv : \tau_{\textsf{cv}}(t_{\textsf{obj}}), mv : \tau_{\textsf{mv}}(t_{\textsf{obj}})$ *then* $t_{\textsf{obj}}, \overline{x} : t_{\textsf{obj}} \vdash |T|^{\Gamma} \stackrel{\cdot}{\leadsto} t_{\textsf{obj}}$.
2. *If* $\texttt{mtype}(\boldsymbol{m}, c) = \overline{c} \to c_0$ *and* $\texttt{mbody}(\boldsymbol{m}, c) = (\overline{x}, T_0)$ *and* $\vdash_{\textsf{FJ}} c_0\ m\texttt{(}\overline{c}\,\overline{x}\texttt{)}\{\texttt{return}\ T_0;\}\ \texttt{OK}\ \texttt{in}\,c$ *then* $\vdash e^c_m : \tau^c_m$.

A consequence of our compilation is that the structure of an **FJ** value V is observable via its translation $k = |V|^{\Gamma}$ in **L**. This is because it is possible to observe errors and non-errors for calls to $\texttt{cast}[c']$ on v for each c', from which we can determine its class c and then determine (inductively) its fields by calling $\texttt{get}[f]$. If this observability seems suspect, note that it is required by the definition of **FJ** due to the class and fields of object values being observable everywhere, including in top-level expressions. Further, the compilation can easily be modified to accommodate similar languages which allow the exact class of an object to be hidden (by omitting or restricting the cast methods) or which have private fields (by omitting the get methods and instead using direct access to the instance data).

We now characterize the translations of values via the following inductive definition. (Note that all **FJ** values have the form $\texttt{new}\,c\texttt{(}\overline{V}\texttt{)}$.)

$$|\texttt{new}\,c\texttt{(}\overline{V}\texttt{)}|^{\Gamma}_{\textsf{val}} \quad \triangleq \quad cv \cdot c(|\overline{V}|^{\Gamma}_{\textsf{val}})$$

The following lemma shows that for values this is equivalent to the previous translation $|\cdot|^{\Gamma}$, modulo some evaluation steps that reduce $\texttt{bnd}\,x{\leftarrow}\texttt{cmd}\ \texttt{(ret}\,v\texttt{)};\ k$ to $[v/x]k$. This is a standard evaluation rule for monadic commands, and easily derived from the sum interpretation of commands described in Section 2. We show some details of the proof just to give the flavor of such proofs.

Lemma 2. *(value translation)*
If $\Gamma = t_{\textsf{obj}}, cv : \tau_{\textsf{cv}}(t_{\textsf{obj}}), mv : \tau_{\textsf{mv}}(t_{\textsf{obj}}), \Gamma'$ *and* $\Psi : \Gamma$ *with both* $\Psi(cv) = e_{\textsf{cv}}$ *and* $\Psi(mv) = e_{\textsf{mv}}$ *terminating (and closed)*
then for all V *we have* $\Psi(|V|^{\Gamma}) \mapsto^* k$ *and* k *final*
iff $\Psi(|V|^{\Gamma}_{\textsf{val}}) \mapsto^* v$ *and* v *val* *and* $k = \texttt{ret}\,v$.

PROOF. (sketch) By induction on V. We have just one case.
CASE: $V = \texttt{new}\, c\texttt{(}\overline{V}\texttt{)}$. Then

$$\boxed{\begin{aligned}&\Psi(|V|^{\Gamma})\\ &= \Psi(|\texttt{new}\, c\texttt{(}\overline{V}\texttt{)}|^{\Gamma})\\ &= \texttt{bnd}\,\overline{x} \leftarrow \texttt{cmd}\,\Psi(|\overline{V}|^{\Gamma})\,;\texttt{ret}\,\texttt{(}e_{\mathsf{cv}}\cdot c\texttt{)}\,\texttt{(}\overline{x}\texttt{)}\end{aligned}} \qquad \boxed{\begin{aligned}&\Psi(|V|^{\Gamma}_{\mathsf{val}})\\ &= \Psi(|\texttt{new}\, c\texttt{(}\overline{V}\texttt{)}|^{\Gamma}_{\mathsf{val}})\\ &= \texttt{(}e_{\mathsf{cv}}\cdot c\texttt{)}\,\texttt{(}\Psi(|\overline{V}|^{\Gamma}_{\mathsf{val}})\texttt{)}\end{aligned}}$$

[Left $\Longrightarrow$ right] (the other direction is similar)

If LHS $\mapsto^* k$ and k final, then, by inversion on the evaluation, for each V_i there is k_i s.t. $\Psi(|V_i|^{\Gamma}) \mapsto^* k_i$ and k_i final.

