Reflective Metaprogramming

by

Weiyu Miao

BS, East China Univ. of Sci. and Tech., Shanghai, China, 2005

MS, East China Univ. of Sci. and Tech., Shanghai, China, 2008

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer, and Energy Engineering

2013

This thesis entitled:
Reflective Metaprogramming
written by Weiyu Miao
has been approved for the Department of Electrical, Computer, and Energy Engineering

Prof. Jeremy Siek

Prof. Bor-Yuh Evan Chang

Prof. Jaakko Järvi

Prof. Sriram Sankaranarayanan

Prof. Fabio Somenzi

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Miao, Weiyu (Ph.D., ECEE)

Reflective Metaprogramming

Thesis directed by Prof. Jeremy Siek

Reflective Metaprogramming is capable of inspecting code, enables us to write templates, and can automatically instantiate templates according to the meta information obtained by reflection. Reflective metaprogramming supports the writing of generic and reusable software components. However, there are the challenges of designing a reflective metaprogramming language. One challenge is the design of the type system: in a language with type-reflective metaprogramming, type expressions in the residual program may be the result of meta computation, making the type system difficult to guarantee that the residual code in a metaprogram is statically type safe without sacrificing language expressiveness. Another challenge is the deign of reflective metaprogramming language. Some previous work proposed pattern-based reflection for inspecting and generating class members, but with limited ways that generated declarations can be composed.

In this thesis, we present incremental type-checking for reflective metaprograms. It detects errors earlier than C++ templates without sacrificing flexibility: we incrementally type check code fragments as they are created and spliced together during meta computation. We use type variables to represent type expressions that are not yet normalized and a new dynamic variation on existential types to represent residual code fragments. A type error in a code fragment is treated as a run-time error of the meta computation.

In this thesis, we present pattern-based traits, a language design, embodied in PTFJ (Pattern-based Traits for Featherweight Java), that combines pattern-based reflection with the expressiveness of trait composition. Pattern-based traits can group, name, and manipulate sets of member declarations. Parameterizing traits over such sets can further increase the expressiveness of pattern-based reflection.

Dedication

To my parents and my wife, for all of the support you provided. I could not have done it without you.

Acknowledgements

First, I would like to thank my parents, who give me full support to study for my PhD program in the United States. My parents always care about my study and my research. They encouraged me a lot when I was in frustration. Thank my wife, Wenwen Xiang for her support. I studied at University of Colorado at Boulder and she lives at Davis, California. I usually went back to California once every month. I understand that it is difficult for her to take care of our daughter alone, but she is very strong and never complains. Thank my daughter, Cady Miao, who gives more happiness to our family.

I am grateful to my advisor, Professor Jeremy Siek, for his support. Professor Siek gave me a lot of freedom in doing research. I used to have some frustration in my research work, but Professor Siek always supported me and helped me overcome the difficulties. Professor Siek gave me many suggestions and advice on how to write papers and do presentations.

Thank all of my PhD thesis committee members: Professor Siek, Professor Chang, Professor Sankaranarayanan, Professor Somenzi, and Professor Järvi. They were busy working on their academic papers and doing research, but they were able to sequence time reading my thesis and attending my thesis defense.

Last but not least, thank Thomas Nelson and Lou Ordorica for editing my thesis. I am not a native English speaker and inevitably, there was many English grammar mistakes in this thesis. Thomas and Lou helped me correct every English grammar mistakes, pointed out many misuse of words, and adjusted some sentence structures in this thesis.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

In 1968, Friedrich L. Bauer invented the term "Software Crisis" at the first NATO Software Engineering Conference. The crisis refers to widespread difficulties in the construction and maintenance of large and complex software systems. Software products are increasing in complexity, thus the costs to develop and maintain such software products are also increasing. This increase in complexity also reduces software understandability and quality. At the same NATO conference, McIlroy [37] proposed the systemic creation of reusable software components as a solution to the software crisis. A line of software products share similar constructive and reusable software building blocks and we can assemble these reusable blocks into software products efficiently

In the area of software engineering, researchers have proposed many programming guides and methods on how to design and compose reusable software components, such as software product lines [1] (i.e. a set of shared software components are intentionally designed and can be assembled into a line of software products), and component-based software engineering [2] (i.e. a general methodology to support the separation of concerns, by designing software components that encapsulate different functionalities).

To achieve the above software engineering methodology, we need programming language support. We need programming language features and programming paradigms that enable us to write reusable and general software components; and to compose/synthesize those components into a large project flexibly.

[1] http://www.sei.cmu.edu/productlines/
[2] http://en.wikipedia.org/wiki/Component-based_software_engineering

1.1 Reflective Metaprogramming

Reflective metaprogramming is a programming paradigm for writing abstract and general software components. A programming language supports reflective metaprogramming if the programming language supports both metaprogramming and reflection.

Metaprogramming is a programming technique that enables users to write programs that can generate other programs. MetaML [58, 59], MetaOCaml [3] , and Template Haskell [48] are functional programming languages that support metaprogramming. Because those languages enable programmers to write multi-staged programs, and the code at a higher stage can manipulate the code at a lower stage, such metaprogramming is also called multi-staged programming.

1.1.1 C++ Templates

A well-known implementation of metaprogramming is C++ templates [2]. One practical use of C++ templates is the development of reusable libraries, such as the C++ Standard Template Library [31, 38], the C++ Boost Library [1], and the Matrix Template Library [51]. With C++ templates, programmers can write functions and data structures that are generic on types, and develop software components based on concepts (i.e. concept-based programming). C++ templates can be parameterized over types as well as terms and they are instantiated at compile time, which is capable of performing partial evaluation at compile time [2], thus reducing the cost of runtime performance. Moreover, templates can be encoded and used as domain-specific types for domain-specific type checking [2].

Despite the expressiveness of C++ templates, they are still not ideal for writing reusable software components. First, current C++ compilers do not modularly type-check templates before template instantiation [40]. The compilers may generate type errors during instantiation and the error messages point into the generated code instead of the source of the error in a metaprogram. Second, C++ templates have limited compile-time reflection over classes and class members, which

[3] http://www.cs.rice.edu/~taha/MetaOCaml/

makes it difficult for programmers to write templates that can extend or generate classes as well as templates that can increment the functionality of a class member. Third, C++ templates do not have a flexible way of combining generated code. Specifically, C++ templates do not have a flexible way to inject member instantiations into a class nor compose classes to generate a new one.

1.1.2 Language Features for Code Reuse

To overcome the drawbacks of C++ templates mentioned in the previous section, we think that a reflective metaprogramming language should have the following capabilities.

(1) The language should support the writing of modular software components. Modularity means a separation of concerns. A modular software component capsulates a unit of functionality/feature, which is independent of other components and its use environment. Modularity also refers to the modular type-checking of components. Components are intended to be widely used. Ideally, they should be fully type-checked at the definition time, thus type errors generated inside the definition of a component should not be exposed to component users. As we have mentioned, C++ templates enable programmers to write metaprograms that are abstract over types, and even expressions. Templates are instantiated at compile time, but they are not fully type-checked before instantiation. So, when a metaprogram is applied, an error message generated from the metaprogram may point to a place deep inside the metaprogram [40, 52].

(2) The language should support a flexible composition of software components. When a collection of software components is composed into a large program, the restriction of composition may limit the programs that can be produced. For instance, with single inheritance in object-oriented programming, a feature in a subclass can only be used to extend a specific base class. It is difficult to factor out a feature in the subclass and apply it to some other base class. To facilitate code reuse, the language needs to provide the mechanism to incorporate and extend features for a set of software components.

(3) The language should support abstraction and specialization over code. Abstraction over code means to hide away some implementation details of the code, thus making the code more gen-

eral and applicable to different running environments. Specialization means the ability to instantiate the code, allowing the computation to be customized for specific environment. Fundamentally, a function is the abstraction of functionality implemented by a (parameterized) program procedure. When other parts of the program invoke the function, the program procedure is specialized by different arguments. For another example, using C++ templates or Java generics, we can define a generic list that is abstract over the type of its elements. When using the generic list to hold the elements of some specific type, we can first instantiate the list with that type.

(4) The language should support reflection. Reflection means the ability to inspect and analyze code. Reflection is useful or even essential for code generation. Manually instantiating a piece of abstract code can be tedious and error prone. Reflection offers the ability to inquire about the environment where a component is applied and automatically retrieve the variable instances to instantiate the component. For example, if we write a metaprogram that can automatically generate the getters and the setters for the fields of any class, we need the programming language to have the ability to iterate over a class's members and inspect their names and types, which is the ability offered by reflection.

1.2 Reflective Metaprogramming in OO Programming Languages

Reflective metaprogramming is not a replacement for object-oriented programming. Instead, it is a reinforcement that makes object-oriented programming languages more suitable for writing reusable software components.

For an object-oriented programming language, single inheritance [13] is the key mechanism for achieving class reuse, but researchers have pointed out its weakness. Single inheritance is not ideal for factoring out common features shared by a set of classes [19, 47]. Moreover, it cannot reason about or capture the member-level structure of a class [45, 17, 25, 27].

For the support of writing reusable code, researchers have introduced many language features into mainstream object-oriented programming. Many of those features refer to the application of reflective metaprogramming. In general, those language features can be divided into two categories:

class composition and class reflection.

1.2.1 Class Composition

For class composition, multiple inheritance [8, 57, 10] enables a subclass to inherit from more than one superclasses, but it suffers from the diamond problem [56]: a class uses a member from one of its base classes but the member is ambiguous because it can be inherited via multiple paths.

In 1986, Moon proposed Mixin inheritance [41, 9]. Mixin inheritance generalizes single inheritance by allowing a class to inherit from a set of superclasses that have the same set of features. In 1998, Flatt [19] proposed a calculus for mixins: a mixin is a class parameterized over superclass. It is a means of incrementing the functionality for a set of base classes.

Inheritances provide linear composition of classes. In contrast, traits [47] provide a more flexible way of composing classes: symmetric compostion (also called symmetric sum). A trait provides a collection of method implementations, but it may require a set of methods or fields that need to be imported from other traits or classes. A symmetric composition of two traits merges the methods of the traits into a new trait. If there is a name conflict during the composition, then we are provided the primitive method manipulations (such as method renaming, aliasing, exclusion, etc) to solve the name conflict.

1.2.2 Class Reflection

Language features in the class reflection category provide mechanisms for inspecting old classes and generating new ones. Runtime reflection has long been supported with techniques such as the meta-object protocol [34] and Java reflection [4] ; but interest has grown in compile-time reflection because of its lower run-time cost and improved safety. Draheim et al. [15] developed support for reflective program generators with compile-time loops that can iterate over the members of one class to create the members of another class.

Recently, Huang and Smaragdakis introduced a safe language, MorphJ [25, 26, 27]. It extends

[4] http://docs.oracle.com/javase/tutorial/reflect/

Java with the following two programming language features: (1) morph classes, mixin-like structures that can be parameterized over superclasses; and (2) pattern-based reflective declarations, class-nested declarations that contain member patterns so that they can iterate and pattern-match over class members at compile time.

To our knowledge, MorphJ provides both expressiveness in reflection and flexibility in composition. In our paper [39], we proposed pattern-based traits based on MorphJ. Pattern-based traits combine pattern-based reflection with traits, which offer more expressive forms of composition than mixin inheritance.

1.3 Challenges of Implementing Reflective Metaprogramming

To support writing reusable software components, it is desirable to incorporate reflective metaprogramming into a programming language. However, there are the challenges of designing a programming language that supports full-fledged reflective metaprogramming.

One challenge is the design of the type system. For a general-purpose reflective metaprogramming language, such as the language presented by Garcia [23], we need to weigh between expressiveness and static type safety. In a metaprogram, the type of a piece of object code can be result of meta-evaluation, therefore it is difficult to fully type-check the metaprogram before meta-evaluation and guarantee it always generates well-typed code. If you invent a type system that gives such static type safety, then the type system would restrict the set of programs that we can write.

Another challenge is how to handle the conflict between type abstraction and type reflection. Type generics and abstract datatypes intend to make type abstract and invisible to users, but type reflection intends to retrieve and inspect type for users, and makes type concrete. Class encapsulation and access control restrict the use of class members by other classes, but reflection intends to expose the members of class to other classes. So, the challenge is how to reuse and extend the members of a class while still respecting class encapsulation.

1.4 Outline of Contributions

Our research work on reflective metaprogramming can be divided into two aspects. The first aspect is the design of a type system for reflective metaprogramming languages. The type system can check the object language and guarantee static type safety for code fragments whose types are independent of meta-evaluation. Meanwhile, the type system should not be too restricted. It should not prohibit the expressiveness of metaprogramming. The second aspect is the design of programming language features for reflective metaprogramming. The language features should support writing generic, reusable, and adaptable metaprograms. Meanwhile, they should also support inspecting and analyzing types and class members; and they should extract meta information for instantiating and generating code automatically.

For the first aspect, we present incremental type-checking for reflective metaprograms. It type-checks both the meta and object fragments before meta-evaluation. The type system is a variant of the gradual type system of Siek and Taha [50], in which we use type variables to represent type expressions that are not yet normalized and a new dynamic variation on existential types to represent residual code fragments. Incremental type-checking can detect type errors in a type-reflective metaprogram as early as possible without sacrificing flexibility. Our approach is more efficient than other approaches because it avoids unnecessary meta-evaluation overhead.

For the second aspect, we present pattern-based traits, a language design, embodied in PTFJ (Pattern-based Traits for Featherweight Java), that combines pattern-based reflection with the expressiveness of trait composition. Pattern-based traits can group, name, and manipulate sets of member declarations. Parameterizing traits over such sets can further increase the expressiveness of pattern-based reflection. Moreover, we introduce pattern-based reflection at the statement level, which supports statement-level code generation and reuse. We not only extend Java with pattern-based traits, but also study their usage and performance. We implement PtjORM, an ORM tool written in pattern-based traits. Our benchmark tests show that compared to some of the mainstream Java ORM tools, PtjORM improves runtime performance by using compile-time

reflection.

1.5 Dissertation Road Map

This thesis begins with an overview of reflective metaprogramming and some of the language implementations that support reflective metaprogramming (Chapter 2). In Chapter 3, we introduce incremental type-checking for reflective metaprogramming, including the prototype and its constraint and unification based implementation. Chapter 4 gives a brief overview of traits and a line of work related to traits. In Chapter 5, we introduce pattern-based traits and the calculus PTFJ. In Chapter 6, we discuss the extension of PTFJ with statement-level reflection and code generation. Chapter 7 introduces our ORM tool, PtjORM and our benchmark tests. In Chapter 8, we discuss and compare our related work. Chapter 9 is the conclusion of this thesis.

Chapter 2

Type Reflective Metaprogramming

Metaprogramming is the writing of computer programs that generate or manipulate programs. Reflection, in the context of metaprogramming, is the ability to inspect and/or manipulate a program's metadata (such as types). We say that a language supports reflective metaprogramming if it supports both staged computation and reflection. This combination of features enables software developers to build libraries that are versatile and easy to use because the libraries can, during compilation, adapt to the contexts in which they are used.

2.1 Multi-Staged Programming

Programming languages such as C++ templates [2, 3, 12] and MetaML [58, 59] enable metaprogramming by providing multiple stages of computation, where earlier stages can manipulate code for later stages.

C++ templates are a two-staged metaprogramming language, where the language for writing code generators is called meta language (a program that generates code is called metaprogram) and the language for generated code is called object language. C++ templates support partial evaluation. They can generate code specialized for what data is available at compile time. The following C++ program implements the partial evaluation of the power function using function templates.

```
1  template <int N>
2  int powN(int M) { return M * powN<N-1>(M); };
3
4  template<>
5  int powN<0>(int M) { return 1; };
6
7  template int powN<3>(int);
```

When only the value of exponent is given at compile-time, the function template powN generates the power function specialized for that value. When instantiated with value 3, the function template generates a set of instantiations, like the following:

```
template <> struct powN<3>(int M) { return M * powN<2>(M); }
template <> struct powN<2>(int M) { return M * powN<1>(M); }
template <> struct powN<1>(int M) { return M * powN<0>(M); }
template <> struct powN<0>(int M) { return 1; }
```

The final process of combining code fragments and generating the stand-alone expression is actually performed at runtime.

MetaML supports multi-staged (arbitrary numbers of stages) computation with explicit stage annotations. The following implements the partial evaluation of the power function in MetaML.

```
1  (* val powN_core = fn : <int> → int → <int> *)
2  fun powN_core M N = if N = 0 then <1> else <~ M * ~ (powN_core M N-1)>;
3
4  (* val powN = fn : int → <int → int> *)
5  fun powN N = <fun M ⇒ ~ (powN_core <M> N)>;
6
7  fun powN_3 = ~ (powN 3);
```

In the above program, an expression inside a pair of brackets < > represents a piece of code of delayed computation. The escape operator ~ splices a piece of code into another. After meta-evaluation, powN_3 is evaluated into **fun** M ⇒ M * M * M * 1.

2.2 Garcia's Work on Reflective Metaprogramming

The template feature of C++ enables the manipulation of types-as-data and provides a way to obtain the type of an expression. Unlike C++, MetaML does not support type reflection. In 2009, Garcia published a paper that proposes a reflective metaprogramming calculus for MetaML-like languages. He captured the fundamental capabilities of C++ in his calculus [23, 22].

2.2.1 Garcia's Calculus

Figure 2.1: Garcia's reflective metaprogramming calculus.

γ type constants (e.g. **int**, **bool**)
x variables
c value constants, including booleans, integers, etc

code language

$$e ::= x \mid c \mid \lambda x : e^m.e \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \sim e^m \mid \textbf{let meta}\ x = e^m\ \textbf{in}\ e \mid e^m\langle e^m\rangle \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$$

meta language

$$e^m ::= x \mid c \mid \mathcal{T} \mid \gamma \mid e^m \to e^m \mid \lambda x : \tau^m.e^m \mid e^m\ e^m \mid \%e^m \mid \textbf{let}\ x = e^m\ \textbf{in}\ e^m \mid \textbf{if}\ e^m\ \textbf{then}\ e^m\ \textbf{else}\ e^m \mid \prec e \succ$$

$$\mathcal{T} ::= \gamma^? \mid \to^? \mid =_\tau \mid \textbf{dom} \mid \textbf{cod} \mid \textbf{typeof}$$

meta types

$$\tau^m ::= \gamma \mid \textbf{code} \mid \textbf{type} \mid \tau^m \to \tau^m$$

The syntax of Garcia's calculus is shown in Figure 2.1. The calculus presented here differs in several minor ways from Garcia's surface language. We omit functional generators, which are a convenience feature. We include the splice and lift features from Garcia's kernel language. The calculus models a two-staged metaprogramming language. The syntax includes the bracket $\prec e \succ$, escape $\sim e^m$, and lift $\%e$ (i.e. lift an object value into the meta level) constructs from MetaML.

In Garcia's calculus, types are first-class objects in the meta language and they can be used as terms. We can write meta-level expressions that generate types and such expressions are called type expressions: for instance, type expression λx : **bool**. ((**if** x **then int else bool**) $\to$ **bool**) is a meta function that when applied to a boolean, it will be evaluated into either **int** $\to$ **bool** or **bool** $\to$ **bool**.

To support type reflection, the meta language contains type primitives (i.e. type-level operations) to analyze and inquire about types. Expression $\gamma^?(e^m)$ checks if a type is a ground type; $\rightarrow^?(e^m)$ checks if a type is a function type; $e^m =_\tau e^m$ checks type equality; **dom** e^m obtains the domain of a function type; **cod** e^m obtains the codomain of a function type; and **typeof** e^m obtains the type of a piece of code generated by e^m. In the next section, we will give more illustration about the reflection power of Garcia's calculus.