Applying the I.H. to each V_i yields $\Psi(|V_i|^{\Gamma}_{\mathsf{val}}) \mapsto^* v_i$ and v_i val and $k_i = \texttt{ret}\, v_i$.

Then LHS $\mapsto^* \texttt{ret}\,\texttt{(}e_{\mathsf{cv}}\cdot c\texttt{)}\,\texttt{(}\overline{v}\texttt{)}$ and RHS $\mapsto^* \texttt{(}e_{\mathsf{cv}}\cdot c\texttt{)}\,\texttt{(}\overline{v}\texttt{)}$.

But then $\texttt{ret}\,\texttt{(}e_{\mathsf{cv}}\cdot c\texttt{)}\,\texttt{(}\overline{v}\texttt{)} \mapsto^* k$ (since LHS $\mapsto^* k$, and $\mapsto$ is deterministic).

Thus $k = \texttt{ret}\, v$ for some v with v val and RHS $\mapsto^* v$, as required. □

4.3 Class and method vectors (sum-based)

Now, so far nothing in our **FJ** compilation is specific to the sum-based or product-based organization. But, to have a concrete verification of a complete framework, we now consider the full implementation of self-referential class and method vectors. This subsection isn't specific to **FJ**, but applies generally to any e_{dm} with self-reference via cv and mv parameters in dispatch entries. We have delayed the full details until now so that they can be considered in a more concrete context than in Section 3.

We focus on the method-based (sum) organization because it is the road (much) less traveled, and leads to some novel views of some aspects, but everything that follows also works out dually for the class-based (product) organization.

The appropriate sum-based recursive object type $\tau^{\Sigma}_{\mathsf{obj}}$ is as in Section 3 and the corresponding self-referential class and method vectors are as follows. Following Harper (2012) **L** uses $\texttt{fold(}e\texttt{)}$ and $\texttt{unfold(}e\texttt{)}$ for recursive types, and $\texttt{self}\, x\, \texttt{is}\, e$ and $\texttt{unroll(}e\texttt{)}$ for recursive expressions.

$$\begin{aligned}
\tau^{\Sigma}_{\mathsf{obj}} &\triangleq \mu\, t_{\mathsf{obj}}.\sum_{c\in C}\tau^{c}(t_{\mathsf{obj}})\\
e^{\Sigma}_{\mathsf{cv}} &\triangleq \langle c \hookrightarrow \lambda\,\texttt{(}u\texttt{:}\tau^{c}(\tau_{\mathsf{obj}})\texttt{)}\;\texttt{fold(}c\cdot u\texttt{)}\rangle_{c\in C} : \tau_{\mathsf{cv}}(\tau_{\mathsf{obj}}).\\
e^{\Sigma}_{\mathsf{mv}} &\triangleq \texttt{unroll(}e^{\Sigma\,\prime}_{\mathsf{mv}}\texttt{)} : \tau_{\mathsf{mv}}(\tau_{\mathsf{obj}})\\
e^{\Sigma\,\prime}_{\mathsf{mv}} &\triangleq \texttt{self}\; mv\, \texttt{is}\, \langle m \hookrightarrow \lambda\,\texttt{(}this\texttt{:}\tau_{\mathsf{obj}}\texttt{)}\;\texttt{case}\,\texttt{unfold(}this\texttt{)}\;\texttt{\{}c\cdot u \Rightarrow e^{c\,\prime}_{m}\texttt{\}}_{c\in C}\rangle_{m\in M}\\
&\qquad \text{where}\quad e^{c\,\prime}_{m} \triangleq e_{\mathsf{dm}}\cdot c\cdot m\texttt{[}\tau_{\mathsf{obj}}\texttt{]}\,\texttt{(}e_{\mathsf{cv}}\texttt{)}\,\texttt{(}e_{\mathsf{mv}}\texttt{)}\,\texttt{(}u\texttt{)}
\end{aligned}$$

Lemma 3. *(Dynamic dispatch) If $v : \tau^c$ and v val then*

$$e^{\Sigma}_{mv} \cdot m\,(e^{\Sigma}_{cv} \cdot c(v)) \mapsto^* e_{dm} \cdot c \cdot m\,[\tau_{obj}]\,(e^{\Sigma}_{cv})\,(e^{\Sigma}_{mv})\,(v)$$

This lemma exactly characterizes correctness of e_{cv} and e_{mv} as an implementation of dynamic dispatch, and there is a dual proof for the product-based organization. What follows generally doesn't depend the implementation, just on this lemma, except where noted. Hence we generally omit the Σ superscripts in what follows.

4.4 Top-level and compilation correctness

We compile the top-level "external" term T in a program (CT, T) as $\Psi_{\mathsf{ex}}(|T|^{\Gamma_{\mathsf{ex}}})$, via Ψ_{ex} and Γ_{ex} (below) which appropriately omit *this*, and have both $\Psi_{\mathsf{ex}}(cv)$ and $\Psi_{\mathsf{ex}}(mv)$ closed and terminating.

$$\Gamma_{\mathsf{ex}} \triangleq t_{\mathsf{obj}}, cv : \tau_{\mathsf{cv}}(t_{\mathsf{obj}}), mv : \tau_{\mathsf{mv}}(t_{\mathsf{obj}})$$
$$\Psi_{\mathsf{ex}} \triangleq t_{\mathsf{obj}} \mapsto \tau^{\Sigma}_{\mathsf{obj}}, cv \mapsto e^{\Sigma}_{\mathsf{cv}}, mv \mapsto e^{\Sigma}_{\mathsf{mv}}$$

Then using the earlier lemmas we can show that the compilation is operationally sound. This has two parts: one for ordinary **FJ** evaluation steps and one for invalid downcasts. The theorem statement and proof involve **FJ** evaluation contexts $E\{\}$, defined as follows, as in Pierce (2002).

$$\begin{aligned} E\{\} ::= \{\}\ &|E\{\}.f \mid \mathtt{new}\, c(\overline{V}, E\{\}, \overline{T}) \\ &|E\{\} \cdot \boldsymbol{m}(\overline{T}) \mid V \cdot \boldsymbol{m}(\overline{V}, E\{\}, \overline{T}) \mid (c)\ E\{\} \end{aligned}$$

Theorem 4. *(compilation correctness) Suppose for a particular **FJ** class table CT we have $\vdash_{\mathbf{FJ}} T : c$. Then*

1. *if $T \mapsto_{\mathbf{FJ}} T'$ then $\Psi_{ex}(|T|^{\Gamma_{ex}}) \mapsto^* \Psi_{ex}(|T'|^{\Gamma_{ex}})$*
2. *if T has the form $E\{(c)\ \mathtt{new}\, c'\,(\overline{V})\}$ and not $c' <: c$ then $\Psi_{ex}(|T|^{\Gamma_{ex}}) \mapsto^* \mathtt{error}$.*

Proof. (Sketch.) By induction on (closed) T for part 1, using Lemma 3 (dynamic dispatch) to emulate **FJ** calls to $\boldsymbol{m}$ on instances of c with instance data V via $e^c_m\,[\tau_{\mathsf{obj}}]\,(e_{\mathsf{cv}})\,(e_{\mathsf{mv}})\,(v)$ where $|V|^{\Gamma}_{\mathsf{val}} \mapsto^* v$. We similarly use Lemma 2 (value translation) to produce corresponding **L** values when the **FJ** evaluation rule requires certain subterms of T to be values.

4.5 Interpreting **FJ** types as refinements

Figure 3 shows the details of our interpretation of **FJ** types, instantiating the setup in Section 3. Firstly we define M^c and dual C_m which we take as our specification of what class-method combinations are required to dispatch to valid implementations. This is in fact derived from the class table of the **FJ** program here, since **FJ** lacks a mechanism to separately specify such requirements, and we wish to provide the same guarantees as the the **FJ** type system in regards to "method not understood" failures. We still consider this specification as conceptually prior to the actual code implementing classes and methods, and in general it could be derived from a separate specification.