2.2.2 Type Manipulation Example

Garcia's design preserves much of the power and flexibility of C++ templates while providing more support for reflection on meta data such as types. Consider the example of serializing arguments for a remote procedure call: we need to first serialize parameters into bitstrings. The following shows a simple serialization function:

```
1  /* serialize : (type list) → code → code */
2  let meta serialize = fix λrecur : (type list) → code → code.
3      λtys : (type list). λinit : code.
4          case tys of [ ] ⇒ init
5              | ty :: res ⇒
6                  if ty =τ int then
7                      ≺ λi : int. ∼ (recur res ≺ (int2str i) + "," + (∼ init) ≻) ≻
8                  else
9                      if ty =τ bool then
10                         ≺ λb : bool. ∼ (recur res ≺ (bool2str b) + "," + (∼ init) ≻) ≻
11                     else ≺ "Parameter Type Error" ≻;;
```

We assume that the following primitive functions are available.

```
int2str : int → bitstring
bool2str : bool → bitstring
```

We assume that the meta level part of Garcia's surface language is simply extended with recursive functions (reserved word **fix** is the recursive function constructor), and also extended with string, list,

pattern matching over list, and list type constructor, which are borrowed from Standard ML. The serialize function, at compile time, iterates over the types of the arguments of the function, dispatches to a type-specific print function corresponding to the type of each argument, and generates a C-like printf function that converts the parameter into a string and then concatenates the resulting strings. In the program, we utilize the type reflection capability of Garcia's calculus: for instance, expression ty $=_\tau$ **int** is used to inspect whether ty is equal to type **int**. The following shows an example of using the serialize function and the resulting code generated after meta-evaluation:

```
1  ∼ (serialize [bool, int] ≺"" ≻);;
2  ⟹
3  λb : bool.
4      λi : int.
5          ((bool2str b) + "," + (int2str i) + "," + "")
```

2.3 Type-Checking Challenges

A reflective metaprogramming type system can be expressive, resulting in types that are dependent on meta expressions, and the types of code fragments may be the result of meta-evaluation. Despite complexity and expressiveness, we would like such a type system to still maintain high efficiency and modularity. Unfortunately, current metaprogramming type systems do not fit into that standard because they suffer from the following drawbacks. One drawback is lack of modularity, which causes well-typed metaprograms to generate ill-typed object code and expose type errors that reside deep inside a metaprogram. The other drawback is lack of expressiveness, which refers to the situation when a type system is overly modular so that it compromises the expressiveness of a metaprogramming language. In other words, an overly modular type system may restrict the set of metaprograms that we can write.

In the design of type systems for metaprogramming languages, there is a tension between static type safety, which guarantees that well-typed metaprograms always generate well-typed code, and expressiveness, which enables programmers to express their intent in a straightforward manner.

2.3.1 Modular Type-Checking

In the following, we cite the definition of modular type-checking from Siek's journal article [49].

> "A language provides modular type checking when 1) a call to a function, or similarly, an instantiation of a generic, can be type checked using only its type and not its implementation and 2) the definition of function or generic can be type checked in isolation, without using any information about call sites or points of instantiation."

Metaprogramming languages with a modular type system have the following two properties: i. well-typed metaprograms always generate well-typed object code, and ii. type errors residing in the implementation of metaprograms are never exposed to users.

A modular type system can catch type errors at their place of origin, but it restricts the set of metaprograms one can write. On the other hand, non-modular type systems (such as that of C++) accept a larger set of metaprograms, but some of those metaprograms may generate ill-typed code which is difficult to debug because the error messages point into the generated code instead of the source of the error in the metaprogram.

MetaML has an ideal modular type system. Before meta-evaluation, a MetaML program is fully type-checked for both meta and object fragments so that they preserve well-typedness during meta-evaluation.

2.3.2 Two-Staged Type-Checking

Ideally, one would like the type system of a metaprogramming language to guarantee that the residual program (object program) will type check, as is the case in MetaML. However, adding reflective capabilities to the language, like Garcia's calculus, makes the MetaML guarantee next to impossible to achieve.

In Garcia's calculus (see Figure 2.1), types used to describe object expressions are part of the

meta language. So the actual type of an object expression might be the result of meta-evaluation. Consider the following program.

```
1  let meta id =
2      λx : bool. ≺ λy : if x then int else string. y ≻
3  in
4      let id_int = ∼ (id true) in
5          let id_string = ∼ (id false) in
6              (id_int 0, id_string "zero")
```

The body of meta function id contains a piece of residual code where the type of parameter y is a type expression. If the type expression is evaluated into **int**, then the residual code has type **int**→ **int**. If it is evaluated to **string**, then the residual has type **string**→ **string**. The following is another example that shows the type of an expression depends on meta-evaluation. Consider the following example that generates addition functions for different types.

```
1  /* generic_add : type → code */
2  let meta generic_add =
3      λt : type.
4          if (t =τ int) then ≺ +int ≻
5          else if (t =τ float) then ≺ +float ≻
6              else if (t =τ string) then ≺ +string ≻
7                  else if (t =τ list) then ≺ append_list ≻
8                      else %(print_error "addition not supported")
9  in
10     (∼ (generic_add int) 2 3)
```

If we type-check the program with a MetaML-like type system, we cannot find an actual return type for generic_add because its return type depends on function applications. In Garcia's calculus, the function can be typed because all code fragments are given the same type, **code**. Type **code** hides the actual type of a piece of code, making the code un-type-checked when spliced into other code.

So, for a language with reflective metaprogramming, a MetaML-like type system is overly modular so that it compromises the expressiveness of the language. In other words, an overly modular type system may restrict the set of metaprograms that we can write.

Garcia proposed a type system that uses two-staged type-checking. The type system checks the meta language before meta-evaluation and checks the object language after meta-evaluation. Figure 2.2 shows the difference in type-checking between MetaML and Carcia's calculus.

Before meta-evaluation, the type system guarantees that the residual programs inside a metaprogram are well-formed, but not necessarily well-typed. After meta-evaluation, it type-checks the code generated by the metaprogram. For example, the following metaprogram is accepted by Garcia's type system even though it generates the ill-typed code: $(2 * \textbf{true})$.

$\sim ((\lambda\, \mathsf{x} : \textbf{code}.\ \prec 2 * (\sim\ \mathsf{x}) \succ) \prec \textbf{true} \succ)$

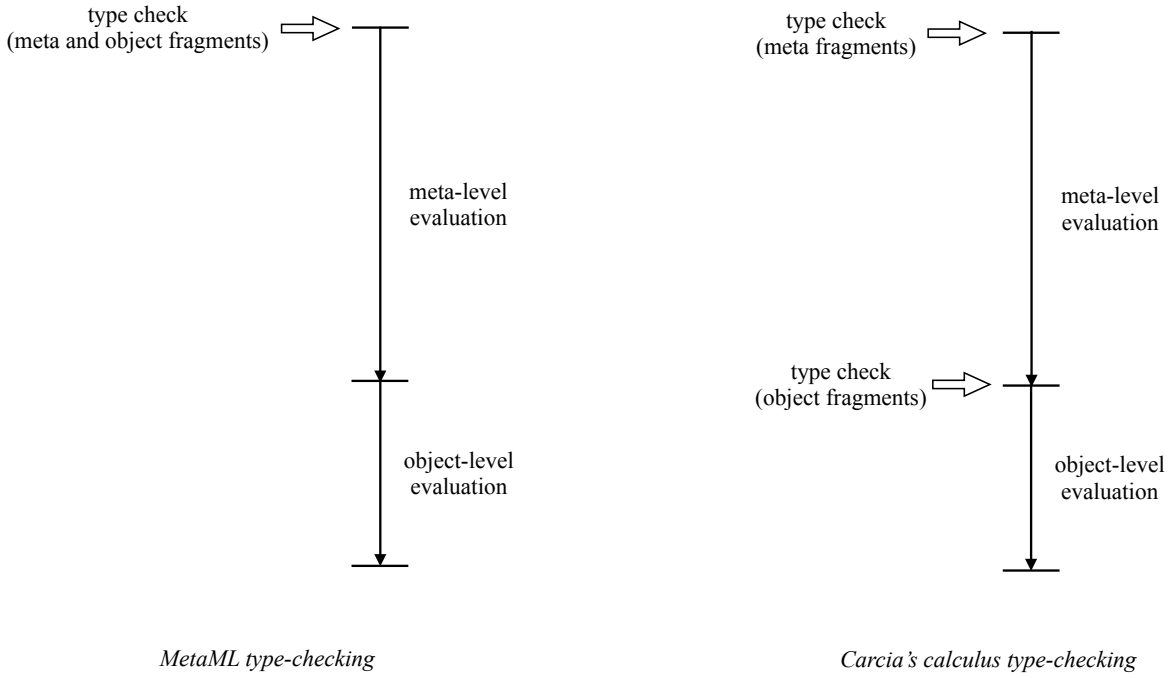
Although Garcia's type system is quite permissive, thereby providing expressiveness, the type system is not modular. That is, type checking a code fragment does not mean that all uses of the code fragment will be well-typed. One symptom of this is that type error messages do not necessarily point to the source of the problem (in the metaprogram) but instead point to the generated code. Thus, programmers spend time tracking down the source of type errors. Consider the following metaprogram in which function f_0 produces ill-typed code.

```
1  let meta f0 =
2      λt : type.
3          let meta id =
4              λx : bool. ≺ λx : (if x then t else bool) . x ≻
5          in
6              ≺ ∼ (id false) 1 ≻        // type error originates at here
7  in
8      let meta f1 = ≺ ∼ (f0 int) ≻ in
9          let meta f2 = ≺ ∼ f1 ≻ in
10             ... // many lines of code and the following fn is deeply nested in the let expressions
```

```
11          let meta f_n = ≺ ∼ f_{n−1} ≻ in
12              ∼ f_n;;
```

Inside the definition of f_0, the expression $\prec \sim$ (id **false**) 1 $\succ$ generates ill-typed code: (λx : **bool**. x) 1. So ideally we would like the compiler to report an error message that accurately points to this expression inside the definition of f_0. Instead, Garcia's type system accepts the definition and the type error is caught after meta-evaluation: after the metaprogram has generated a large amount of code of which the ill-typed code is a small part.

Figure 2.2: The comparison of type-checking strategies between MetaML and Garcia's calculus .



MetaML type-checking

Carcia's calculus type-checking

2.4 Summary

In this chapter, we give an overview of type-reflective metaprogramming and C++ templates, a language feature well-known for template metaprogamming. We discuss Garcia's calculus for type-reflective metaprogramming. The language incorporates the expressiveness of C++ templates but provides a weaker static type safety guarantee than MetaML.

Chapter 3

Incremental Type-Checking

Garcia's calculus [23] for type-reflective metaprogramming provides much of the power and flexibility of C++ templates. But, in Garcia's typing rules for type-checking type-reflective metaprograms, a residual program is not type checked until after meta computation is complete. Ideally, one would like the type system of the metaprogram to also guarantee that the residual program will type check, as is the case in MetaML [58]. However, in a language with type-reflective metaprogramming, type expressions in the residual program may be the result of meta computation, making the MetaML guarantee next to impossible to achieve.

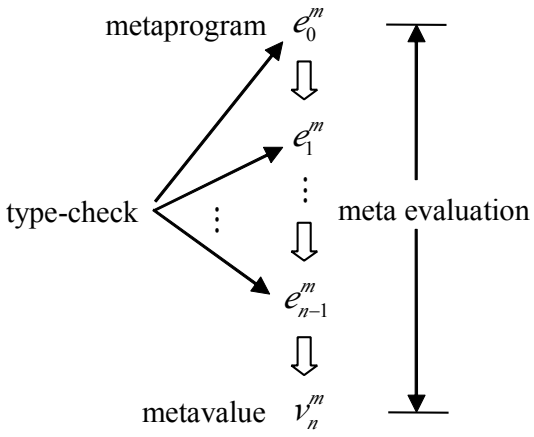
In this chapter, we offer incremental type-checking for checking type-reflective metaprograms. It type-checks object-level code before meta-evaluation, trying to catch errors that can be statically detected. Because the type of a code fragment can be result of meta-evaluation, during meta-evaluation it incrementally type checks code fragments as they are created and spliced together during meta computation. Compared with C++ templates and Template Haskell, incremental type-checking can detect errors earlier without sacrificing flexibility.

3.1 General Concept of Incremental Type-Checking

In our paper [40], we proposed type-checking metaprograms throughout meta-evaluation, thereby catching type errors as soon as there is enough type information to discover them. We call this type-checking approach incremental type-checking because it incrementally type-checks code fragments as they are produced and spliced together. Figure 3.1 shows a metaprogram e_0^m

that, through n steps of evaluation, becomes the value v_n^m. Incremental type-checking requires the code fragments to be type-checked thoroughly before meta-evaluation and between each step of computation.

Figure 3.1: Incremental type-checking



We can apply incremental type-checking to check object-level functions whose argument types are meta-level type expressions. The following object expression (written in Garcia's calculus syntax 2.1) is an example:

$\lambda \mathsf{x}_1 : \mathsf{e}_1^m.\ \lambda \mathsf{x}_2 : \mathsf{e}_2^m.\ \ldots\ \lambda \mathsf{x}_{n-1} : \mathsf{e}_{n-1}^m.\ \lambda \mathsf{x}_n : \mathsf{e}_n^m.\ \mathsf{e}$

Suppose all the variables are typed with type expressions that need to be evaluated. It is difficult to fully type-check the initial program before meta-evaluation, but after some small steps of meta-evaluation such that the type of x_1 is evaluated into some simple type τ. (i.e. in the below code, a simple type represents a type that is fully evaluated and it does not contain any type variables. The formal definition of simple type is in Figure 3.3). The evaluation results in the following:

$\lambda \mathsf{x}_1 : \tau_1^s.\ \lambda \mathsf{x}_2 : \mathsf{e}_2^m.\ \ldots\ \lambda \mathsf{x}_{n-1} : \mathsf{e}_{n-1}^m.\ \lambda \mathsf{x}_n : \mathsf{e}_n^m.\ \mathsf{e}$

we can then type-check the expression to discover if there is a type error caused by the typing of x_1, that is x_1 has type τ_1^s. If there is no type error, then we meta-evaluate the expression more steps forward until the simple type of x_2 is available and then we can type-check the expression again.

This continues until after all the type expressions are evaluated and the code eventually generated is definitely well-typed.

Next, we use the following toy example to illustrate incremental type-checking and show it can catch type-errors earlier than two-staged type-checking.

```
1  let meta f0 =
2      let meta h =
3          λx : int. ≺ λy: if x == 0 then bool else int. y/x ≻
4      in
5          ≺ ∼ (h 0) ≻
6  in
7      let meta f =
8          ... // many lines of code
9      in ∼ f0;;
```

In line 5, when h is applied to 0, it generates code $\prec \lambda y : \textbf{if } 0 == 0 \textbf{ then bool else int}.\ y/0 \succ$, which is evaluated into $\prec \lambda y : \textbf{bool}.\ y/0 \succ$. With incremental type-checking, because the actual type of y is available now, the type system re-type-checks the program and will find the piece of code is not well-typed. With two-staged type-checking, that piece of code is type-checked at the end of the program (line 9) after it has escaped from the brackets.

We can apply incremental type-checking to type-check the use of a meta-level if-expression whose two branches have different concrete types. The following abstract metaprogram is an example:

```
let meta x1 = if e11^m then ≺ e12 ≻ else ≺ e13 ≻ in
    let meta x2 = if e21^m then ≺ e22 ≻ else ≺ e23 ≻ in
        ... // intermediate meta−level let expressions are omitted.
            let meta xn = if en1^m then ≺ en2 ≻ else ≺ en3 ≻ in
                e
```

Suppose that for each of the meta-level if-expressions appearing in the metaprogram above, the type of the code fragment in the then branch is different from the type of the code fragment in

the else branch. Before meta-evaluation, it is difficult to fully type-check the entire metaprogram because the type system does not know the actual types of those if-expressions, so the types of the variables in the metaprogram are unknown. But after some meta-evaluation steps such that, for the first meta-level if expression, condition e_{11}^{m} is evaluated to some boolean value, one of the branches is selected, and the typing for variable x_1 becomes available; we can then type-check the metaprogram again to find out if there is a type error caused by the typing for variable x_1. So, the entire metaprogram is incrementally type-checked as more of the if-expressions are evaluated.

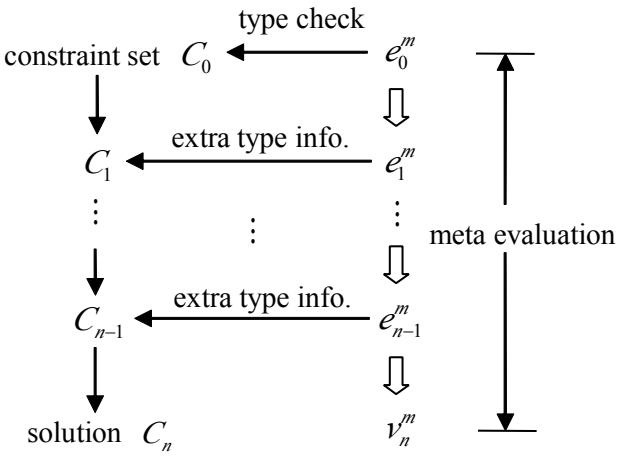
Next, we use the concrete example, function generic_add, to illustrate the incremental type-checking of the use of meta-level if-expressions whose branches have different concrete types. In the following metaprogram, function generic_add receives a type and returns some addition operator specific for that type if such addition operator is defined in generic_add; otherwise, it returns **None**.

```
1  let meta generic_add =
2      λt : type.
3          if (t =τ int) then (Some ≺ +int ≻)
4          else if (t =τ float) then (Some ≺ +float ≻)
5              else if (t =τ string) then (Some ≺ +string ≻)
6                  else if (t =τ list) then (Some ≺ append_list ≻)
7                      else (%(print_error "addition not supported"); None)
8  in
9      let meta add_int_pair_opt = (generic_add int) in
10         match add_int_pair_opt with None → None
11             | Some add_int_pair →
12                 let meta add_int = ≺ λx : list. λy : list. ((∼ add_int_pair) (x, y)) ≻ in
13                     let meta z =
14                         ... // many lines of code are omitted
15                     in
16                         Some ((∼ add_int) [1,2] [3,4])
```

For the above metaprogram, the type error is located in line 12: code ∼ add_int_pair is an object-

level function that accepts a pair of integers and returns the sum of both parts of the pair. But in the same line, the function is applied to a pair of lists, which is not well-typed. Because the return type of function generic_add depends on the input type, the compiler does not know the type of expression (generic_add **int**) until we evaluate the expression and obtain the evaluation result. After function application (generic_add **int**) is evaluated into **Some** $\prec +_{int} \succ$, the compiler can catch the type error: the type of variable add_int_pair_opt is (**code** (**int** $*$ **int** $\rightarrow$ **int**)) **option**, the type of add_int_pair is **code** (**int** $*$ **int** $\rightarrow$ **int**)); thus the type of ($\sim$ add_int_pair) is **int** $*$ **int** $\rightarrow$ **int**, which cannot be applied to a pair of lists, that is (x, y). With incremental type-checking, the compiler can catch the type error after a few steps of meta-evaluation. In contrast, with two-staged type-checking, the compiler catches the type error after finishing the meta-evaluation of the entire metaprogram.

Figure 3.2: Constraint-based incremental type-checking



Literally, type checking the entire metaprogram at every step of meta-evaluation would incur considerable performance cost. Instead, we derive a set of typing constraints (equalities between type expressions) from the metaprogram and then incrementally solve and update these constraints during meta-evaluation (see Figure 3.2). As meta-evaluation proceeds, more type information is available, which can add to the set of constraints. Solving the constraint set via unification simplifies the set and also can discover any type equality conflict, which indicates a type error. After meta-

evaluation, if the constraint set is still satisfiable, then the generated code is guaranteed to be well-typed.

3.2 Formalization

In this section, we formalize incremental type-checking in a calculus based on Garcia's calculus. We extend Garcia's calculus with existential types that have explicit type information for code fragments. This section presents the prototype of incremental type-checking. The prototype is not necessarily implementable, rather it is intended to be simple and straightforward, presenting a concise overview of incremental type-checking. We suppose that readers can easily understand the essence of incremental type-checking by learning its prototype.

3.2.1 Syntax

Figure 3.3 shows the core calculus of a type-reflective metaprogramming language. A large portion of the syntax is borrowed from Garicia's calculus for type-reflective metaprogramming. The syntax of proper code, written as e^o, represents a piece of object code generated by a metaprogram after meta-evaluation. Like Garcia's calculus, we simplify the calculus by integrating all of the type-reflective operations into one category: type primitives, written as $\mathcal{T}$. It contains a set of type-level operators, each of which can be applied to a sequence of type expressions. In the meta types, notation **code** is modified into $\exists\overline{\alpha}.\ \textsf{code}\ \tau$, where type τ means that, in our type system, it is a requirement to type-check a piece of object code thoroughly before meta-evaluation. The type variables appearing in τ are existentially quantified, which means that if a piece of code is well-typed and has meta-type $\exists\overline{\alpha}.\ \textsf{code}\ \tau$, then, during meta-evaluation, we can always find some object types to substitute those existentially quantified type variables in τ (substituting quantified type variables in a type for type instances is called **type instantiation**), and the resultant type from the instantiation is also the type of the code.

Figure 3.3: A core calculus for reflective metaprogramming and most part of the syntax is borrowed from Garcia's calculus. The difference is that in metatypes, we introduce existential types and the type for object-level code becomes explicit.

---

x	variables: meta-level and object-level variables
α	object type variables
c	value constants
$\mathcal{T}$	type primitives (e.g. **dom**, **cod**, $=_\tau$, $\gamma^?$, $\to^?$, **typeof**)
γ	type constants (e.g. **int**, **bool**)

simple types

$\tau^s ::= \gamma \mid \tau^s \to \tau^s$

object types

$\tau ::= \gamma \mid \alpha \mid \tau \to \tau$

object language

$e ::= c \mid x \mid \lambda x : e^m.\ e \mid e\ e \mid \sim e^m \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid$
$\textbf{let meta}\ x = e^m\ \textbf{in}\ e$

types for proper code

$\tau^o ::= \gamma \mid x \mid \tau^o \to \tau^o$

proper code

$e^o ::= c \mid x \mid \lambda x : \tau^o.\ e^o \mid e^o\ e^o \mid \textbf{let}\ x = e^o\ \textbf{in}\ e^o$

meta types

$\tau^m ::= \gamma \mid \exists\overline{\alpha}.(\textbf{code}\ \tau) \mid \textbf{type} \mid \tau^m \to \tau^m$

meta language

$e^m ::= c \mid \mathcal{T} \mid \gamma \mid e^m \to e^m \mid$
$x \mid \lambda x : \tau^m.\ e^m \mid e^m\ e^m \mid \prec e \succ \mid \%e^m \mid$
$\textbf{if}\ e^m\ \textbf{then}\ e^m\ \textbf{else}\ e^m$

meta values

$v^m ::= c \mid \gamma \mid v^m \to v^m \mid \lambda x : \tau^m.\ e^m \mid \prec e^o \succ$

3.2.2 Non-parametric Existential Types

We use existential types (written $\exists\alpha.\tau$, where α is an existentially quantified type variable that appears in type τ) to describe a meta expression, which has residual object code fragment whose type information is not complete or not available before meta-evaluation. However, we do not use the standard parametric interpretation of existential types. Instead, we use a non-parametric variation on existential types which generalizes dynamic types [53].

For parametric existential types, when an expression has a parametric existential type, we guarantee that there exist some type instances such that, after we substitute the existentially quantified type variables in the existential type with those type instances, the resultant type is also the type of the expression. For those existentially quantified type variables in a type, their type instances are statically determined, and thus do not change for different evaluation environments.

For the non-parametric version, we just assume the existence of type instances for those existentially quantified type variables. Such assumptions about type existence have to be verified during meta-evaluation. For example, the expression $\prec \lambda$x : **if true then int else bool**. x $\succ$ can have type $\exists\alpha$. **code** $(\alpha \to \alpha)$, which assumes that there exists some type instance for type variable α. During meta-evaluation, when the type of parameter x is evaluated into **int**, we then know that there exists some type instance for α and the type instance is **int**. There are some cases when the assumption of type existence is not true. For example, the following meta expression:

$\prec \lambda$x : (**if true then int else bool**) $\to$ **int**.
  λy : **typeof** (**if true then** $\prec$ 1 $>=$ 2 $\succ$ **else** $\prec$ 3 $\succ$).
    (x y) $\succ$

will have type $\exists\alpha$. **code** $((\alpha \to$ **int**$) \to \alpha \to$ **int**$)$ before meta-evaluation. But after the expression is evaluated into expression $\prec \lambda$x : **int** $\to$ **int**. λy : **bool**. (x y) $\succ$, we find out that there is no type instance for type variable α because of the following instantiation conflict: to match the type of parameter x, variable α should be instantiated into **int**, but to match the type of parameter y, variable α should be instantiated into **bool**. Obviously, **int** is not equal to **bool**.

The instantiation of a non-parametric existential type may depend on the evaluation result of a meta expression. Even for a single metaprogram, a non-parametric existential type can have different type instances as long as the generated code is well-typed. For instance, when an existential type resides in the type of a meta function, it may have different type instantiations with different meta function applications. Consider the following meta program:

```
1  ∼ (let meta f = λx : bool. ≺ λy : if x then int else string. y ≻ in
2        let meta g = ≺ ∼ (f true) 0 ≻ in
3           let meta h = ≺ ∼ (f false) "abc" ≻ in
4              ≺ (∼ g, ∼ h) ≻)
```

In line 1, we define a meta function that accepts a boolean and returns an object-level identity function. The type of the meta function can be **bool** $\rightarrow \exists\alpha.$ **code** $(\alpha \rightarrow \alpha)$. Type variable α is the type of parameter y and it can be instantiated into either **int** or **string**, depending on the evaluation result of type expression **if** x **then int else string**. The meta function is used in both line 2 and line 3. In line 2, the spliced code generates the object-level identity function, which is applied to integer 0. So, in line 2, α should be instantiated into **int** because the type of a parameter should be equal to the type of its argument. In line 3, α should be instantiated into **string** because the generated identity function is applied to a string. During meta-evaluation, we can verify that the code generated the above metaprogram is well-typed. In line 2, f is applied to **true**, thus type expression **if** x **then int else string** is evaluated into **int**, which is equal to the type instance for α in line 2. By contrast, in line 3, f is applied to **false** and the same type expression is evaluated into **string**, which is equal to the type instance for α in line 3. After meta-evaluation, the generated code is ((λy : **int**. y) 0, (λy : **string**. y) "abc") and it is well-typed.

In 2006, Siek and Taha proposed gradual typing [50, 53, 54]. It is the work related to incremental type-checking. For incremental type-checking, the compiler needs to dynamically (i.e. during meta-evaluation) type-check code fragments because their types can depend on meta expressions. In contrast, gradual typing gives a programmer control over which portions of a program should be dynamically type-checked based on the presence or absence of type annotations. In the

gradually-typed lambda calculus [53], the dynamic type, written ? for short, represents a type that is statically unknown. So, an expression whose type is assigned to ? means that the actual type of the expression is not available at compile time. In Section 3.1, we mentioned that we can type-check metafunction generic_add by using non-parametric existential types. The function implements a generic addition and has type **type** $\rightarrow$ ($\exists\alpha$. **code** ($\alpha * \alpha \rightarrow \alpha$)) **option**. Using dynamic types and gradual typing, we can also type-check the function. If the type-reflective metaprogramming language supports gradual typing, then the following implementation of the generic_add function can type-check:

```
1 /* generic_add : type → (code (? * ? → ?)) option */
2 let meta generic_add =
3     λt : type.
4         if (t =τ int) then (Some ⟨ code (? * ? → ?) ⟩ ≺ +int ≻)
5         else if (t =τ float) then (Some ⟨ code (? * ? → ?) ⟩ ≺ +float ≻)
6             else if (t =τ string) then (Some ⟨ code (? * ? → ?) ⟩ ≺ +string ≻)
7                 else if (t =τ list) then (Some ⟨ code (? * ? → ?) ⟩ ≺ append_list ≻)
8                     else (%(print_error "addition not supported"); None)
```

The syntax $\langle\tau^m\rangle e^m$ means meta-level type cast. In the above metaprogram, the type-specific addition operations are type cast into one common meta type: **code** (? * ? $\rightarrow$?), and thus the function has type **type** $\rightarrow$ (**code** (? * ? $\rightarrow$?)) **option**. In gradual typing, the use of type equality is replaced by the type consistency relation, written $\sim$. The dynamic type is consistent with any ground types (or called type constants) but two different ground types are inconsistent with each other. Two function types are consistent if their domains and codomains are consistent, respectively. So, the type of any addition operation is consistent with type ? * ? $\rightarrow$?.

Existential types can capture more precise information about an expression than types that use the dynamic type. For example, type ? * ? $\rightarrow$? cannot capture the type feature of a binary operation. Mathematically, a binary operation maps a set of $S \times S$ to S where S is a non-empty

set [1] . In computer science, a binary operation receives a pair, and the elements in the pair should have the same type as the return value. Type $? * ? \to ?$ describes a set of expressions that are more general than binary operations: the type of a function is consistent with $? * ? \to ?$ as long as the function accepts a pair as its first or only argument, for instance, a function of type **int** $*$ **string** $\to$ **int** and a function of type **int** $*$ **int** $\to$ **int** $\to$ **int**. Such inaccuracy might prevent us from discovering type errors at a early stage (for instance, before meta-evaluation). For example, consider the following metaprogram:

```
1  let meta generic_add_app =
2      λt : type.
3          match (generic_add t) with None → None
4              | Some add_t → Some ≺ (∼ add_t) (1, "one") ≻
5  in
6      let v =
7          match (generic_add_app int) with None → ≺ None ≻
8              | Some c → ≺ Some (∼ c) ≻
9      in
10         ∼ v
```

In the body of function generic_add_app (line 2-4), the program uses function generic_add and the generated addition operation is applied to pair (1, "one"). According to the definition of a binary operation, it only accepts a pair whose elements have the same type. So, the definition of generic_add_app is not well-typed no matter what type the function is applied to. Ideally, we would like the type system to catch the type error when type-checking the function's definition (i.e. the type error message should point to line 2). Using gradual typing, because the dynamic type is consistent with any type, the type system cannot catch the type error statically. Instead, the type system catches the type error in the generated code: $+_{int}$ (1, "one"). In contrast, the type system can catch the type error in time and at the right place if it uses incremental type-checking. With non-parametric existential types, the type of function generic_add is **type** $\to$ ($\exists\alpha$. **code** ($\alpha * \alpha \to \alpha$

[1] http://en.wikipedia.org/wiki/Binary_operation

)) **option**. Thus, the type of add_t (see line 4) is $\exists\alpha.$ **code** $(\alpha * \alpha \to \alpha)$. When type-checking code $(\sim$ add_t$)$ (1, "one"), the type-checker needs to find a type instance for α (i.e. for the instantiation of the existential type) to make the code well-typed, but there does not exist an object type τ such that $\tau * \tau$ is equal to **int** $*$ **string**.

3.2.3 Typing Rules

In this section, we present the type rules. First, we give some preliminary definitions for typing. We define $[\tau/\alpha](\cdot)$ as the substitution of a free object type variable for an object type in a metatype, an object type, a meta expression, or an object expression. So, $[\overline{\tau}/\overline{\alpha}](\cdot)$, written ϕ for short, is the simultaneous substitution of free object type variables for object types.

Figure 3.4 shows the type consistency and the naive subtyping [61] relations for the metatypes. For example, from rule C-Ext we know metatypes $\exists\alpha.$ **code** $(\alpha \to$ **int**$)$ and $\exists\alpha,\beta.$ **code** $(\beta \to \alpha)$ are consistent because we can find two substitutions $[\beta/\alpha]$ and [**int**$/\alpha]$ such that $[\beta/\alpha](\alpha \to$ **int**$)$ is equal to [**int**$/\alpha](\beta \to \alpha)$.

Naive subtyping is covariant in the domain of function types (shown from rule S-Fun) instead of contravariant like normal subtyping. Rule S-Ext defines the naive subtype relation of two existential metatypes. Suppose we have two different existential metatypes τ_1^m and τ_2^m, and if $\tau_1^m <:_n \tau_2^m$, then τ_1^m must be more specific than τ_2^m. For example, we know that $\exists\alpha.$ **code** $(\alpha \to \alpha)$ is a naive subtype of $\exists\alpha,\beta.$ **code** $(\alpha \to \beta)$ because there is a substitution $[\alpha/\beta]$ such that $(\alpha \to \alpha) = [\alpha/\beta](\alpha \to \beta)$. Obviously, the first one is more specific than the second one because the first one refers to the set of unary operations whose domain and codomain must have the same type while the second refers to any kind of unary functions.

We say that τ^m is an upper bound of τ_1^m and τ_2^m if $\tau_1^m <:_n \tau^m$ and $\tau_2^m <:_n \tau^m$. Two metatypes do not necessarily have an upper bound. For example, **int** and **bool** do not have an upper bound. However, any two existential metatypes must have at least one upper bound, that is $\exists\alpha.$ **code** α. Suppose τ_1^m and τ_2^m have some upper bound(s), then we define $\tau_1^m \vee_n \tau_2^m$ as the least upper bound of τ_1^m and τ_2^m. We define $\vee_n$ as follows:

Figure 3.4: This figure shows type consistency and naive subtyping relations for metatypes

Type consistency $\boxed{\tau^m \sim \tau^m}$

$$\text{(C-Con)}\frac{}{\gamma \sim \gamma} \qquad \text{(C-Fun)}\frac{\tau_1^m \sim \tau_3^m \qquad \tau_2^m \sim \tau_4^m}{\tau_1^m \to \tau_2^m \sim \tau_3^m \to \tau_4^m}$$

$$\text{(C-Typ)}\frac{}{\mathbf{type} \sim \mathbf{type}} \qquad \text{(C-Ext)}\frac{\exists \phi_1 . \exists \phi_2 .\ \phi_1(\tau_1) = \phi_2(\tau_2)}{\exists \overline{\alpha}.(\mathbf{code}\ \tau_1) \sim \exists \overline{\beta}.(\mathbf{code}\ \tau_2)}$$

Naive subtyping $\boxed{\tau^m <:_n \tau^m}$

$$\text{(S-Con)}\frac{}{\gamma <:_n \gamma} \qquad \text{(S-Fun)}\frac{\tau_1^m <:_n \tau_3^m \qquad \tau_2^m <:_n \tau_4^m}{\tau_1^m \to \tau_2^m <:_n \tau_3^m \to \tau_4^m}$$

$$\text{(S-Typ)}\frac{}{\mathbf{type} <:_n \mathbf{type}} \qquad \text{(S-Ext)}\frac{\exists \phi .\ \tau_1 = \phi(\tau_2)}{\exists \overline{\alpha}.(\mathbf{code}\ \tau_1) <:_n \exists \overline{\beta}.(\mathbf{code}\ \tau_2)}$$

τ^m is $(\tau_1^m \vee_n \tau_2^m)$ iff

i. $(\tau_1^m <:_n \tau^m) \wedge (\tau_2^m <:_n \tau^m)$, **and**

ii. $\forall \tau_t^m.\ (\tau_1^m <:_n \tau_t^m) \wedge (\tau_2^m <:_n \tau_t^m) \Rightarrow (\tau^m <:_n \tau_t^m)$.

For example, consider the following three metatypes.

$\exists \alpha.$ **code** α

$\exists \alpha, \beta.$ **code** $(\alpha \to \beta)$

$\exists \alpha.$ **code** $(\alpha \to \alpha)$

They are all the upper bounds of **code** (**int** $\to$ **int**) and **code** (**bool** $\to$ **bool**), but only $\exists \alpha.$ **code** $(\alpha \to \alpha)$ is the least upper bound of the two and it is also the most specific among all the upper bounds.

Figure 3.5 shows a selection of the typing rules for the object language and the meta language. We borrowed the rules for typing constants and type primitives from Garica's paper [23]. Typing rules T-Const and M-Const type-check expression constants. We provide function *type* that maps constants to their types, for instance, booleans are mapped to type **bool**, string constants are mapped to type **string**, integers are mapped to type **int**, and so on. We omit the typing rules for type primitives $\to^?$, **cod**, and $=^\tau$ as those rules are conventional and straightforward.

For object-level functions, when the type annotation of a function parameter is not a simple type (not yet evaluated), we allow the parameter to take any type. Consider the following type rule, where e^m is some type expression but not a simple type, that is $\nexists \tau^s.(\tau^s = e^m)$, therefore the type of x is allowed to be any object type τ_1 that allows the body e to type check.

$$\frac{\Gamma; \Psi \vdash e^m : \mathbf{type} \qquad \Gamma; \Psi \vdash \tau_1 \text{ ok} \qquad \Gamma, x : \tau_1; \Psi \vdash e : \tau_2}{\Gamma; \Psi \vdash \lambda x : e^m.\ e : \tau_1 \to \tau_2}$$

Further along in the meta computation, when the type of the parameter becomes a simple type, the following rule applies instead.

$$\frac{\Gamma, x : \tau^s; \Psi \vdash e : \tau}{\Gamma; \Psi \vdash \lambda x : \tau^s.\ e : \tau^s \to \tau}$$

Typing rule M-Code fully type-checks the residual object expression inside the brackets and it introduces existential types as a variant of the gradual type for a piece of code by existentially

Figure 3.5: Selected typing rules from the type system for reflective metaprogramming calculus.