Unlike for types, the refinements indicated for the results of dispatch entries e^c_m for a single method m can differ between classes due to unrequired class-method combinations. So, we choose $\rho_m(\vec{r})$ as the appropriate refinement for required combinations, and then (below in Figure 3) we choose $\rho^c_m(\vec{r})$ as $\top$ (which includes all commands, hence `fail`) when the combination c, m isn't specified as required.

Next Figure 3 introduces type variables r^c and r_m which conceptually indicate "instances of c" and "recognizers of m". However, taking a behavioral view, we actually characterize r^c in terms of behavior, namely the methods that instances of r^c recognize. Thus, r^c includes all objects that recognize all methods that instances of c do.

The refinement r^m directly indicates that method m is recognized. θ_0 is a set of entailments that are safe based directly on what class-method combinations are required. However, this directness may exclude some combinations that behaviorally should be included, based on the above characterization of r^c. Thus θ is constructed so that it is a superset of the entailments in θ_0, closed with respect to the behavioral view. As we shall see, θ is sufficient to justify subsumption between class types in **FJ** (which is built into some of the **FJ** typing rules), also called subclassing, while θ_0 is not.

The last part of the union in the definition of θ represents a dual concern: that given a specified set of required class-method combinations, knowledge that a particular object recognizes a particular method may be sufficient to deduce that the object also recognizes some other methods. We call this dual concept supermethoding, and include it here to emphasize that it is the natural dual of subclassing. Further, we note that what appears to be essentially the same concept has been studied significantly in the mature field of *formal concept analysis*, for an overview see the text by Ganter et al. (1997).

We now verify that these definitions satisfy the conditions in Section 3.2

Lemma 5. *For all classes c, c', if $CT \vdash_{\mathsf{FJ}} c' <: c$ then $(r^{c'} \leq_{t_{obj}} r^c)$ is in θ.*

Proof. (sketch) Roughly by construction: in **FJ**, subclassing c to form c' leads to each method of c either being inherited or overridden in c' (with the same type), and so on transitively, hence c' has all methods that c does.

Specification of required class-method combinations

$$M^c \triangleq \{\boldsymbol{m}{\in}M \mid \mathtt{mtype}(\boldsymbol{m}, c) \text{ is defined}\} \cup \{\mathtt{get}[f] \mid f \in F^c\} \cup \{\mathtt{cast}[c] \mid c \in C\} \qquad C_m \triangleq \{c \in C \mid m \in M^c\}$$

Refinement Variables and Entailment Constraints

$$\vec{r} \triangleq \{r^c\}_{c\in C} \cup \{r_m\}_{m\in M} \quad \text{(with the } r^c \text{ and } r_m \text{ all distinct)}$$

$$\theta_0 \triangleq \{r^c \leq_{t_{\mathsf{obj}}} r_m \mid c \in C, m \in M^c\}$$

$$\begin{aligned}\theta \triangleq \theta_0 &\cup \{\, r^c \leq_{t_{\mathsf{obj}}} r^{c'} \mid \text{for all } r_m.\ r^c \leq_{t_{\mathsf{obj}}} r_m \text{ in } \theta_0 \text{ if } r^{c'} \leq_{t_{\mathsf{obj}}} r_m \text{ in } \theta_0\} \\ &\cup \{r_m \leq_{t_{\mathsf{obj}}} r_{m'} \mid \text{for all } r^c.\ r^c \leq_{t_{\mathsf{obj}}} r_{m'} \text{ in } \theta_0 \text{ if } r^c \leq_{t_{\mathsf{obj}}} r_m \text{ in } \theta_0\}\end{aligned}$$

Class Refinements	**Method Refinements**
$\rho^c(\vec{r}) \triangleq \prod_{(c'f)\in F^c} r^{c'}$	$\rho_{\boldsymbol{m}}(\vec{r}) \triangleq \mathtt{ret}\,((\prod_{c\,i\in\overline{\mathsf{arg}}_{\boldsymbol{m}}} r^c) \rightharpoonup \left(\begin{matrix}\mathtt{ret}\, r^{\mathsf{res}_{\boldsymbol{m}}} \\ \vee\, \mathtt{error}\end{matrix}\right))$ $\rho_{\mathsf{get}[f]}(\vec{r}) \triangleq \mathtt{ret}\, r^c$ for each $c\,f \in F$ $\rho_{\mathsf{cast}[c]}(\vec{r}) \triangleq \mathtt{ret}\, r^c \vee \mathtt{error}$ for each $c \in C$
$\rho_{\mathsf{cv}}(\vec{r}) \triangleq \prod_{c\in C} (\rho^c(\vec{r}) \rightharpoonup r^c)$	$\rho_{\mathsf{mv}}(\vec{r}) \triangleq \prod_{m\in M} (r_m \rightharpoonup \rho_m(\vec{r}))$

Dispatch Entry Refinements

$$\rho^c_m \triangleq \forall\,(t_{\mathsf{obj}} \sqsupseteq \vec{r} : \theta)\ .\ \rho_{\mathsf{cv}}(\vec{r}) \rightharpoonup \rho_{\mathsf{mv}}(\vec{r}) \rightharpoonup \rho^c(\vec{r}) \rightharpoonup {\rho^c_m}'(\vec{r})$$

$$\text{where } {\rho^c_m}'(\vec{r}) \triangleq \begin{cases}\rho_m(\vec{r}) & \text{if } m \in M^c \\ \top & \text{otherwise}\end{cases}$$

Refinements of $\tau^{\Sigma}_{\mathsf{obj}}$ (sum-based)

$$\boxed{\rho^c_{\mathsf{obj}}(\vec{r}) \triangleq \bigwedge_{m\in M^c} \rho^m_{\mathsf{obj}}(\vec{r})} \qquad \boxed{\rho^m_{\mathsf{obj}}(\vec{r}) \triangleq \bigvee_{c\in C_m} (c \cdot \rho^c(\vec{r}))}$$

$$\vec{\rho}_\Sigma \triangleq \mu\,\vec{r}.(\rho^c_{\mathsf{obj}}(\vec{r}))_{c\in C}, (\rho^m_{\mathsf{obj}}(\vec{r}))_{m\in M}\ \mathtt{in}\ \vec{r}$$

$$\mathtt{inst}[c] \triangleq \rho^c_{\mathsf{obj}}(\vec{\rho}_\Sigma) \qquad \mathtt{recog}[m] \triangleq \rho^m_{\mathsf{obj}}(\vec{\rho}_\Sigma)$$

Figure 3: Interpreting **FJ** types as **L** refinements

Lemma 6. *For all $c \in C$ and $m \in M$ we have $e^c_m \in_{\tau^c_m} \rho^c_m$*

Proof. (sketch) We show that for all τ_{obj}, $\vec{\rho} \sqsubseteq \tau_{\mathsf{obj}}$ with $\vec{\rho}$ *sat.* θ, and all $e_{\mathsf{cv}} \in \rho_{\mathsf{cv}}(\vec{\rho})$, $e_{\mathsf{mv}} \in \rho_{\mathsf{mv}}(\vec{\rho})$, $v_u \in \rho^c(\vec{\rho})$
that $[\tau_{\mathsf{obj}}/t_{\mathsf{obj}}][\vec{\rho}/\vec{r}][e_{\mathsf{cv}}/cv][e_{\mathsf{mv}}/mv][v_u/u]k^{c,\Gamma}_m \in {\rho^c_m}'(\vec{r})$ in each case.

The case for a defined $\boldsymbol{m}$ involves the translation $|T|^\Gamma$ of the method body $\vdash_{\mathsf{FJ}} T : \mathsf{res}_{\boldsymbol{m}}$ which we treat by induction on the **FJ** typing derivation, generalizing appropriately.

Lemma 7. *$\vec{\rho}_\Sigma$ satisfies θ.*

Proof. From the definitions of $\vec{\rho}$, $\rho^c_{\mathsf{obj}}(\vec{r})$ and $\rho^m_{\mathsf{obj}}(\vec{r})$ we have each required $\rho^c \leq \rho_m$ and $\rho^c \leq \rho^{c'}$ simply by inclusions between the sets M^c and C_m. (No entailments for $\rho^c(\vec{\rho})$ are involved.)