Object-level typing context $\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha$

Meta-level typing context $\Psi ::= \cdot \mid \Psi, x : \tau^m$

Well-formed object types: $\boxed{\Gamma; \Psi \vdash \tau \text{ ok}}$

$$\frac{}{\Gamma; \Psi \vdash \gamma \text{ ok}} \qquad \frac{\alpha \in \Gamma}{\Gamma; \Psi \vdash \alpha \text{ ok}} \qquad \frac{\Gamma; \Psi \vdash \tau_1 \text{ ok} \quad \Gamma; \Psi \vdash \tau_2 \text{ ok}}{\Gamma; \Psi \vdash \tau_1 \to \tau_2 \text{ ok}}$$

Well-typed object expressions: $\boxed{\Gamma; \Psi \vdash e : \tau}$

$$\text{(T-Const)}\frac{\mathit{type}(c) = \gamma}{\Gamma; \Psi \vdash c : \gamma} \qquad \text{(T-Var)}\frac{x : \tau \in \Gamma}{\Gamma; \Psi \vdash x : \tau} \qquad \text{(T-Abs1)}\frac{\Gamma, x : \tau^s; \Psi \vdash e : \tau}{\Gamma; \Psi \vdash \lambda x : \tau^s.\ e : \tau^s \to \tau}$$

$$\text{(T-Abs2)}\frac{\Psi \vdash e^m : \mathbf{type} \quad \Gamma; \Psi \vdash \tau_1 \text{ ok} \quad \Gamma, x : \tau_1; \Psi \vdash e : \tau_2}{\Gamma; \Psi \vdash \lambda x : e^m.\ e : \tau_1 \to \tau_2}$$

$$\text{(T-App)}\frac{\Gamma; \Psi \vdash e_1 : \tau \to \tau_0 \quad \Gamma; \Psi \vdash e_2 : \tau}{\Gamma; \Psi \vdash e_1\ e_2 : \tau_0} \qquad \text{(T-Esp)}\frac{\Gamma; \Psi \vdash e^m : \exists \overline{\alpha}.(\mathbf{code}\ \tau) \quad \Gamma; \Psi \vdash \overline{\tau} \text{ ok}}{\Gamma; \Psi \vdash \sim e^m : [\overline{\tau}/\overline{\alpha}]\tau}$$

$$\text{(T-Let)}\frac{\Gamma; \Psi \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1; \Psi \vdash e_2 : \tau_2}{\Gamma; \Psi \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2} \qquad \text{(T-MLet)}\frac{\Gamma; \Psi \vdash e^m : \tau^m \quad \Gamma; \Psi, x : \tau^m \vdash e : \tau}{\Gamma; \Psi \vdash \mathbf{let\ meta}\ x = e^m\ \mathbf{in}\ e : \tau}$$

Well-typed meta expressions: $\boxed{\Gamma; \Psi \vdash e^m : \tau^m}$

$$\text{(M-Const)}\frac{\mathit{type}(c) = \gamma}{\Gamma; \Psi \vdash c : \gamma} \qquad \text{(M-Dom)}\frac{\Gamma; \Psi \vdash e^m : \mathbf{type}}{\Gamma; \Psi \vdash \mathbf{dom}\ e^m : \mathbf{type}} \qquad \text{(M-Grd)}\frac{\Gamma; \Psi \vdash e^m : \mathbf{type}}{\Gamma; \Psi \vdash \gamma^?\ e^m : \mathbf{bool}}$$

$$\text{(M-TypeOf)}\frac{\Gamma; \Psi \vdash e^m : \exists \overline{\alpha}.(\mathbf{code}\ \tau)}{\Gamma; \Psi \vdash \mathbf{typeof}\ e^m : \mathbf{type}} \qquad \text{(M-FunTy)}\frac{\Gamma; \Psi \vdash e_1^m : \mathbf{type} \quad \Gamma; \Psi \vdash e_2^m : \mathbf{type}}{\Gamma; \Psi \vdash e_1^m \to e_2^m : \mathbf{type}}$$

$$\text{(M-Var)}\frac{x : \tau^m \in \Psi}{\Gamma; \Psi \vdash x : \tau^m} \qquad \text{(M-Abs)}\frac{\Gamma; \Psi \vdash \tau^m : \mathbf{type} \quad \Gamma; \Psi, x : \tau^m \vdash e^m : \tau_0^m}{\Gamma; \Psi \vdash \lambda x : \tau^m.e^m : \tau_0^m}$$

$$\text{(M-App)}\frac{\Gamma; \Psi \vdash e_1^m : \tau_1^m \to \tau^m \quad \Gamma; \Psi \vdash e_2^m : \tau_2^m \quad \tau_1^m \sim \tau_2^m}{\Gamma; \Psi \vdash e_1^m\ e_2^m : \tau^m}$$

$$\text{(M-Code)}\frac{\Gamma, \overline{\alpha}; \Psi \vdash e : \tau \quad \overline{\alpha} \sharp\ ftv(\Gamma; \Psi)}{\Gamma; \Psi \vdash \prec e \succ : \exists \overline{\alpha}.(\mathbf{code}\ \tau)} \qquad \text{(M-Lift)}\frac{\Gamma; \Psi \vdash e^m : \gamma}{\Gamma; \Psi \vdash \%e^m : \mathbf{code}\ \gamma}$$

$$\text{(M-If)}\frac{\Gamma; \Psi \vdash e_1^m : \mathbf{bool} \quad \Gamma; \Psi \vdash e_2^m : \tau_2^m \quad \Gamma; \Psi \vdash e_3^m : \tau_3^m \quad \exists \tau^m.\ \tau^m = \tau_2^m \vee_n \tau_3^m}{\Gamma; \Psi \vdash \mathbf{if}\ e_1^m\ \mathbf{then}\ e_2^m\ \mathbf{else}\ e_3^m : \tau^m}$$

quantifying the object type variables appearing in the type of the object expression. In typing rule M-Code, function ftv collects all the free object type variables appearing in the type environments and returns a set of object type variables. Proposition $\overline{\alpha} \sharp ftv(\Gamma, \Psi)$ specifies that the set of the object type variables appearing in $\overline{\alpha}$ is disjoint with the set of the free object type variables appearing in Γ and Ψ.

Typing rule T-Esp type-checks an escape of a piece of code. When a piece of code is escaped, we instantiate the existential type by substituting those quantified type variables for some witness types that we infer, so that the instantiated type is the type of the escaped code. A piece of code can be "polymorphic" as the existential type can be instantiated into different types. For example, in the following metaprogram:

```
1 let meta f =
2     λt : bool. ≺ λx : if t then int else bool. x ≻
3 in
4     (≺ ∼ (f true) 0 ≻,
5      ≺ ∼ (f false) true ≻)
```

The meta-level function f is used in a pair of code fragments and we assume the meta language is extended with the pair constructor. By rule T-Abs1 and rule M-Code, function f will have the type:

$$\textbf{bool} \to \exists\alpha.(\textbf{code}\ (\alpha \to \alpha))$$

In line 4 the expression of code escape, $\sim$ (f **true**), should have type **int** $\to$ **int** by rule T-Esp: because the expression is applied to integer 0, for type variable α, the only witness type we can pick to make the whole metaprogram well-typed is **int**, and thus α is instantiated into **int**. In contrast, in line 5, expression $\sim$ (f **false**) has type **bool** $\to$ **bool** because it is applied to boolean **true** and α is instantiated into **bool**.

In typing rule M-App, if a meta-level function application is well-typed, then the parameter type is consistent with the type of the argument. Using type consistency enables the compiler to rule out some meta-level function applications that are not statically well-typed. For example, the

following metaprogram is not well-typed:

```
1  let meta f =
2      λx : ∃α.(code (int → α)). ≺ (∼ x) 0 ≻
3  in
4      let meta c =
5          ≺ λy : bool. y ≻
6      in
7          (f c)
```

because in line 7, variable c has type **code** (**bool** → **bool**), but it is not consistent with the argument type of meta-level function f, which is $\exists\alpha.$(**code** (**int** → α)). So the meta-level function application f c is not well-typed. However, using type-consistency enables a meta-level function to accept some code fragments whose types are partially or totally unavailable before meta-evaluation. Consider the following metaprogram:

```
1  let meta f =
2      λx : ∃α.(code (int → α)).
3          ≺ (∼ x) 0 ≻
4  in
5      let meta c =
6          ≺ λy : if true then int else bool. y ≻
7      in
8          (f c)
```

because the argument type inside code expression ≺ λy : **if true then int else bool**. y ≻ is unknown before meta-evaluation, we can type-check the code expression to have type $\exists\alpha.$(**code** ($\alpha \to \alpha$)), which is consistent with the argument type of meta-level function f. So, in line 8, meta-level function application f c is well-typed. During meta-evaluation, type expression **if true then int else bool** is evaluated into **int** and f c is evaluated into ≺ (λy : **int**. y) 0 ≻, which is also well-typed.

In typing rule M-If, meta-level **if** expression can be dynamically typed as it allows two branches to have different metatypes but the metatypes should have the least upper bound of the two branch

types.

3.2.4 Meta-evaluation Rules

We formalize the meta-evaluation of our type-reflective metaprogramming calculus with small-step operational semantics. An evaluation context is an expression with a hole, denoted $\square$. Suppose E is an evaluation context and e is an expression, then the operation of placing e (called redex) into the hole of context E is called plugging, expressed with the notation $E[e]$. An evaluation context with a redex plugged in (i.e. $E[e]$) is reduced into some expression, while a non-value expression can be decomposed into an evaluation context and a redex. Evaluation contexts are used specifically to define program evaluation steps, for instance:

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$$

and the reduction relation $e \longrightarrow e'$ specifies how computations are performed.

Figure 3.6 shows the selected evaluation contexts for our type-reflective metaprogramming calculus. We adopt the presentation of evaluation contexts for a two-staged programming language from Garcia's calculus [23]. In the figure, let E be an evaluation context whose hole is in an object context (called an object evaluation context) and E^m be an evaluation context whose hole is in a meta context (called a meta-evaluation context). Because the object and the meta language are interleaved, the evaluation contexts are also interleaved, which means the object and the meta-evaluation contexts are mutually recursive. Except for the empty evaluation context, all the other evaluation contexts are defined in the form of the composition of two evaluation contexts. We use meta-evaluation context to explain how an evaluation context works, and object evaluation context adopts the same mechanism. A meta-evaluation context E^m has an outer evaluation context and an inner evaluation context. The hole of the inner context accepts a meta expression. The evaluation result of the inner context is plugged into the outer context, thus the inner context should be consistent with the outer context. For example, meta-evaluation context $E^m[\square\ e^m]$ has the outer meta context E^m and the inner meta context for evaluating meta function application: $\square\ e^m$

where the hole accepts a meta expression. Context $E[\lambda x : \square.e]$ is also a meta-evaluation context: although its outer and inner contexts are object context, the hole is located at the meta-level, which should accept a meta expression.

Figure 3.6: Selected evaluation contexts.

meta-evaluation contexts E^m

$$
\begin{array}{rcl}
E^m & ::= & E^m[\mathbf{dom}\ \square] \mid E^m[\mathbf{cod}\ \square] \mid E^m[\gamma^?\ \square] \mid E^m[\rightarrow^?\ \square] \mid \\
& & E^m[\square =^\tau e^m] \mid E^m[\tau^s =^\tau \square] \mid E^m[\mathbf{typeof}\ \square] \mid E^m[\square \rightarrow e^m] \mid \\
& & E^m[\tau^s \rightarrow \square] \mid E^m[\square\ e^m] \mid E^m[v^m\ \square] \mid E^m[\%\square] \mid \\
& & E^m[\mathbf{if}\ \square\ \mathbf{then}\ e^m\ \mathbf{else}\ e^m] \mid E[\lambda x : \square.e] \mid E[\sim \square] \mid \\
& & E[\mathbf{let\ meta}\ x = \square\ \mathbf{in}\ e]
\end{array}
$$

Object evaluation contexts E

$$
\begin{array}{rcl}
E & ::= & \square \mid E[\lambda x : \tau^s.\square] \mid E[\square\ e] \mid E[e^o\ \square] \mid E[\mathbf{let}\ x = \square\ \mathbf{in}\ e] \mid \\
& & E[\mathbf{let}\ x = e^o\ \mathbf{in}\ \square] \mid E^m[\prec \square \succ]
\end{array}
$$

Figure 3.7 shows the reduction rules. We define notion $[e^m/x]e$ as the substitution of x in e for e^m, and notation $[e^m/x]e_0^m$ as the substitution of x in e_0^m for e^m. For type inquiry, $\mathbf{typeof}\ \prec e^o \succ$ (suppose it resides in the evaluation context E^m), its reduction rule uses the type-checker to obtain the type of e^o. When obtaining the type of e^o, we know e^o has already been type-checked and it must be well-typed; otherwise, the compiler will generate a type error before evaluating $\mathbf{typeof}\ \prec e^o \succ$. To obtain the type of e^o, we need to know the typing environments. So, we use function *Ctxt* to generate the object typing environments (the meta typing environment is always empty) from E^m. If *Ctxt* is applied to some context $E^m[\sigma^m]$ where σ^m is an inner meta-evaluation context, then the result is $\mathit{Ctxt}(E^m)$. If *Ctxt* is applied to $\square$, then the result is $\cdot$ (i.e. an empty typing environment). If *Ctxt* is applied to some context $E[\sigma]$ where σ is an inner object evaluation context but is not in the syntactic form of $\lambda x : \tau^s.\square$ or $\mathbf{let}\ x = e^o\ \mathbf{in}\ \square$, then the result is $\mathit{Ctxt}(E)$. If *Ctxt* is applied to $E[\lambda x : \tau^s.\square]$, then the result is $\mathit{Ctxt}(E), x : \tau^s$. If *Ctxt* is applied to $E[\mathbf{let}\ x = e^o\ \mathbf{in}\ \square]$, then the result is $\mathit{Ctxt}(E), x : \tau^s$ where τ^s is the type of e^o: $\mathit{Ctxt}(E); \cdot \vdash e^o : \tau^s$. The rest of the reduction rules in Figure 3.7 are self-explanatory.

Figure 3.7: The reduction rules.

Object expression reduction $\boxed{e \longrightarrow e'}$

$$\sim \prec e^o \succ \longrightarrow e^o \qquad \textbf{let meta } x = v^m \textbf{ in } e \longrightarrow [v^m/x]e$$

Meta expression reduction $\boxed{e^m \longrightarrow e^{m\prime} \text{ in } E^m}$

$$\textbf{dom } (\tau_1 \to \tau_2) \longrightarrow \tau_1 \text{ in } E^m \qquad \textbf{cod } (\tau_1 \to \tau_2) \longrightarrow \tau_2 \text{ in } E^m$$

$$\tau =^\tau \tau \longrightarrow \textbf{true} \text{ in } E^m \qquad \frac{\tau_1 \neq \tau_2}{\tau_1 =^\tau \tau_2 \longrightarrow \textbf{false} \text{ in } E^m}$$

$$\gamma^? \gamma \longrightarrow \textbf{true} \text{ in } E^m \qquad \gamma^? (\tau_1 \to \tau_2) \longrightarrow \textbf{false} \text{ in } E^m$$

$$\to^? \gamma \longrightarrow \textbf{false} \text{ in } E^m \qquad \to^? (\tau_1 \to \tau_2) \longrightarrow \textbf{true} \text{ in } E^m$$

$$\frac{\mathit{Ctxt}(E^m) = \Gamma \qquad \Gamma; \cdot \vdash e^o : \tau}{\textbf{typeof } e^o \longrightarrow \tau \text{ in } E^m}$$

$$(\lambda x : \tau^m . e^m)\ v^m \longrightarrow [v^m/x]e^m \text{ in } E^m \qquad \%c \longrightarrow \prec c \succ \text{ in } E^m$$

$$\textbf{if true then } e_1^m \textbf{ else } e_2^m \longrightarrow e_1^m \text{ in } E^m \qquad \textbf{if false then } e_1^m \textbf{ else } e_2^m \longrightarrow e_2^m \text{ in } E^m$$

The single-step meta-evaluation rules are defined in Figure 3.8. As we mentioned perviously, a non-value expression can be decomposed into an evaluation context and a redex (also not a value); and a single-step evaluation of the expression is the reduction of the redex. Rule OCtxt-Eval shows the case when a single-step evaluation is the reduction of an object-level redex. We do not need to type-check the result of such single-step evaluation because the computation of an object expression preserves type (i.e. an object expression does not manipulate types). The next four rules show the case when a single-step evaluation is the reduction of a meta-level redex. In more detail, rule MCtxt-Eval1 and MCtxt-Eval2 are for the evaluation of meta expressions, for example, (**if** $\square$ **then int else bool**)[$\gamma^?$ **bool**], while rule MCtxt-Eval3 and MCtxt-Eval4 are for the evaluation of object expressions, for example, (λ x : $\square$. (x + 1))[**dom** (**bool** $\rightarrow$ **int**)]. Because the reduction of a meta expression may generate/change type information, we need to type-check the evaluation result: meta-evaluation results in **error** as soon as a type error is detected. Consider the following program:

```
1  λx : int → bool.
2      λy : cod (typeof ≺ x ≻).
3          let meta z = fibonacci(100) in      // function fibonacci(n) as time complexity: 2^n
4              if x(z) then 2∗y else y
```

and the program can be composed by the following redex and the meta-evaluation context: the redex is **typeof** $\prec$ x $\succ$ and the context E_0^m is

```
□[λ x : int → bool.
    λy : □.
      let meta z = fibonacci(100) in
        if x(z) then 2∗y else y][cod □]
```

By function *Ctxt*, we can obtain the type of variable x from E_0^m. The type is **int** $\rightarrow$ **bool**, which is also the value of **type** $\prec$ x $\succ$. After the meta reduction is performed, the whole program E_0^m[**int** $\rightarrow$ **bool**] needs to be type-checked to make sure it is still well-typed. Next, we decompose E_0^m[**int** $\rightarrow$ **bool**] into redex **cod** (**int** $\rightarrow$ **bool**) and context E_1^m:

```
□[λ x : int → bool.
    λy : □.
      let meta z = fibonacci(100) in
        if x(z) then 2∗y else y]
```

After **cod** (**int** → **bool**) is reduced to **bool**, we type-check $E_1^m[\mathbf{bool}]$, which is the following function:

```
λx : int → bool.
  λy : bool.
    let meta z = fibonacci(100) in
      if x(z) then 2∗y else y
```

and the function is not well-typed: y has type **bool** but in line 4 it is used as an integer. This example shows that the type error caused by variable y can be caught as soon as the type of y is available and the fibonacci function in the next line does not need to be executed.

Figure 3.8: The single-step meta-evaluation rules.

Single-step meta-evaluation for E $\quad \boxed{E[e] \longmapsto E[e']}$

$$(\text{OCtxt-Eval})\ \frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$$

Single-step meta-evaluation for E^m $\quad \boxed{E^m[e^m] \longmapsto E^m[e^{m\prime}] \text{ or } \mathbf{error}}$

$$(\text{MCtxt-Eval1})\ \frac{e^m \longrightarrow^m e^{m\prime} \text{ in } E^m \quad \Gamma; \Psi \vdash E^m[e^{m\prime}] : \tau^m}{E^m[e^m] \longmapsto E^m[e^{m\prime}]} \qquad (\text{MCtxt-Eval2})\ \frac{e^m \longrightarrow^m e^{m\prime} \text{ in } E^m \quad \Gamma; \Psi \nvdash E^m[e^{m\prime}] : \tau^m}{E^m[e^m] \longmapsto \mathbf{error}}$$

$$(\text{MCtxt-Eval3})\ \frac{e^m \longrightarrow^m e^{m\prime} \text{ in } E^m \quad \Gamma; \Psi \vdash E^m[e^{m\prime}] : \tau}{E^m[e^m] \longmapsto E^m[e^{m\prime}]} \qquad (\text{MCtxt-Eval4})\ \frac{e^m \longrightarrow^m e^{m\prime} \text{ in } E^m \quad \Gamma; \Psi \nvdash E^m[e^{m\prime}] : \tau}{E^m[e^m] \longmapsto \mathbf{error}}$$

3.2.5 Soundness

We have given the formalization of the incremental type system for a type-reflective metaprogramming language, we next reason about its behavior, in particular we show that the type system is sound. The detailed proof for the following properties is listed in the Appendix.

Theorem 1 (Progress)**.**

(1) If e is closed and $\Gamma; \Psi \vdash e : \tau$, then e is a proper object code, or there is some e' such that $e \longmapsto e'$, or $e \longmapsto$ **error**.

(2) If e^m is closed and $\Gamma; \Psi \vdash e^m : \tau^m$, then e^m is a meta value, or there is some $e^{m\prime}$ such that $e^m \longmapsto e^{m\prime}$, or $e^m \longmapsto$ **error**.

Proof. Straightforward induction on typing derivations. □

Theorem 2 (Preservation)**.**

(1) If $\Gamma; \Psi \vdash e : \tau$ and $e \longmapsto e'$, then $\Gamma; \Psi \vdash e' : \tau'$.

(2) If $\Gamma; \Psi \vdash e^m : \tau^m$ and $e^m \longmapsto e^{m\prime}$, then $\Gamma; \Psi \vdash e^{m\prime} : \tau^{m\prime}$.

Proof. Straightforward induction on typing derivations. □

Theorem 3 (Type Safety)**.**

(1) If $\Gamma; \Psi \vdash e : \tau$ and $e \longmapsto^* e'$, then $\Gamma; \Psi \vdash e' : \tau'$, and e' is a proper object code, or $\exists e''.e' \longmapsto e''$, or $e' \longmapsto$ **error**.

(2) If $\Gamma; \Psi \vdash e^m : \tau^m$ and $e^m \longmapsto^* e^{m\prime}$, then $\Gamma; \Psi \vdash e^{m\prime} : \tau^{m\prime}$, and $e^{m\prime}$ is a meta value, or $\exists e^{m\prime\prime}.e^{m\prime} \longmapsto e^{m\prime\prime}$, or $e^{m\prime} \longmapsto$ **error**.

Proof. Using Progress and Preservation. □

3.3 Implementation

In this section, we present a unification-based implementation of incremental type-checking. In general, unification[2] is the process of solving the satisfiability problem. It is used during type inference for solving type equality constraints and generating type variable substitutions. Similar to type inference, incremental type-checking also generates type equality constraints and solves the constraints via unification.

First, we introduce the syntax of the language in which incremental type-checking is implemented. The language has two parts: the surface language and the kernel language. The surface language is close to the user language, which provides rich language syntax for writing expressive programs in a convenient way. The compiler translates the surface language into the kernel language. The kernel language syntax is concise, capturing the kernel features of a programming language, but can be attached with some explicit information for internal use only. In practice, the concise kernel syntax makes ti easy for us to explain the technical part of the language and also easy for readers to understand.

3.3.1 Surface Language

Figure 3.9 shows the core syntax of the surface language. The surface language provides rich syntax for users to write expressive programs. In the figure, the syntax of meta-level if-expressions has two forms. Syntax **if** e^m **then** e^m **else** e^m **withtype** τ^m enables programmers to specify the type of an if-expression, where τ^m is the specified type. So far, we have not found an effective algorithm to compute the least upper bound of two meta types, thus we ask programmers to give the meta type of an if-expression, which is expected to be the least upper bound of the types of two branches. But it is also acceptable if the given type is not the least upper bound. Syntax **if** e^m **then** e^m **else** e^m represents an if-expression that programmers do not give its type. In this case, the compiler infers the type, which is an upper bound of the types of two branches, but the

[2] Detailed explanation of unification is available at http://en.wikipedia.org/wiki/Unification_(computer_science).

inferred type is not guaranteed to be the least upper bound of the types of two branches.

3.3.2 Kernel Language

Figure 3.10 shows the syntax of the kernel language. The kernel language captures the core language features of type-reflective metaprogramming, but also extends some parts of the surface language syntax with type annotations and type constraints for unification-based type-checking.

A type constraint C is a set, an element of which can be either the equality of two object types, the equality of two meta types, the consistency of two object types, or the consistency of two metatypes. As we have already mentioned in this chapter, two ground types are consistent if and only if they are the same type. Similar to the dynamic type (dyn) in gradual typing [53, 50], a type variable can be consistent with any type. The type consistency relation is reflective and symmetric, but not transitive. For example, there is no conflict in the constraint $\{\alpha \sim \mathbf{int}, \alpha \sim \mathbf{bool}\}$.

An object-level lambda abstraction has two syntactic forms. If the type of the parameter is a simple type, then the syntax is the same as the lambda abstraction in the surface language. If the type of the parameter is some meta expression t^m which is not a simple type, then we attach a fresh type variable to the meta expression, as we see from the syntax $\lambda x : t^m = \alpha.\ t$. During type-checking, variable α is used as the type of parameter x.

In the meta language, syntax **if** t^m **then** $t^m : \sigma^m$ **else** $t^m : \sigma^m$ **withtype** σ^m represents a kernel-level if expression. We explicitly specific the type of an if expression and also the types of its two branches. Syntax $C \Rightarrow t^m$ represents a meta expression which has a type constraint as a precondition. Similarly, syntax $C \Rightarrow t$ represents an object expression which has a type constraint as a precondition. A type variable that appears in t^m is constrained by C if the type variable appears in C. During meta-evaluation, we need to check that the constraint generated from t^m is consistent with constraint C. For example, consider the following metaprogram:

Figure 3.9: The core part of the surface language syntax. The language supports type-reflective metaprogramming and is used for the implementation of incremental type-checking.

---

x	meta/object level variable
α, β	object type variables
c	value constants
$\mathcal{T}$	type primitives (e.g. **dom**, **cod**, $=_\tau$, $\gamma^?$, $\rightarrow^?$, **typeof**)
γ	type constants (e.g. **int**, **bool**)

simple types

$\tau^s \quad ::= \quad \gamma \mid \tau^s \rightarrow \tau^s$

object types

$\tau \quad ::= \quad \gamma \mid \alpha \mid \tau \rightarrow \tau$

object language

$e \quad ::= \quad c \mid x \mid \lambda x : e^m.\ e \mid e\ e \mid \sim e^m \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid$
$\textbf{let meta}\ x = e^m\ \textbf{in}\ e$

meta types

$\tau^m \quad ::= \quad \gamma \mid \exists\overline{\alpha}.(\textbf{code}\ \tau) \mid \textbf{type} \mid \tau^m \rightarrow \tau^m$

meta language

$e^m \quad ::= \quad c \mid \mathcal{T} \mid \gamma \mid e^m \rightarrow e^m \mid$
$x \mid \lambda x : \tau^m.\ e^m \mid e^m\ e^m \mid \prec e \succ \mid \%e^m \mid$
$\textbf{let}\ x = e^m\ \textbf{in}\ e^m \mid \textbf{if}\ e^m\ \textbf{then}\ e^m\ \textbf{else}\ e^m \mid$
$\textbf{if}\ e^m\ \textbf{then}\ e^m\ \textbf{else}\ e^m\ \textbf{withtype}\ \tau^m$

Figure 3.10: The core part of the kernel language syntax. The language supports type-reflective metaprogramming and is used for the implementation of incremental type-checking.

---

x		meta/object level variable
α, β		type variables
c		value constants
$\mathcal{T}$		type primitives (e.g. **dom**, **cod**, $=_\tau$, $\gamma^?$, $\rightarrow^?$, **typeof**)
γ		type constants (e.g. **int**, **bool**)
type relations		
ω	$::=$	$\sigma = \sigma \mid \sigma \sim \sigma \mid \sigma^m = \sigma^m \mid \sigma^m \sim \sigma^m$
type constraint		
C	$::=$	$\emptyset \mid C \cup \{\omega\}$
simple types		
σ^s, ρ^s	$::=$	$\gamma \mid \sigma^s \rightarrow \sigma^s$
object types		
σ, ρ	$::=$	$\gamma \mid \alpha \mid \sigma \rightarrow \sigma$
object language		
t	$::=$	$c \mid x \mid \lambda x : \sigma^s.\ t \mid t\ t \mid \sim t^m \mid$
		$\lambda x : t^m = \alpha.\ t \mid C \Rightarrow t$
proper code		
t^o	$::=$	$c \mid x \mid \lambda x : \sigma^s.\ t^o \mid t^o\ t^o \mid C \Rightarrow t^o$
meta types		
σ^m, ρ^m	$::=$	$\gamma \mid$ **code** $\sigma \mid$ **type** $\mid \sigma^m \rightarrow \sigma^m$
meta language		
t^m	$::=$	$c \mid \mathcal{T} \mid \gamma \mid t^m \rightarrow t^m \mid$
		$x \mid \lambda x : \sigma^m.\ t^m \mid t^m\ t^m \mid \prec t \succ \mid \%t^m \mid$
		if t^m **then** $t^m : \sigma^m$ **else** $t^m : \sigma^m$ **withtype** σ^m
		$C \Rightarrow t^m$
meta value		
v^m	$::=$	$c \mid \mathcal{T} \mid \gamma \mid v^m \rightarrow v^m \mid \lambda x : \sigma^m.\ t^m \mid \prec t^o \succ \mid C \Rightarrow v^m$

```
1  {α = int} ⇒
2      ((λ x : bool.
3            ≺ λy : if x then int else string = α. y ≻)
4          false)
```

the meta function application (line 2-4) is guarded by the precondition $\{\alpha = \textbf{int}\}$. During meta-evaluation, the type of parameter y is evaluated into **string**, which generates the constraint $\{\alpha = \textbf{string}\}$. So, there is a conflict between the precondition of α and the actual value of α.

3.3.3 Language Translation and Type-checking

Before explaining the translation rules, we first present and explain some of the definitions and functions that the rules use. We have already introduced notation $[\sigma/\alpha]$, which means the substitution of type variable α for object type σ. A sequence of substitutions $[\sigma_1/\alpha_1, ..., \sigma_n/\alpha_n]$, written as θ, means the simultaneous substitution of type variables α_1, ..., α_{n-1}, and α_n for types σ_1, ..., σ_{n-1}, and σ_n.

Generally, a unification algorithm solves a set of equations and generates a sequence of substitutions if the equations are solvable. In his book [44] (Chapter 22.4), Benjamin Pierce presented a unification algorithm which solves a set of type equality constraints generated by type inference. The following is the unification algorithm in Pierce's book for solving a set of type equality constraints:

unify(C) =
  if $C = \emptyset$, **then** []
  else let $C \cup \{\sigma = \rho\} = C'$ **in**
    if $\sigma = \rho$ **then** *unify*(C')
    else if $\sigma = \alpha$ **and** $\alpha \notin$ *vars*(ρ) **then** *unify*$([\rho/\alpha]C') \circ [\rho/\alpha]$
    else if $\rho = \alpha$ **and** $\alpha \notin$ *vars*(σ) **then** *unify*$([\sigma/\alpha]C') \circ [\sigma/\alpha]$
    else if $\sigma = \sigma_1 \to \sigma_2$ **and** $\rho = \rho_1 \to \rho_2$ **then** *unify*$(C' \cup \{\sigma_1 = \rho_1, \sigma_2 = \rho_2\})$
    else *fail*

In the algorithm, notation $\theta_1 \circ \theta_2$ means the composition of two sequences of substitutions. Suppose

$\theta_1 = [\tau_1/\alpha_1, ..., \tau_n/\alpha_n]$ and $\theta_2 = [\rho_1/\beta_1, ..., \rho_n/\beta_n]$, then $\theta_1 \circ \theta_2$ is defined as:

$[\tau_1/\alpha_1, ..., \tau_n/\alpha_n, \theta_1(\rho_1)/\beta_1, ..., \theta_1(\rho_n)/\beta_n]$

In Figure 3.11, we extend the Pierce's unification algorithm with the ability to solve type consistency constraints. Function $\mathbb{U}$ is the main function and it calls two sub-functions: *unify_eq* and *unify_co*. Function *unify_eq* solves the type equality constraints in a set of constraints C and returns a set of simplified constraints where type equalities are in the form of $\alpha = \sigma$. Function *unify_co* receives the constraints returned by *unify_eq* and solves the type consistencies in the constraints. To prove that function *unify_eq* is equivalent to Pierce's unification algorithm *unify*, we first give the following definition: given a substitution θ, if $C = \{\alpha = \sigma \mid \text{for each } \sigma/\alpha \in \theta\}$, then we say that θ is convertible to the set of type equality constraints C, written $\theta \hookrightarrow C$. The following property shows that *unify_eq* is equivalent to *unify*.

Theorem 4 (Correctness of Unification)**.** *For any constraint set C and substitution θ, if C has no type consistency constraints and $\theta \hookrightarrow C'$, then we have unify$(C) \circ \theta \hookrightarrow$ unify_eq(C, C').*

Proof. Prove by mathematical induction. For the detailed proof, see the appendix. □

Function *unify_co*, when meeting a type consistency constraint, such as $\alpha \sim \sigma$, does not substitute α for σ in the rest of the constraints because type consistency relation is not transitive. So, constraint set $\{\alpha \sim \textbf{int}, \alpha \sim \beta, \beta = \textbf{bool}\}$ is solvable and the solution is $\{\alpha \sim \textbf{int}, \alpha \sim \textbf{bool}, \beta = \textbf{bool}\}$. In contrast, constraint set $\{\alpha \sim \textbf{int}, \alpha = \beta, \beta = \textbf{bool}\}$ is not solvable, because α is equal to β and β is equal to **bool**, thus α is equal to **bool** (equality is transitive), therefore **bool** $\sim$ **int**. But **bool** is not consistent with **int**.

During type-checking, we need to compute an upper bound of two types. Figure 3.12 presents two functions. Function $\mathcal{UB}$ computes an upper bound of two object types, while function $\mathcal{UB}^m$ computes an upper bound of two meta types. Please note that the functions are not guaranteed to return the least upper bound of two types. For example, $\mathcal{UB}(\textbf{int} \to \textbf{int}, \textbf{bool} \to \textbf{bool})$ is evaluated to $\alpha \to \beta$ (suppose α and β are fresh type variables), but the least upper bound for **int** $\to$ **int** and **bool** $\to$ **bool** is $\alpha \to \alpha$.

Figure 3.11: The function $\mathbb{U}$, which solves a set of type equality and type consistency constraints, and generates a set of simplified constraints if the constraints are solvable. A simplified constraint is either $\alpha = \sigma$ or $\alpha \sim \sigma$.

Unification for type constraints:

$\mathbb{U}(C) =$
  let $C_r = \mathit{unify\_eq}(C, \{\})$ **in**
    $\mathit{unify\_co}(C_r)$

$\mathit{unify\_eq}(C, C_r) =$
  if $C = \{\}$ **then** C_r
  else let $C_0 \cup \{\omega\} = C$ **in**
    match ω **with**
      $\sigma = \rho \rightarrow$
        if $\sigma = \rho$ **then** $\mathit{unify\_eq}(C_0, C_r)$
        else if $\sigma = \sigma_1 \rightarrow \sigma_2$ **and** $\rho = \rho_1 \rightarrow \rho_2$ **then** $\mathit{unify\_eq}(C_0 \cup \{\sigma_1 = \rho_1,\ \sigma_2 = \rho_2\}, C_r)$
        else if $\sigma = \alpha$ **and** $\alpha \notin \mathit{vars}(\rho)$ **then** $\mathit{unify\_eq}([\rho/\alpha]C_0, [\rho/\alpha]C_r \cup \{\alpha = \rho\})$
        else if $\rho = \alpha$ **and** $\alpha \notin \mathit{vars}(\sigma)$ **then** $\mathit{unify\_eq}([\sigma/\alpha]C_0, [\sigma/\alpha]C_r \cup \{\alpha = \sigma\})$
        else *fail*
    | $\sigma^m = \rho^m \rightarrow$
        if $\sigma^m = \rho^m$ **then** $\mathit{unify\_eq}(C_0, C_r)$
        else if $\sigma^m = \sigma_1^m \rightarrow \sigma_2^m$ **and** $\rho^m = \rho_1^m \rightarrow \rho_2^m$ **then** $\mathit{unify\_eq}(C_0 \cup \{\sigma_1^m = \rho_1^m,\ \sigma_2^m = \rho_2^m\}, C_r)$
        else if $\sigma^m =$ **code** σ **and** $\rho^m =$ **code** ρ **then** $\mathit{unify\_eq}(C_0 \cup \{\sigma = \rho\}, C_r)$
        else *fail*
    | $\_ \rightarrow \mathit{unify\_eq}(C_0, C_r \cup \{\omega\})$

$\mathit{unify\_co}(C) =$
  if $C = \{\}$ **then** $\{\}$
  else let $C_0 \cup \{\omega\} = C$ **in**
    match ω **with**
      $\sigma \sim \rho \rightarrow$
        if $\sigma = \rho$ **then** $\mathit{unify\_co}(C_0)$
        else if $\sigma = \sigma_1 \rightarrow \sigma_2$ **and** $\rho = \rho_1 \rightarrow \rho_2$ **then** $\mathit{unify\_co}(C_0 \cup \{\sigma_1 \sim \rho_1,\ \sigma_2 \sim \rho_2\})$
        else if $\sigma = \alpha$ **and** $\alpha \notin \mathit{vars}(\rho)$ **then** $\mathit{unify\_co}(C_0) \cup \{\alpha \sim \rho\}$
        else if $\rho = \alpha$ **and** $\alpha \notin \mathit{vars}(\sigma)$ **then** $\mathit{unify\_co}(C_0) \cup \{\alpha \sim \sigma\}$
        else *fail*
    | $\sigma^m \sim \rho^m \rightarrow$
        if $\sigma^m = \rho^m$ **then** $\mathit{unify\_co}(C_0)$
        else if $\sigma^m = \sigma_1^m \rightarrow \sigma_2^m$ **and** $\rho^m = \rho_1^m \rightarrow \rho_2^m$ **then** $\mathit{unify\_co}(C_0 \cup \{\sigma_1^m \sim \rho_1^m, \sigma_2^m \sim \rho_2^m\})$
        else if $\sigma^m =$ **code** σ **and** $\rho^m =$ **code** ρ **then** $\mathit{unify\_co}(C_0 \cup \{\sigma \sim \rho\})$
        else *fail*
    | $\_ \rightarrow \mathit{unify\_co}(C_0) \cup \{\omega\}$

During type-check, we need to check the naive subtype relation of two object types and two meta types. For example, to check if type $\alpha \to \alpha$ is a naive subtype of $\beta_1 \to \beta_2$, we need to find out if there exists some substitution for β_1 and β_2 that makes type $\beta_1 \to \beta_2$ equal to $\alpha \to \alpha$. Using the unification algorithm, we find out that such substitution exists: $[\beta_1/\alpha, \beta_2/\alpha]$. In turn, to check if $\beta_1 \to \beta_2$ is a naive subtype of $\alpha \to \alpha$, we also need to find out if there exists some substitution for α that makes $\alpha \to \alpha$ equal to $\beta_1 \to \beta_2$, but first we must presume the most general condition for β_1 and β_2, that is $\beta_1 \neq \beta_2$. Using the unification algorithm, we find out that there exists a substitution for α if and only if $\beta_1 = \beta_2$, which violates our presumption. In general, to check if type T_1 is a naive subtype of type T_2, we first replace each type variable in T_1 with a unique ground type that does not appear in T_1 and T_2; then we unify T_1 and T_2. If there is a solution, then T_1 is a naive subtype of T_2. For instance, to check if $(\alpha \to \mathbf{int}) \to \mathbf{int} \to \alpha$ is a naive subtype of $\beta \to \beta$, we first substitute α for some ground type that is different from **int**, say **bool**, then we unify $(\mathbf{bool} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{bool}$ and $\beta \to \beta$, which has no solution. Therefore, $(\alpha \to \mathbf{int}) \to \mathbf{int} \to \alpha$ is not a naive subtype of $\beta \to \beta$. We define the notion $\sigma_1 <:_n \sigma_2$, which means object type σ_1 is a naive subtype of object type σ_2; and notion $\sigma_1^m <:_n \sigma_2^m$, which means meta type σ_1^m is a naive subtype of meta type σ_2^m.

Figure 3.13, 3.14, and 3.15 show the rules which translate the surface language to the kernel language, type-check the kernel language, and meanwhile generate type constraints.

In Figure 3.13, Rule T6.1 translates a lambda abstraction whose parameter type is a non-value meta expression. In this case, we use a fresh type variable to represent the type of the parameter and also attach the type variable to the meta expression.

As we previously mentioned, the type of an object expression can be the result of meta-evaluation. Consider the following metaprogram at the surface level,

```
1 let meta t =
2     λx : bool. ≺ λy : if x then string else int. y ≻      // bool → code (α → α)
3 in
4     ≺ (∼ (t true) "abc", ∼ (t false) 3) ≻
```

Figure 3.12: Two auxiliary functions: function $\mathcal{UB}$ computes an upper bound of two object types, and function $\mathcal{UB}^m$ computes an upper bound of two meta types. Assume function *gen_fresh_tvar* is available, which always generates a fresh type variable.