Lemma 8. *$e^\Sigma_{\mathit{cv}} \in \rho_{\mathit{cv}}(\vec{\rho}_\Sigma)$ and $e^\Sigma_{\mathit{mv}} \in \rho_{\mathit{mv}}(\vec{\rho}_\Sigma)$.*

Proof. (sketch) By the properties of refinements in Section 2.

Lemma 9.
If ($m = \boldsymbol{m}$ and $\mathtt{mbody}(c, m)$ defined), or ($m = \mathtt{get}[f]$ and $f \in \mathtt{fields}(c)$), or $m = \mathtt{cast}[c]$ then $\quad \rho^c_m \leq \forall\,(t_{\mathit{obj}} \sqsupseteq \vec{r} : \theta)\;.\;\rho_{\mathit{cv}}(\vec{r}) \rightharpoonup \rho_{\mathit{mv}}(\vec{r}) \rightharpoonup \rho^c(\vec{r}) \rightharpoonup \rho_m(\vec{r})$

Proof. For these cases the definitions of the two refinements coincide.

Theorem 10. *If $\vdash_{\mathit{FJ}} T{:}c$ then $\Psi_{\mathit{ex}}(|T|^{\Gamma_{\mathsf{ex}}}) \in (\mathtt{ret\,inst}\,[c]) \vee \mathtt{error}$.*

Proof. (sketch) By induction on the typing derivation for T, and using lemma 6 with the subsequent lemmas discharging the assumptions of that lemma.

(Alternatively, the result follows from the type preservation and progress theorems of **FJ** sketched by Pierce (2002) and our earlier lemma that **FJ** reduction can be simulated via the interpretation in **L**, but this is perhaps less convincing as a demonstration of reasoning using semantic refinements.)

Corollary 11. *If $\vdash_{\mathit{FJ}} T{:}c$ then $\Psi_{\mathit{ex}}(|T|^{\Gamma_{\mathsf{ex}}})$ will not evaluate to `fail` indicating "message not understood".*

Thus, our interpretation of the type system of **FJ** as semantic refinements will correctly accept the translations of all well-typed **FJ** programs. As well it will of course accept any program that doesn't result in a "message not understood" or "field not understood" error even if the **FJ** type system rejects it. It can also rule out downcast failures in essentially the same way, or better, characterize exactly what conditions will lead to downcast failures.

Of course, a disadvantage is that the semantic approach generally does not as directly lead to practical tools such as refinement checkers. But, a refinement checker like that studied (and built) by Davies (2005) can be considered a proof checker for certain quite restricted language of proofs of semantic properties that can be conveniently expressed via a few annotations within or alongside a program. Making semantic refinements the primary notion not only leads to some technical simplifications, it clarifies the nature of syntactic refinements and the exact limitations that should be expected when using a refinement checker.

5 Conclusion

By separating structural from behavioral considerations we have repositioned the problem of typing for object-oriented programming from one of designing languages (structural type theories) to one of designing specifications (behavioral type theories). Rather than privileging the "message not understood" error, we instead treat it on a par with other conditions, such as "down-cast errors", that naturally arise when using dynamic dispatch, and which are much more difficult to account for in a purely structural framework. More broadly, avoiding the characteristic errors associated with dynamic dispatch becomes a particular instance of avoiding a broader class of errors, such as array bounds check errors. The emphasis on the semantic interpretation of behavioral typing may be further generalized to account for richer properties, such as the equational properties inherent in the Liskov-Wing behavioral notion of behavioral subtyping (Liskov and Wing 1994), by passing from predicates to binary relations defined over a structural type system.