Object type upper bound computation:

$\mathcal{UB}(\sigma, \rho) =$
  if $\sigma = \rho$
    then σ
  else if $\sigma = \sigma_1 \to \sigma_2$ **and** $\rho = \rho_1 \to \rho_2$
    then
      let $\sigma_3 = \mathcal{UB}(\sigma_1, \rho_1)$ **in**
        let $\rho_3 = \mathcal{UB}(\sigma_2, \rho_2)$ **in**
          $\sigma_3 \to \rho_3$
  else
    gen_fresh_tvar()

Meta type upper bound computation:

$\mathcal{UB}^m(\sigma^m, \rho^m) =$
  if $\sigma^m = \rho^m$
    then σ^m
  else if $\sigma^m =$ **code** σ_0 **and** $\rho^m =$ **code** ρ_0
    then
      let $\sigma = \mathcal{UB}(\sigma_0, \rho_0)$ **in**
        code σ
  else if $\sigma^m = \sigma_1^m \to \sigma_2^m$ **and** $\rho^m = \rho_1^m \to \rho_2^m$
    then
      let $\sigma_3^m = \mathcal{UB}^m(\sigma_1^m, \rho_1^m)$ **in**
        let $\rho_3^m = \mathcal{UB}^m(\sigma_2^m, \rho_2^m)$ **in**
          $\sigma_3^m \to \rho_3^m$
  else
    fail

In line 2, the meta-level function returns an object-level function. In the object-level function, the actual type of variable y is the evaluation result of expression **if** x **then string else int**. In line 4, when the meta function is applied to **true**, the type of y is **string**, thus it can accept "abc". When the meta function is applied to **false**, the type of y is **int**, thus it can accept integer 3. So, the above program is well-typed. During language translation and type-checking, the type of y is represented by some fresh type variable, say α. For the code in line 4, if we use a normal constraint-based type-checker, we would get constraint $\{\alpha = \text{string};\ \alpha = \text{int}\}$, which is unsolvable. To support the"type polymorphism" of an object expression, we substitute the type variables in the type of the expression for fresh type variables (see Rule T8 in Figure 3.13). In line 4, suppose α is substituted for α_1 and α_2 respectively, then we have constraint $\{\alpha_1 = \text{string};\ \alpha_2 = \text{int};\ \alpha \sim \alpha_1;\ \alpha \sim \alpha_2\}$, which has no conflict.

In some cases, although the type of an object expression is the result of meta-evaluation, the actual type of the expression is restricted by some constraint. We change the above program, making the type of variable y constrained at definition time:

```
1  let meta t =
2      λx : bool. ≺ λy : if x then string else int. y + 1 ≻
3  in
4      ≺ (∼ (t true) "abc", ∼ (t false) 3) ≻
```

The modified program is not well-typed. Because we have the extra constraint $\{\alpha = \textbf{int}\}$ generated from the code in line 2, the entire constraint becomes $\{\alpha_1 = \textbf{string};\ \alpha_2 = \textbf{int};\ \alpha \sim \alpha_1;\ \alpha \sim \alpha_2\} \cup \{\alpha = \textbf{int}\}$, which has the type consistency conflict: $\textbf{string} \sim \textbf{int}$.

In Figure 3.15, Rule T26 translates and type-checks a meta-level function application. In this rule, we add the constraint that the type of the parameter should be consistent with the type of the argument. Rule T30 translates and type-checks an if expression whose type is not specified. In this case, we use function $\mathcal{UB}^m$ to calculate an upper bound for the types of two branches. For example, consider the following surface-level metaprogram (please note that a comment begins with //).

```
1  let serialize =      // fun_double : string → (code (α → string)) option
```

```
2     λs : string.
3       if s == "INT" then Some ≺ λx : int. (string_of_int x) ≻          // int → string
4       else if s == "STR" then Some ≺ λx : string. x ≻                  // string → string
5       else if s == "BOOL" then Some ≺ λx : int. (string_of_bool x) ≻   // bool → string
6       else None
7   in
8     match (serialize "INT") with Some f → Some ≺ ((∼ f) true) ≻
9       | None → None
```

Assume the type of function serialize is inferred as **string** $\rightarrow$ (**code** ($\alpha \rightarrow$ **string**)) **option**. By Rule T8, we assume the type and the constraint of code $\sim$ f in line 8 are $\alpha_1 \rightarrow$ **string** and $\{\alpha \rightarrow \textbf{string} \sim \alpha_1 \rightarrow \textbf{string}\}$ (α_1 is fresh). By Rule T7, we assume the type and the constraint of code ((∼ f) **true**) in line 8 are α_2 (α_2 is fresh) and $\{\alpha_1 \rightarrow \textbf{string} = \textbf{bool} \rightarrow \alpha_2\}$. So, statically the above program is well-typed, although it has a type error during meta-evaluation. Rule T31 translates and type-checks an if expression with the expression's type attached. In this case, we need to check that the type is an upper bound of the types of the two branches, and also check that the type should be the naive subtype of the upper bound computed by function $\mathcal{UB}^m$. According to Rule T31, meta expression λx : **bool**. **if** x **then** ≺ λx : **int**. x ≻ **else** ≺ λx : **int**. x ∗ x ≻ **withtype code** ($\alpha \rightarrow \alpha$) is rejected by our type system: by function $\mathcal{UB}^m$, we obtain an upper bound of **code** (**int** $\rightarrow$ **int**), but user-specified upper bound **code** ($\alpha \rightarrow \alpha$) is not a naive subtype of **code** (**int** $\rightarrow$ **int**). Such user-specified upper bound checking can prevent some metaprograms that always generate ill-typed code. Consider the following program as an example:

```
1   let f =
2     λx : bool. if x then ≺ λx : int. x ≻ else ≺ λx : int. x ∗ x ≻ withtype code (α → α)
3   in
4     λy : bool. ≺ ∼ (f y) "abc" ≻
```

Because the if expression cannot generate a function that accepts a string, the function application ∼ (f y) "abc" (in line 4) is never well-typed.

Last but not least, we need two rules at the top level, which check if a surface-level expression is well-typed or not (in the rules, function $\mathbb{C}$ collects and combines all the constraints that appear in an expression):

$$\text{(TL1)}\frac{\emptyset;\emptyset \vdash e \rightsquigarrow t : \sigma \quad \mathbb{C}(t) = C \quad \mathbb{U}(C) = C'}{e\ \mathbf{wt}} \qquad \text{(TL2)}\frac{\emptyset;\emptyset \vdash e^m \rightsquigarrow t^m : \sigma^m \quad \mathbb{C}(t^m) = C \quad \mathbb{U}(C) = C'}{e^m\ \mathbf{wt}}$$

3.3.4 Meta-evaluation

During meta-evaluation, we still need to perform type-checking, to catch potential type errors caused by the types and the typing information generated during meta-evaluation. In the kernel language, there are three kinds of expressions that may generate typing information and type constraints during meta-evaluation.

The first is if expression: **if** t_1^m **then** $t_2^m : \sigma_2^m$ **else** $t_3^m : \sigma_3^m$ **withtype** σ^m. During meta-evaluation, it generates the constraint $\{\sigma_2^m = \sigma^m\}$ or $\{\sigma_3^m = \sigma^m\}$, depending on the evaluation result of t_1^m. Consider the following metaprogram:

```
1  {β → β = string → β0} ⇒
2      (~ ({α → α ~ β → β} ⇒
3          ((λ x: bool.
4              if x then ≺ λx : int. x ≻: code (int → int)
5              else ≺ λx : string. x ≻: code (string → string)
6                  withtype code (α → α))
7            true))
8       "abc")
```

During type-checking, we infer the constraints $\{\beta \to \beta = \textbf{string} \to \beta_0;\ \alpha \to \alpha \sim \beta \to \beta\}$. During meta-evaluation, after the condition of the if expression (at line 4) is evaluated into **true**, we have the generated constraint { **code** (**int** → **int**) = **code** ($\alpha \to \alpha$) }. By unifying this constraint with the constraints statically inferred, we get the type inconsistency error between type **int** and type **string**.

Figure 3.13: Translation to the kernel language, part one: the rules of translating the surface object language to the kernel object language, including type-checking the kernel object language and generating type constraints.

Object type translation $\boxed{\Gamma; \Psi \vdash \tau \rightsquigarrow \sigma}$

$$\text{(T1)}\frac{}{\Gamma; \Psi \vdash \gamma \rightsquigarrow \gamma} \qquad \text{(T2)}\frac{\alpha \in \Gamma}{\Gamma; \Psi \vdash \alpha \rightsquigarrow \alpha} \qquad \text{(T3)}\frac{\Gamma; \Psi \vdash \tau_1 \rightsquigarrow \sigma_1 \quad \Gamma; \Psi \vdash \tau_2 \rightsquigarrow \sigma_2}{\Gamma; \Psi \vdash \tau_1 \to \tau_2 \rightsquigarrow \sigma_1 \to \sigma_2}$$

Object expression translation $\boxed{\Gamma; \Psi \vdash e \rightsquigarrow t : \sigma}$

$$\text{(T4)}\frac{\mathit{type}(c) = \gamma}{\Gamma; \Psi \vdash c \rightsquigarrow c : \gamma} \qquad \text{(T5)}\frac{x : \sigma \in \Gamma}{\Gamma; \Psi \vdash x \rightsquigarrow x : \sigma}$$

$$\text{(T6.1)}\frac{\nexists \tau^s.\ e^m = \tau^s \quad \Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \mathbf{type} \quad \alpha \text{ is fresh} \quad \Gamma, x : \alpha; \Psi \vdash e \rightsquigarrow t : \sigma}{\Gamma; \Psi \vdash \lambda x : e^m.\ e \rightsquigarrow \lambda x : t^m = \alpha.\ t : \alpha \to \sigma}$$

$$\text{(T6.2)}\frac{\Gamma; \Psi \vdash \tau^s \rightsquigarrow \sigma^s \quad \Gamma, x : \sigma^s; \Psi \vdash e \rightsquigarrow t : \sigma}{\Gamma; \Psi \vdash \lambda x : \tau^s.\ e \rightsquigarrow \lambda x : \sigma^s.\ t : \sigma^s \to \sigma}$$

$$\text{(T7)}\frac{\Gamma; \Psi \vdash e_1 \rightsquigarrow t_1 : \sigma_1 \quad \Gamma; \Psi \vdash e_1 \rightsquigarrow t_2 : \sigma_2 \quad C = \{\sigma_1 = \sigma_2 \to \alpha\} \quad \alpha \text{ is fresh}}{\Gamma; \Psi \vdash e_1\ e_2 \rightsquigarrow C \Rightarrow (t_1\ t_2) : \alpha}$$

$$\text{(T8)}\frac{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \mathbf{code}\ \sigma \quad \overline{\alpha} = \mathit{vars}(\sigma) \quad \overline{\beta} \text{ are fresh} \quad |\overline{\alpha}| = |\overline{\beta}| \quad \rho = [\overline{\beta}/\overline{\alpha}]\sigma}{\Gamma; \Psi \vdash \sim e^m \rightsquigarrow \ \{\sigma \sim \rho\} \Rightarrow (\sim t^m) : \rho}$$

$$\text{(T9)}\frac{\Gamma; \Psi \vdash e_1 \rightsquigarrow t_1 : \sigma_1 \quad \Gamma, x : \sigma_1; \Psi \vdash e_1 \rightsquigarrow t_2 : \sigma_2}{\Gamma; \Psi \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow (\lambda x : \sigma_1.\ e_2)\ e_1 : \sigma_2}$$

$$\text{(T10)}\frac{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \sigma^m \quad \Gamma; \Psi, x : \sigma^m \vdash e \rightsquigarrow t : \sigma}{\Gamma; \Psi \vdash \mathbf{let}\ \mathbf{meta}\ x = e^m\ \mathbf{in}\ e \rightsquigarrow \sim ((\lambda x : \sigma^m.\ \prec t \succ)\ t^m) : \sigma}$$

Figure 3.14: Translation to the kernel language, part two: this figure shows the rules of translating the surface meta types to the kernel meta types, including the checking of well-formed meta types.

Meta type translation $\boxed{\Gamma; \Psi \vdash \tau^m \rightsquigarrow \sigma^m}$

$$\text{(T11)}\frac{}{\Gamma; \Psi \vdash \gamma \rightsquigarrow \gamma} \qquad \text{(T12)}\frac{\overline{\beta} \text{ are fresh} \quad |\overline{\alpha}| = |\overline{\beta}| \quad \Gamma, \overline{\beta}; \Psi \vdash [\overline{\beta}/\overline{\alpha}]\tau \rightsquigarrow \sigma}{\Gamma; \Psi \vdash \exists \overline{\alpha}.(\mathbf{code}\ \tau) \rightsquigarrow \mathbf{code}\ \sigma}$$

$$\text{(T13)}\frac{}{\Gamma; \Psi \vdash \mathbf{type} \rightsquigarrow \mathbf{type}} \qquad \text{(T14)}\frac{\Gamma; \Psi \vdash \tau_1^m \rightsquigarrow \sigma_1^m \quad \Gamma; \Psi \vdash \tau_2^m \rightsquigarrow \sigma_2^m}{\Gamma; \Psi \vdash \tau_1^m \to \tau_2^m \rightsquigarrow \sigma_1^m \to \sigma_2^m}$$

Figure 3.15: Translation to the kernel language, part three: the rules of translating the surface meta expressions to the kernel meta expressions, including the type-checking of the kernel meta expressions and generating type constraints.

Meta expression translation $\boxed{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \sigma^m}$

$$(\text{T15})\frac{}{\Gamma; \Psi \vdash \gamma \rightsquigarrow \gamma : \mathbf{type}} \qquad (\text{T16})\frac{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \mathbf{code}\ \sigma}{\Gamma; \Psi \vdash \mathbf{typeof}\ e^m \rightsquigarrow \mathbf{typeof}\ t^m : \mathbf{type}}$$

$$(\text{T17})\frac{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \mathbf{type}}{\Gamma; \Psi \vdash \mathbf{dom}\ e^m \rightsquigarrow \mathbf{dom}\ t^m : \mathbf{type}} \qquad (\text{T18})\frac{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \mathbf{type}}{\Gamma; \Psi \vdash \mathbf{cod}\ e^m \rightsquigarrow \mathbf{cod}\ t^m : \mathbf{type}}$$

$$(\text{T19})\frac{\Gamma; \Psi \vdash e_1^m \rightsquigarrow t_1^m : \mathbf{type} \qquad \Gamma; \Psi \vdash e_2^m \rightsquigarrow t_2^m : \mathbf{type}}{\Gamma; \Psi \vdash e_1^m =^{\tau} e_2^m \rightsquigarrow t_1^m =^{\tau} t_2^m : \mathbf{bool}}$$

$$(\text{T20})\frac{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \mathbf{type}}{\Gamma; \Psi \vdash \gamma^?\ e^m \rightsquigarrow \gamma^?\ t^m : \mathbf{bool}} \qquad (\text{T21})\frac{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \mathbf{type}}{\Gamma; \Psi \vdash \rightarrow^?\ e^m \rightsquigarrow \rightarrow^?\ t^m : \mathbf{bool}}$$

$$(\text{T22})\frac{\Gamma; \Psi \vdash e_1^m \rightsquigarrow t_1^m : \mathbf{type} \qquad \Gamma; \Psi \vdash e_2^m \rightsquigarrow t_2^m : \mathbf{type}}{\Gamma; \Psi \vdash e_1^m \rightarrow e_2^m \rightsquigarrow t_1^m \rightarrow t_2^m : \mathbf{type}}$$

$$(\text{T23})\frac{x : \sigma^m \in \Psi}{\Gamma; \Psi \vdash x \rightsquigarrow x : \sigma^m} \qquad (\text{T24})\frac{\mathit{type}(c) = \gamma}{\Gamma; \Psi \vdash c \rightsquigarrow c : \gamma}$$

$$(\text{T25})\frac{\Gamma; \Psi \vdash \tau^m \rightsquigarrow \sigma^m \qquad \Gamma; \Psi, x : \sigma^m \vdash e^m \rightsquigarrow t^m : \rho^m}{\Gamma; \Psi \vdash \lambda x : \tau^m.\ e^m \rightsquigarrow \lambda x : \sigma^m.\ t^m : \sigma^m \rightarrow \rho^m}$$

$$(\text{T26})\frac{\Gamma; \Psi \vdash e_1^m \rightsquigarrow t_1^m : \sigma_{11}^m \rightarrow \sigma_{12}^m \qquad \Gamma; \Psi \vdash e_2^m \rightsquigarrow t_2^m : \sigma_2^m}{\Gamma; \Psi \vdash e_1^m\ e_2^m \rightsquigarrow \{\sigma_{11}^m \sim \sigma_2^m\} \Rightarrow (t_1^m\ t_2^m) : \sigma_{12}^m}$$

$$(\text{T27})\frac{\Gamma; \Psi \vdash e \rightsquigarrow t : \sigma}{\Gamma; \Psi \vdash \prec e \succ \rightsquigarrow \prec t \succ : \mathbf{code}\ \sigma} \qquad (\text{T28})\frac{\Gamma; \Psi \vdash e^m \rightsquigarrow t^m : \gamma}{\Gamma; \Psi \vdash \%e^m \rightsquigarrow \%t^m : \mathbf{code}\ \gamma}$$

$$(\text{T29})\frac{\Gamma; \Psi \vdash e_1^m \rightsquigarrow t_1^m : \sigma_1^m \qquad \Gamma; \Psi, x : \sigma_1^m \vdash e_2^m \rightsquigarrow t_2^m : \sigma_2^m}{\Gamma; \Psi \vdash \mathbf{let}\ x = e_1^m\ \mathbf{in}\ e_2^m \rightsquigarrow (\lambda x : \sigma_1^m.\ e_2^m)\ e_1^m : \sigma_2^m}$$

$$(\text{T30})\frac{\begin{array}{c}\Gamma; \Psi \vdash e_1^m \rightsquigarrow t_1^m : \mathbf{bool} \qquad \Gamma; \Psi \vdash e_2^m \rightsquigarrow t_2^m : \sigma_2^m \\ \Gamma; \Psi \vdash e_3^m \rightsquigarrow t_3^m : \sigma_3^m \qquad \mathcal{UB}^m(\sigma_2^m, \sigma_3^m) = \sigma^m\end{array}}{\begin{array}{l}\Gamma; \Psi \vdash \mathbf{if}\ e_1^m\ \mathbf{then}\ e_2^m\ \mathbf{else}\ e_3^m \rightsquigarrow \\ \qquad \mathbf{if}\ e_1^m\ \mathbf{then}\ e_2^m : \sigma_2^m\ \mathbf{else}\ e_3^m : \sigma_3^m\ \mathbf{withtype}\ \sigma^m : \sigma^m\end{array}}$$

$$(\text{T31})\frac{\begin{array}{ccccc}\Gamma; \Psi \vdash e_1^m \rightsquigarrow t_1^m : \mathbf{bool} & & \Gamma; \Psi \vdash e_2^m \rightsquigarrow t_2^m : \sigma_2^m & & \Gamma; \Psi \vdash e_3^m \rightsquigarrow t_3^m : \sigma_3^m \\ \Gamma; \Psi \vdash \tau^m \rightsquigarrow \sigma^m & \sigma_2^m <:_n \sigma^m & \sigma_3^m <:_n \sigma^m & \mathcal{UB}^m(\sigma_2^m, \sigma_3^m) = \rho^m & \sigma^m <:_n \rho^m\end{array}}{\begin{array}{l}\Gamma; \Psi \vdash \mathbf{if}\ e_1^m\ \mathbf{then}\ e_2^m\ \mathbf{else}\ e_3^m\ \mathbf{withtype}\ \tau^m \rightsquigarrow \\ \qquad \mathbf{if}\ e_1^m\ \mathbf{then}\ e_2^m : \sigma_2^m\ \mathbf{else}\ e_3^m : \sigma_3^m\ \mathbf{withtype}\ \sigma^m : \sigma^m\end{array}}$$

The second kind of expression is lambda abstraction: $\lambda x : t^m = \alpha.\ t$, whose parameter type is not a value. During meta-evaluation, it generates constraint $\{\alpha = \sigma^s\}$ when type expression t^m is evaluated into some simple type σ^s. For program

λy : **if false then int else string** = α. $\{\alpha = \textbf{int}\} \Rightarrow$ (y + y)

when the parameter of y is evaluated into **string**, it generates the constraint $\{\alpha = \textbf{string}\}$, which has conflict with the statically inferred constraint $\{\alpha = \textbf{int}\}$. Here is another example: for program

```
1  {code (int → int) ∼ code (α → α)} ⇒
2      ((λ x : code (int → int). x)
3          ≺ λy : if true then string else int = α. y ≻)
```

Statically, α is consistent with **int**. During meta-evaluation, when the type of parameter y is evaluated into **string**, we have the constraint $\{\textbf{string} = \alpha\}$. So, there is a conflict.