In our main example we have derived the key safety property provided by the **FJ** type system through a combination of structural and behavioral typing. Being semantically defined, behavioral typing is, in general, not mechanically checkable; whether a program exhibits (or fails to exhibit) a particular behavior is a matter of proof. In this respect our formulation is coherent with the general trend toward the integration of program verification as part of standard software development practices. For this to be practical, it is necessary to develop tools that can, in common cases, perform automatic verification, or semi-automatic verification via modest "proof hints" such as annotations specifying expected invariants. For example, it appears that the existing tool SML CIDRE developed by Davies (2005) is sufficiently expressive and efficient to handle SML code corresponding to our main example, including refinements of abstract types via refinements in SML modules. More broadly, it would be interesting to integrate structural and behavioral typing in a single dependent type theory in which one may regard type refinements as propositional functions, and then apply automated reasoning systems, such as Coq (Bertot and Castéran 2004), to perform the verification. It would appear that in such a framework the **FJ** type checker would emerge as a tactic that handles the verification of the absence of "not understood" errors. This should naturally extend to full Java type checking, and other languages involving dispatch, including more involved aspects such as the variance of generics which we expect to fit well with behavioral refinements. Further, we expect this to suggest some natural extensions, for example enriching subtyping of generics with *strictness* of type parameters, or the more general *constrained inclusions* considered by Davies (2005), with the formulation of even more precise tactics and refinement checking tools being naturally open-ended.

The semantic foundations for behavioral typing suggest other interesting directions for research. As mentioned earlier, by passing to a relational interpretation of refinements we may express properties, such as parametricity properties, that hold of a particular language, or to verify properties such as the behavioral subtyping condition mentioned earlier, that hold of particular programs. Another direction is to observe that the structural treatment of dynamic dispatch naturally gives rise to a semantic account of object-oriented concepts such as subclassing. Briefly, rather than consider subclass relationships to be a matter of declaration or construction, as they are in **FJ**, we may instead define such relationships behaviorially in terms of the dispatch matrix. For example, one may consider c to be a subclass of c' whenever every method that is well-defined on instances of c' is also well-defined on instances of c, a semantic formulation of what is stated by declaration in **FJ**. It would also be interesting to extend our methods to concepts such as multiple dispatch (pattern matching on tuples of objects), or more exotic programming concepts such as predicate dispatch. These seem ripe for consideration from a behavioral/verification viewpoint, without requiring substantial changes to the underlying structural type theory.

Acknowledgement The authors are grateful to Martìn Abadi, Stephanie Balzer and Matthias Felleisen for their very helpful comments on this work.

References

M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

J. Aldrich. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 101–116, 2013.

N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 15–26. ACM, 2007.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, Nov. 1999.

L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Summary in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

L. Cardelli. Bad engineering properties of object-oriented languages. *ACM Computing Surveys (CSUR)*, 28(4es):150, 1996.

L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, Dec. 1985.

L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, Jan. 1989.

W. R. Cook. A proposal for making eiffel type-safe. *The Computer Journal*, 32(4):305–311, 1989.

W. R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, 2009.

K. Crary and R. Harper. Syntactic logical relations for polymorphic and recursive types. *Electronic Notes in Theoretical Computer Science*, 172:259–299, 2007.

R. Davies. *Practical Refinement-Type Checking.* PhD thesis, Carnegie Mellon University School of Computer Science, May 2005. Available as Technical Report CMU–CS–05–110.

R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 198–208, 2000.

E. Denney. Refinement types for specification. In *Programming Concepts and Methods PROCOMET'98*, pages 148–166. Springer, 1998.

J. Dunfield. *A Unified System of Type Refinements.* PhD thesis, Carnegie Mellon University, August 2007.

J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computation Structures*, FOSSACS'03, pages 250–266, 2003.

K. Fisher and J. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

T. Freeman and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario*, June 1991.

B. Ganter, R. Wille, and C. Franzke. *Formal Concept Analysis: Mathematical Foundations.* Springer-Verlag New York, Inc., 1997.

R. Harper. *Practical foundations for programming languages.* Cambridge University Press, 2012.

R. Harper. Practical foundations for programming languages (second edition). Available at `http://www.cs.cmu.edu/~rwh/plbook/2nded.`, 2014.

A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 1999.

S. L. P. Jones. Haskell 98: Introduction. *J. Funct. Program.*, 13(1):0–6, 2003.

B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

U. Norell. Dependently typed programming in Agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

M. Odersky and T. Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014.

F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

A. M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, 1996.

J. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development.* Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.

H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257, 1998.