The third kind of expression is meta-level function application: $t_1^m\ t_2^m$. After t_1^m is evaluated into function $\lambda x : \sigma^m.t^m$, and t_2^m is evaluated into value v_2^m, we need to check that the constraints in t^m have no conflicts with the constraints in t_2^m before applying the substitution of x for t_2^m in t^m. Consider the following kernel metaprogram (please note that a line beginning with // is a comment.):

```
1  {code (α → int) ∼ code (β → β)} ⇒
2      // − − − − − begin fun app
3      // − − − − − begin fun1
4      ((λ f : code (β → β).
5          ≺ {β1 → β1 = string → β2} ⇒
6              (({β → β ∼ β1 → β1} ⇒ (∼ f)) "abc") ≻)
7      // − − − − − end fun1
8          // − − − − − begin fun2
9          ≺ λy : if true then int else string = α.
10             {α = int} ⇒ (y + y) ≻)
11         // − − − − − end fun2
12     // − − − − − end fun app
```

During meta-evaluation, when the type of parameter y is evaluated into **int**, the extra constraint $\{\alpha = \textbf{int}\}$ is generated and fun2 is evaluated into

```
1  {α = int} ⇒
2      ≺ λy : int.
3              {α = int} ⇒ (y + y) ≻
```

Next, we use unification to check if the substitution of f for fun2 will cause type conflicts. The constraints to be unified include the constraints in fun1, the constraints in fun2, and constraint $\{\textbf{code}\ (\alpha \to \textbf{int}) = \textbf{code}\ (\beta \to \beta)\}$, which is the constraint generated from the top-level constraint $\{\textbf{code}\ (\alpha \to \textbf{int}) \sim \textbf{code}\ (\beta \to \beta)\}$ (in line 1), by converting the type consistency relation into the type equality relation. During type-checking, we require the type of a parameter to be consistent with its argument. The reason is that a function can be applied to arguments of different types. To support such "polymorphism", we use type consistency. During meta-evaluation, the reduction of a function application is the substitution of a parameter for its argument. To make sure the result of the substitution is well-typed, we require the type of the argument to be equal to the type of the parameter.

Figure 3.17 shows the evaluation contexts for the meta language and the object language. Readers can find the explanation of the similar evaluation contexts in Garcia's paper [23] or in Section 3.2.4. Figure 3.16 shows the reduction rules for the implementation of our reflective metaprogramming calculus. An expression is reduced under the constraints collected from its outer evaluation context. For Rule E2, when reducing a lambda abstraction $\lambda x : \sigma^s = \alpha.\ t$, we first check that there are no conflicts among the constraints passed from the outer evaluation context, the constraints generated from the parameter, and the constraints collected from the body t. For Rule E13, when reducing a function application $(\lambda x : \sigma^m.\ t^m)\ v^m$, we first check that there are no conflicts with the constraints collected from t^m and the constraints collected from v^m. For Rule E11, when reducing **typeof** t^o, we need a type-checker to obtain the type of t^o. Implementing a type-checker for proper code is fairly easy (like a type-checker for simply-typed lambda calculus), therefore in this thesis, we omit the detailed description of the implementation. For Rule E14 and

Figure 3.16: The reduction rules.

Object expression reduction $\boxed{C \models t \longrightarrow t'}$

$$\text{(E1)}\ \ C \models \sim \prec t^o \succ \longrightarrow\ t^o$$

$$\text{(E2)}\ \frac{C' = C \cup \{\sigma^s = \alpha\} \cup \mathbb{C}(t) \qquad \mathbb{U}(C') = C''}{C \models \lambda x : \sigma^s = \alpha.\ t\ \longrightarrow\ \{\sigma^s = \alpha\} \Rightarrow \lambda x : \sigma^s.\ t}$$

Meta expression reduction $\boxed{C; \Gamma \models t^m \longrightarrow t^{m\prime}}$

$$\text{(E3)}\ \ C; \Gamma \models \mathbf{dom}\ (\sigma_1 \to \sigma_2)\ \longrightarrow\ \sigma_1 \qquad \text{(E4)}\ \ C; \Gamma \models \mathbf{cod}\ (\sigma_1 \to \sigma_2)\ \longrightarrow\ \sigma_2$$

$$\text{(E5)}\ \ C; \Gamma \models \sigma =^\tau \sigma\ \longrightarrow\ \mathbf{true} \qquad \text{(E6)}\ \frac{\sigma_1 \neq \sigma_2}{C; \Gamma \models \sigma_1 =^\tau \sigma_2\ \longrightarrow\ \mathbf{false}}$$

$$\text{(E7)}\ \ C; \Gamma \models \gamma^?\ \gamma\ \longrightarrow\ \mathbf{true} \qquad \text{(E8)}\ \ C; \Gamma \models \gamma^?\ (\sigma_1 \to \sigma_2)\ \longrightarrow\ \mathbf{false}$$

$$\text{(E9)}\ \ C; \Gamma \models \to^?\ \gamma\ \longrightarrow\ \mathbf{false} \qquad \text{(E10)}\ \ C; \Gamma \models \to^?\ (\sigma_1 \to \sigma_2)\ \longrightarrow\ \mathbf{true}$$

$$\text{(E11)}\ \frac{\Gamma \vdash t^o : \sigma}{C; \Gamma \models \mathbf{typeof}\ t^o\ \longrightarrow\ \sigma} \qquad \text{(E12)}\ \ C; \Gamma \models \%c\ \longrightarrow \prec c \succ$$

$$\text{(E13)}\ \frac{\mathbb{C}(t^m) \cup \mathbb{C}(v^m) \cup C = C' \qquad \mathbb{U}(C') = C''}{C; \Gamma \models (\lambda x : \tau^m.t^m)\ v^m\ \longrightarrow\ [v^m/x]t^m}$$

$$\text{(E14)}\ \frac{\mathbb{U}(C \cup \{\sigma_1^m = \sigma^m\}) = C'}{C; \Gamma \models \mathbf{if\ true\ then}\ t_1^m : \sigma_1^m\ \mathbf{else}\ t_2^m : \sigma_2^m\ \mathbf{withtype}\ \sigma^m\ \longrightarrow\ \{\sigma_1^m = \sigma^m\} \Rightarrow t_1^m}$$

$$\text{(E15)}\ \frac{\mathbb{U}(C \cup \{\sigma_2^m = \sigma^m\}) = C'}{C; \Gamma \models \mathbf{if\ false\ then}\ t_1^m : \sigma_1^m\ \mathbf{else}\ t_2^m : \sigma_2^m\ \mathbf{withtype}\ \sigma^m\ \longrightarrow\ \{\sigma_2^m = \sigma^m\} \Rightarrow t_2^m}$$

Figure 3.17: The selected evaluation contexts.

meta-evaluation contexts E^m

$$\begin{aligned} E^m\ ::=\ & E^m[\mathbf{dom}\ \square]\ \mid\ E^m[\mathbf{cod}\ \square]\ \mid\ E^m[\gamma^?\ \square]\ \mid\ E^m[\to^?\ \square]\ \mid \\ & E^m[\square =^\tau t^m]\ \mid\ E^m[\sigma^s =^\tau \square]\ \mid\ E^m[\mathbf{typeof}\ \square]\ \mid\ E^m[\square \to t^m]\ \mid \\ & E^m[\sigma^s \to \square]\ \mid\ E^m[\square\ t^m]\ \mid\ E^m[v^m\ \square]\ \mid\ E^m[\%\square]\ \mid \\ & E^m[\mathbf{if}\ \square\ \mathbf{then}\ t^m : \sigma^m\ \mathbf{else}\ t^m : \sigma^m\ \mathbf{withtype}\ \sigma^m]\ \mid \\ & E[\lambda x : \square = \alpha.\ t]\ \mid\ E[\sim \square]\ \mid\ E^m[C \Rightarrow \square] \end{aligned}$$

Object evaluation contexts E

$$E\ ::=\ \square\ \mid\ E[\lambda x : \sigma^s.\square]\ \mid\ E[\square\ t]\ \mid\ E[t^o\ \square]\ \mid\ E^m[\prec \square \succ]\ \mid\ E[C \Rightarrow \square]$$

Rule E15, when reducing a meta-level if expression, we need to check that the generated constraint from one of the branches has no conflict with the constraints from its outer evaluation context.

Figure 3.19 shows the single-step meta-evaluation rules. There are two auxiliary functions we need to use. One is function *Ctxt*, which collects the typing information of the object-level variables in an evaluation context. We do not need to collect the typing of the meta-level variables: typing information is only used for reducing **typeof** t^o and obtaining the type of t^o (see Rule E11 in Figure 3.16) and there is no meta variable in t^o.

The other function is $\mathbb{C}$, which collects the constraints in an evaluation context. Collecting the constraints in an evaluation context is different from collecting the constraints in an expression. First, it includes a constraint set C if and only if the evaluation hole is within the scope of C. For example, consider the following meta-evaluation context:

$$\square[C_1 \Rightarrow \square][\square \rightarrow (C_2 \Rightarrow t^m)][C_3 \Rightarrow \square][(C_4 \Rightarrow v^m)\ \square][C_5 \Rightarrow \square][\%\square]$$

Function $\mathbb{C}$ collects the constraints C_1, C_3, and C_5 from the above context. Second, for each collected constraint, if the constraint is a type consistency constraint, we change it into a type equality constraint. Figure 3.18 shows the definition of function $\mathbb{C}$ which collects the constraints inside an evaluation context.

3.4 Summary

For a type-reflective metaprogram, the type of a code fragment can be the result of meta-evaluation. To design a type system for such metaprograms, we need to weigh between static type safety and language expressiveness. A strict type system could compromise the flexibility of metaprogramming while a loose type system could generate imprecise error messages.

In this chapter, we present incremental type-checking, a novel type-checking mechanism for type-reflective metaprogramming languages. Incremental type-checking injects type-checking during the evaluation steps because more type information is available as the result of meta-evaluation. Incremental type-checking can detect type errors in a type-reflective metaprogram as early as possible without sacrificing flexibility.

Figure 3.18: The definition of function $\mathbb{C}$, which collects constraints inside an evaluation context.

Collecting constraints inside E^m $\quad \boxed{\mathbb{C}(E^m) = C}$

$$\mathbb{C}(E^m[\textbf{dom } \square]) = \mathbb{C}(E^m) \qquad \mathbb{C}(E^m[\textbf{cod } \square]) = \mathbb{C}(E^m) \qquad \mathbb{C}(E^m[\gamma^? \ \square]) = \mathbb{C}(E^m)$$
$$\mathbb{C}(E^m[\rightarrow^? \ \square]) = \mathbb{C}(E^m) \qquad \mathbb{C}(E^m[\square =^\tau t^m]) = \mathbb{C}(E^m) \qquad \mathbb{C}(E^m[\sigma^s =^\tau \square]) = \mathbb{C}(E^m)$$
$$\mathbb{C}(E^m[\textbf{typeof } \square]) = \mathbb{C}(E^m) \qquad \mathbb{C}(E^m[\square \rightarrow t^m]) = \mathbb{C}(E^m) \qquad \mathbb{C}(E^m[\sigma^s \rightarrow \square]) = \mathbb{C}(E^m)$$
$$\mathbb{C}(E^m[\square \ t^m]) = \mathbb{C}(E^m) \qquad \mathbb{C}(E^m[v^m \ \square]) = \mathbb{C}(E^m) \qquad \mathbb{C}(E^m[\% \square]) = \mathbb{C}(E^m)$$
$$\mathbb{C}(E^m[\textbf{if } \square \textbf{ then } t^m : \sigma^m \textbf{ else } t^m : \sigma^m \textbf{ withtype } \sigma^m]) = \mathbb{C}(E^m)$$
$$\mathbb{C}(E[\lambda x : \square = \alpha.\ t]) = \mathbb{C}(E) \qquad \mathbb{C}(E[\sim \square]) = \mathbb{C}(E)$$
$$\mathbb{C}(E^m[C \Rightarrow \square]) = \mathbb{C}(E^m) \cup C' \quad \text{where } C' = C \cup \{\sigma_1^m = \sigma_2^m \mid \sigma_1^m \sim \sigma_2^m \in C\} \cup \{\sigma_1 = \sigma_2 \mid \sigma_1 \sim \sigma_2 \in C\}$$

Collecting constraints inside E $\quad \boxed{\mathbb{C}(E) = C}$

$$\mathbb{C}(\square) = \emptyset \qquad \mathbb{C}(E[\lambda x : \sigma^s.\ \square]) = \mathbb{C}(E) \qquad \mathbb{C}(E[\square \ t]) = \mathbb{C}(E)$$
$$\mathbb{C}(E[t^o \ \square]) = \mathbb{C}(E) \qquad \mathbb{C}(E^m[\prec \square \succ]) = \mathbb{C}(E^m)$$
$$\mathbb{C}(E[C \Rightarrow \square]) = \mathbb{C}(E) \cup C' \quad \text{where } C' = C \cup \{\sigma_1^m = \sigma_2^m \mid \sigma_1^m \sim \sigma_2^m \in C\} \cup \{\sigma_1 = \sigma_2 \mid \sigma_1 \sim \sigma_2 \in C\}$$

Figure 3.19: The single-step meta-evaluation rules.

Single-step meta-evaluation for E $\quad \boxed{E[t] \longmapsto E[t']}$

$$\text{(OCtxt-Eval)}\frac{\mathbb{C}(E) = C \qquad C \models t \longrightarrow t'}{E[t] \longmapsto E[t']}$$

Single-step meta-evaluation for E^m $\quad \boxed{E^m[t^m] \longmapsto E^m[t^{m\prime}]}$

$$\text{(MCtxt-Eval)}\frac{\mathbb{C}(E^m) = C \qquad \mathit{Ctxt}(E^m) = \Gamma \qquad C; \Gamma \models t^m \longrightarrow^m t^{m\prime}}{E^m[t^m] \longmapsto E^m[t^{m\prime}]}$$

We present a prototype of incremental type-checking, the formalization of the type system, and the properties which show that generated code is always well-typed. In this chapter, we give a unification-based implementation of incremental type-checking. Instead of type-checking a metaprogram repetitively, we generate, modify, and solve the type constraints generated by our type system.

Chapter 4

Traits

In 2003, Nathanael Schärli et al. invented traits [47]. A trait is a modular building block for classes. It usually provides an implementation of a simple functionality or feature that can be reused by classes or other traits. Additionally, traits provide a flexible way of composing and manipulating features. The concept of traits has a big influence on our work of pattern-based traits, which utilize the composition and method manipulation power of traits to synthesize generated features. In this chapter, we give a brief introduction to traits, including their advantages for code reuse, their syntax, and trait-based metaprogramming (Reppy and Turon's work [45]).

4.1 Limitation of Inheritance-Based Code Reuse

In the field of object-oriented programming, starting with the invention of inheritance by Dahl et al. [13], researchers developed many language features to enable code reuse for classes. Multiple inheritance (Borning [8], Stroustrup [57], and Cardelli [10]) enables a subclass to inherit behaviors from multiple superclasses. Mixin inheritance (Moon [41], Gilad and Cook [9], Matthew et al. [19]) generalizes single inheritance by enabling a subclass to be applied to many different parent classes, thereby extending them with the same set of features. A mixin is a class parameterized over the superclass, so it separates the implementation dependency between a subclass and its superclass.

Inheritance-based code reuse, like the features mentioned above, has limitations. Multiple inheritance can cause implementation ambiguity (called the diamond problem [56]), which refers to the case that a member from a base class is inherited via multiple paths. Fundamentally, the issue

is that multiple inheritance plays two conflicting roles. On the one hand, multiple inheritance is used to implement class polymorphism. At the type level, it is fine for a class to inherit from a set of superclasses that have members with the same type or the same method signature. On the other hand, multiple inheritance is also used to achieve implementation reuse. However, methods with the same type/signature may have different kinds of implementation, which causes implementation ambiguity. Mixins are composed linearly and method ambiguity can be partially solved by method overriding (there is still unsolved method ambiguity that requires programmers to disambiguate via the view operator [19]). However, as specified in the paper by Schärli et al. [47], it is not always possible to find a linear order to compose a set of mixins. Like multiple inheritance, mixin inheritance plays two roles. It is intended for implementation reuse but also complicates the class hierarchy by creating unnecessary subclass relations.

4.2 Traits: A Unit of Behaviors

In 2003, Nathanael Schärli et al. introduced the concept of traits [47, 16]. Traits improve on mixins by generalizing from linear composition to arbitrary composition using symmetric sums and trait nesting. The original proposal for traits was in an untyped setting (the Squeak dialect of Smalltalk) but in the following years, many researchers have investigated type systems for traits (Fisher [18], Odersky et al. [43], Smith and Drossopoulou [55], Allen et al [4], Liquori and Spiwack [36], Bettini et al [7]). In 2007, Reppy and Turon [45] introduced the notion of a trait function, parameterizing traits over member names.

4.2.1 Trait Syntax

A trait expresses a unit of composable methods for the construction of classes. In the following we give the syntax for a basic trait, which is simplified from the trait definition proposed by Reppy and Turon [45] for Java.

trait-declaration ::=

trait *trait-name*

requires { *requirements* }

provides { *member-definitions* }

A trait has a **provides** clause and an optional **requires** clause. The **provides** clause of a trait implements a set of methods. The definitions of those methods may depend on methods not provided by the trait, and those dependencies are listed in a trait's **requires** clause. Trait requirements include method signatures and fields. The following defines the syntax for trait members and trait expressions.

member-definition ::= *method-definition* | **use** *trait-expressions*

trait-expression ::= *trait-name* | *trait-member-manipulation*

The members provided by a trait include method declarations and **use** clauses for composing current trait with other traits, that is to import methods from other traits. A trait expression refers to a trait name (i.e. trait application) or a trait-member-level manipulation, such as member exclusion, member renaming, member aliasing, etc. So, we can compose a trait directly with another trait, or first process the names in the other trait to avoid name conflicts and then compose them together.

4.2.2 A Trait Example

Next, we illustrate the use of traits with a toy example about constructing a class for logging. In the following, two traits are given to construct the class. Trait Log provides public method warning (a public method in a trait is a method accessible by other traits/classes) and requires two members: field isLoggerOn to store the state of whether the logger is on or off and method print to implement the print of log messages. Trait Write provides the two implementations for the print method.

```
1 trait Log
2 requires { boolean isLoggerOn; void print(String s); }
3 provides {
4     public void warning(String sourceClass, String sourceMethod, String msg) {
5         /* partial implementation of warning */
```

```
6           if (! this.LoggerOn) return;
7           ...
8           this.print("Warning: " + msg + "\n");
9           this.print("at " + sourceClass + "." + sourceMethod + "\n")
10          ...
11      }
12  }
13  trait Write
14  provides {
15      public void writeToStdOut(String s) { System.out.print(s); }
16      public void writeToFile(String filename, String s) {
17          /* partial implementation of writeToFile */
18          try {
19              ...
20              output = new java.io.FileWriter(filename);
21              ...
22              bufferedWriter.write(s);
23          } catch (java.io.IOException e) {
24              e.printStackTrace(System.err);
25          }
26      }
27  }
```

If you think of a trait as an implementation of a single feature, then we can generate a more complex feature with the composition of traits. Trait composition has two forms. One is asymmetric composition: when a trait application (i.e. trait **use** clause) is nested inside a trait declaration, it creates a composition order of traits. The other form is the symmetric composition of traits, which refers to the ability to compose a set of traits simultaneously and without generating a composition order. Normally, the symmetric composition is implicitly performed over a set of trait uses. For example, we can construct the following traits LogToStdOut and LogToFile by first

symmetrically composing the traits mentioned above. Trait LogToStdOut prints log messages to a standard output while trait LogToFile prints log messages into a file.

```
1  trait LogToStdOut
2  requires { boolean isLoggerOn; }
3  provides {
4      // Write and Log are symmetrically composed together, the symmetric sum operator is implicit
5      use Write[alias writeToStdOut to print], Log;
6  }
7  trait LogToFile
8  requires { boolean isLoggerOn; String filename; }
9  provides {
10     public print(String s) { this.writeToFile(this.filename, s); }
11      // Write and Log are symmetrically composed together, the symmetric sum operator is implicit
12     use Write, Log;
13 }
```

In both traits, though method print is available, it is no longer required. In trait LogToStdOut, we apply name aliasing to writeToStdOut (in line 4). During meta-level computation, the compiler automatically generates the delegate method print. When the print is called, it forwards the call to method writeToStdOut. In trait LogToFile, we manually define method print, which forwards the call to writeToFile (line 10). At the call of writeToFile, the first argument filename is a required field of the trait.

Traits are building blocks for classes. A class can use a set of traits, importing their method implementations. In the following, we define two classes, LogToFile_C and LogToStdOut_C. Both implements the Log interface.

```
1  interface Log { void warning(String sourceClass, String sourceMethod, String msg); }
2  class LogToFile_C implements Log {
3      private boolean isLoggerOn; private String filename;
4      public LogToFile_C(boolean _isLoggerOn, String _filename) { ... }
```

```
5       use LogToFile;
6       ... // more members
7   }
8   class LogToStdOut_C implements Log {
9       private boolean isLoggerOn;
10      public LogToStdOut_C(boolean _isLoggerOn) { ... }
11      use LogToStdOut;
12      ... // more members
13  }
```

Class LogToFile_C directly uses trait LogToFile and imports all the methods declared in trait LogToFile, Write, and Log. Similarly, class LogToStdOut_C directly uses trait LogToStdOut and imports all the methods declared in trait LogToStdOut, Write, and Log.

Traits separate implementation from concepts. Traits are purely defined for code reuse and for providing implementation of functionalities. As we see from the above example, the flexible composition of traits enables a class to incorporate the implementations of different functionalities. Like the code reuse aspect of multiple inheritance, symmetric composition and trait application enable a class to simultaneously import a set of features from different traits, but unlike the type aspect of multiple inheritance, traits do not introduce any subtype relation nor type polymorphism, which instead is handled by interfaces.

The evaluation of trait composition is called trait flattening [42]. Trait flattening usually happens at compile time: the compiler composes traits, removes composition hierarchy, checks method conflicts, performs method manipulation (i.e. method aliasing, exclusion, overriding and so on), and inlines the methods from traits into classes at the end. Meanwhile, trait definitions are compiled away.

4.2.3 Traits for Metaprogramming

Traits are reusable components. To make traits more general and adaptable to different use environments, Reppy and Turon [45] introduced the concept of trait functions and demonstrated that they can be generically used to support metaprogramming.

A trait function is a trait template, which can be parameterized over types, member names, and meta values (constants). The following is the syntax of a trait function.

trait-function-declaration ::=

trait *trait-name*(*member-name-parameters*, *type-parameters*, *value-parameters*)

requires { *requirements* }

provides { *member-definitions* }

According to the original definition, a trait does not provide any fields. But when using traits to implement features, we need states. In many cases, when implementing a feature using traits, we need to encapsulate the fields which are bound to the feature, and have full control over those fields. Because the body of a trait function can have provided fields, we can implement a trait function that generates the setter and the getter for a generic field (the following code is adopted from Reppy and Turon's paper [45]):

```
1 trait PropT ($f, $g, $s, T)
2 provides {
3     private T $f;
4     public void $s (T x) { $f = x; }
5     public T $g () { return $f; }
6 }
```

Suppose we need to construct class Account that has field full_name of type String and field balance of type **int**. We can use trait PropT to generate the getters and setters for those fields instead of writing them manually:

```
1 class Account {
2     public Account(String full_name, int balance) {
```

```
3         this.full_name = full_name; this.balance = balance;
4     }
5     use PropT (full_name, getFull_name, setFull_name, String);
6     use PropT (balance, getBalance, setBalance, int);
7 }
```

During meta-evaluation, the members in trait PropT are instantiated for each of the trait applications and instantiated members are inlined into class Account.

Here is another example: we define trait function of BackupT which can extend a class with the field backup feature.

```
1  trait BackupT ($s, $g, $bf, $backupf, $restoref, T)
2  requires {
3      T $g();
4      void $s(T x);
5  }
6  provides {
7      private T $bf;
8      public void $backupf() { $bf = $g(); }
9      public void $restoref() { $s($bf); }
10 }
```

In the following, class BackupAccount, a subclass of class Account uses trait BackupT to extend Account with the balance backup feature:

```
1  class BackupAccount extends Account {
2      use BackupT (setBalance, getBalance, balance_backup, backupBalance, restoreBalance, int);
3  }
```

Trait functions enable programmers to write a set of member templates for class extension. However, current trait functions require users to manually infer names and types for each trait application. Moreover, users have to write a trait instance for each member that needs to be extended by the trait. If a class has M members and the class is supposed to be extended by N

traits, then users need to write $M * N$ trait application instances. It is cumbersome to write those instances manually if $M * N$ is large number. Reppy and Turon noticed the drawback and in their paper [45], they proposed to use pattern matching for automatically generating instances. This approach is briefly discussed in their technical report [60], but it has not been formalized.

Reflection facilitates metaprogramming. Inspired by trait functions and reflection, in the next chapter we propose pattern-based traits, the combination of pattern-based reflection with traits, which offer expressive forms of composition and the ability to inspect members and retrieve meta information for automatic instantiation.

4.3 Summary

In this chapter, we begin with the discussion about the limitation of inheritance-based code reuse. We give an overview of traits and a line of language features related to traits. A trait is unit of single functionality and it is intended for code reuse. Traits provide the symmetric sum, a more flexible composition than inheritance.

Chapter 5

Pattern-Based Traits

Programming languages with the feature of class reflection provide mechanisms for inspecting old classes and generating new ones. In 2006, Fähndrich et al [17] proposed compile-time reflection by providing high-level patterns and templates, which we refer to as pattern-based reflection. Huang and Smaragdakis [25, 26] increased the expressiveness and safety guarantees of pattern-based reflection and realized their results in the MorphJ language. In MorphJ, a pattern is a unit for iterating over class members and generating new members. Because patterns are defined in morph classes (like mixins that are parameterized over superclasses), MorphJ relies on mixin inheritance (with its linear ordering) for composing generated class members, which limits expressiveness. In the introduction of our paper [39], we give an example about composing a morph class for generating logged getters. The example shows that it can be difficult to create a desired morph class only by using linear composition.

In this chapter, we present pattern-based traits, where pattern-based reflection is transplanted into the setting of traits, thereby taking advantage of expressive forms of composition. Pattern-based traits increase the expressiveness of pattern-based reflection by providing the programmer with increased control over how the generated fragments of functionality are combined.

Pattern-based traits are parameterized over sets of member declarations (not just member names). They generalize trait functions [45] and provide the programmer with control over which set of declarations a pattern is applied to. Pattern-based traits provide mechanisms for grouping, naming, and manipulating sets of member declarations.

5.1 Pattern-Based Traits Syntax

In this section, we review the notion of traits and pattern-based reflection and we introduce the key features of our language, called Pattern-Based Traits for Java (PTJ for short). We employ a running example about generating logged getter methods. With this example, we show how to add synchronization and logging to overlapping sets of methods.

5.1.1 Traits in PTJ

Traits in PTJ are derived from the basic notion of traits described in Chapter 4. A trait in PTJ expresses a unit of code reuse. It has the **provides** clause containing a set of implemented methods, which may depend on methods not provided by the trait, and those dependencies are listed in the trait's **requires** clause. The following is the syntax for traits in PTJ.

trait-declaration ::=

trait *trait-name*<*group-parameters*>

requires { *requirements* }

provides { *members* }

Traits in PTJ are parameterized by group variables. As usual, trait requirements include method signatures and fields. The members provided by a trait include methods, fields, and trait expressions. A trait may need fields for internal use; it is better to declare such fields within the trait. The first addition for PTJ is pattern declarations, which integrate pattern-based reflection into traits. The second addition is named groups of members. Borrowed from the basic trait syntax, a PTJ trait has the **use** clause to import members from other traits. In PTJ, trait expression refers to trait application, trait member manipulation, and interface satisfaction assertion. Trait application applies the trait to a sequence of group expressions (group expressions are discussed in Section 5.1.4), for reflecting over the group members and generating specialized code. Like basic traits, PTJ traits provide the capability of member-level manipulation, such as member renaming, member aliasing, and member exclusion. Interface satisfaction assertion enables users to access

generated members by explicitly specifying their interfaces. The following defines the syntax for trait members and trait expressions.

member ::= *method-definition*
| *field-definition*
| *pattern-declaration*
| **group** *group-name* { *members* }
| **use** *trait-expressions*

trait-expression ::= *trait-name*<*group-expressions*>
| *trait-member-manipulation*
| *interface-satifaction-assertion*

In the next section we discuss patterns and their use in an example.

5.1.2 Patterns

Reflective patterns in PTJ are similar to those (e.g. **for** constructors) in MorphJ [25] in that one can pattern-match against either fields or methods. In addition, our reflective patterns can pattern-match against class constructors. The body of the pattern is evaluated once for each declaration in the given range, and the resulting declarations are inlined into the enclosing trait or class. The syntax for patterns in PTJ is as follows.

pattern-declaration ::=
pattern [*pattern-name*]<*pattern-parameters*> *pattern* **in** *range*
{ *members* }

pattern-paramter ::= *name-parameter*
| *type-parameter*
| *type-list-parameter*

pattern ::= *field-pattern*
| *method-pattern*
| *constructor-pattern*

In the syntax, the pattern name is optional. A pattern parameter refers to a name variable, a type variable, or a type list variable. A type pattern variable can pattern-match over all non-void-types, including Java primitive datatypes. A type pattern parameter can be type bounded, which supports intensional type analysis [24]. A type list pattern parameter is a parameter that matches over a list of types. As mentioned previously, a group refers to a collection of methods and fields. For instance, the body of a class is a group as it contains a collection of methods and fields, but the notion of a group is more general than a class body.

We have explained enough features of PTJ to look at some examples. The following trait adds logging to the public and non-void-returning methods in group M, where those methods do not have the **void** return type.

```
1  trait LogMethods<group M>
2  provides {
3      pattern P<name m, T, U*> public T @m(U) in M {
4          public T logged#@m(U x) {
5              System.out.println("Enter method " + @m::toString());
6              T t = this.@m(x);
7              System.out.println("Leave method " + @m::toString());
8              System.out.println("Method " + @m::toString() + " returns " + t);
9              return t;
10         }
11     }
12 }
```

The **pattern** constructor combines iteration and pattern matching. Inside trait LogMethods, pattern P1 (lines 3-11) generates the logging methods for methods that return something. The syntax <**name** m, T, U*> in line 3 declares three pattern variables (one name variable, one type variable, and one type list variable). The syntax **public** T @m(U) is a method pattern. We use the @ symbol to distinguish a name variable from a name constant. The compiler can infer a type list variable (e.g. variable U), so we do not need to use any annotation to distinguish it from a type variable.

In line 4, the # symbol is the name concatenation operator. The syntax @m::toString() calls the meta method toString in name @m, where the :: symbol represents the access of a meta method. Meta method toString returns the string representation of a name at compile time. Meta methods are defined for the inspection over meta-level objects, such as types and names. They are invoked at compile time.

PTJ supports nested patterns. In the following, we present trait VisitFields which generates field visitor methods for each field in group M.

```
1  trait VisitFields<group M>
2  requires {
3      Object visit(Object n);
4  }
5  provides {
6      pattern P_f <name f, T extends Object> * T @f in M {
7          pattern P_get <> public T get#@f() in M {
8              pattern P_set <> public void set#@f(T) in M {
9                  public void visit#@f() {
10                     T fd = this.get#@f();
11                     fd = (T) this.visit(fd);
12                     this.set#@f(fd);
13                 }
14             }
15         }
16     }
17 }
```

Creating visitors for fields supports the implementation of the visitor design pattern [3]. Inside VisitFields, there are three levels of pattern declarations. Pattern P_f iterates over all the fields in group M. In the pattern's header, type parameter T is bounded by Object, so P_f pattern-matches the fields whose types are object types, so that it excludes fields of primitive types. Symbol * is the

universal modifier, a modifier pattern that pattern-matches over all kinds of modifiers. Modifier patterns will be discussed in Section 5.2.2. Pattern P_get matches the getter method for field @f and pattern P_set matches the setter method for @f. If field @f has the getter and the setter methods, then VisitFields creates a visitor method for @f. In the visitor method, the field's value is applied to function visit. Function visit usually performs the transformation of an object and we can implement it for different functions.

5.1.3 Groups

According to the syntax given in Section 5.1.1, a group construction is a named trait/pattern body. It can define its own methods, generate methods via patterns, and import methods from other traits via trait applications. At runtime, a group value, like a namespace, binds a collection of methods.

Before showing an example about how to use groups, we first present the following trait. The trait adds synchronization to the public and non-void-returning methods specified in group M. It requires the class that uses the trait to provide a lock field of type Mutex. We assume type Mutex has the methods acquire and release;

```
1 trait SynMethods<group M>
2 requires { Mutex lock; }
3 provides {
4     pattern <name m,T,U*> public T @m(U) in M {
5         public T syn#@m(U x) {
6             lock.acquire(); T t = this.@m(x); lock.release(); return t;
7         }
8     }
9 }
```

Next, we wish to both synchronize and add logging to the methods of a class. We can accomplish this with the use of named groups of member declarations.

```
1 trait SynAndLogMethods<group M>
2 requires { Mutex lock; }
3 provides {
4   group G { use LogMethods<M>; }
5   use SynMethods<G>;
6 }
```

The members within a group are inlined into the enclosing trait, and they are also associated with the given name, in this case G. In line 5, the logged methods wrapped by group G are passed to SynMethods, which generates synchronized and logged methods.

5.1.4 Ranges

We have defined the necessary traits for our example. We use them to augment the following class that represents a bus schedule. We assume class Employee and class Ticket are given.

```
1 class Bus {
2   public Report safety_check(Employee inspector) {...}
3   public Ticket sell() {...}
4   public void refund(Ticket ticket) {...}
5   public Ticket transfer(Ticket ticket, Bus bus) {...}
6   ...
7 }
```

We wish to synchronize the refund, sell, and transfer methods and add logging to safety_check, sell, and transfer. To select the right methods to which apply our traits, we have one last feature of PTJ to introduce, ranges.

In PTJ, ranges are the same as group expressions, which represent different kinds of group applications. A range generates a collection of members that a pattern can iterate over. The syntax for range expressions is as follows.

range ::= *group-expression*

group-expression ::= *identifier*

| *identifier* + *identifier*

| *group-expression*\{*identifiers*}

| *group-expression*|{*identifiers*}

The identifier refers to a group name, a group variable, or a class name.

The plus operator generates the symmetric sum of the two groups. The difference operator (\) removes the members specified by a set of identifiers from the range. The restriction operator (|) generates the range that consists of just the specified members.

Using the restriction operator, we can generate the following extended bus class with the desired combination of synchronized and logged methods.

```
1 class SafeBus extends Bus {
2     private Mutex lock;
3     public SafeBus() { super(); lock = new lock() };
4     use SynMethods<Bus|{refund}>;
5     use LogMethods<Bus|{safety_check}>;
6     use SynAndLogMethods<Bus|{sell,transfer}>
7 }
```

Alternatively, we can first define the following versatile and general trait GeneralSynLogMethods for method synchronization and/or logging, and then create the subclass SafeBus with GeneralSynLogMethods.

```
trait GeneralSynLogMethods<group Syn, group Log, group SynLog> {
    use SynMethods<Syn>, LogMethods<Log>, SynAndLogMethods<SynLog>;
}

class SafeBus extends Bus {
    private Mutex lock;
    public SafeBus() { super(); lock = new lock() };
    use GeneralSynLogMethods<Bus|{refund}, Bus|{safety_check}, Bus|{sell,transfer}>
}
```

5.1.5 Interface Satisfaction Assertion

During meta-evaluation, a pattern generates a set of methods for later use. Before meta-evaluation, it is hard for our type system to obtain the type information of those generated methods, thus preventing us from using those methods. For example, the following program defines two classes. Class Account has method withdraw and class LoggedAccount uses trait LogMethods (see definition in Section 5.1.2) to log method withdraw.

```
1  class Account {
2      private double balance;
3      public bool withdraw(float x) { ... }
4  }
5  class LoggedAccount extends Account {
6      // generate bool loggedWithdraw(float)
7      use LogMethods<Account>;
8  }
9  // main program
10 new LoggedAccount().loggedWithdraw(1000); // type error
```

During meta-evaluation, LogMethods generates method loggedWithdraw which is then inlined into class LoggedAccount. In line 10, the program calls LoggedAccount from an instance of class LoggedAccount. Because the type system cannot statically find loggedWithdraw in class LoggedAccount, it shows a type error. One way to access loggedWithdraw is via pattern-reflection at the meta level. In the following program, we use trait T to access method loggedWithdraw and class MyLoggedAccount uses trait T.

```
1  trait T<group X>
2  provides {
3      pattern P <> public bool loggedWithdraw(float) in X {
4          public bool callLoggedWithdraw(float x) {
5              // access loggedWithdraw inside a pattern
6              return this.loggedWithdraw(x);
7          }
```

```
8       }
9   }
10  class MyLoggedAccount extends LoggedAccount {
11      use T<LoggedAccount>;
12  }
```

However, callLoggedWithdraw is still a generated method in class MyLoggedAccount and we cannot call it outside the class.

Adopting the idea of Java type cast, we introduce interface satisfaction assertion for a trait expression. Interface satisfaction assertion has the following syntax. Member signatures include field signatures and method signatures.

interface-satifaction-assertion ::= *trait-expression* **satisfies** { *member-signatures* }

Interface satisfaction assertion enables users to explicitly specify a sequence of members that a trait expression can generate. In the following program, class loggedAccount uses interface satisfaction assertion to assert that the application of trait LogMethods can generate method **bool** loggedWithdraw(**float**) during meta-evaluation. The type system assumes method loggedWithdraw is available.

```
1   class LoggedAccount extends Account {
2       use LogMethods<Account> satisfies { bool loggedWithdraw(float); };
3   }
4   // main program
5   new LoggedAccount().loggedWithdraw(1000); // type ok
```

We use a new keyword **satisfies** instead of using Java's **implements** because we do not want to confuse interface satisfaction assertion with interface implementation in Java. For interface implementation, Java statically checks whether a class implements an interface. Here, the checking of interface satisfaction assertion is performed during meta-evaluation. Interface satisfaction assertion does not hide type information. Suppose a trait statically provides method m. If the trait is used and asserted to satisfy the interface of method n, we can still access its method m.

5.2 Member Access Control

For class encapsulation, member access control enables a class to seal its internal members, and to restrict its members from being used outside the class. Unlike Java runtime reflection, which allows programs to disable default access control, compile-time reflection needs to respect class encapsulation so that generated code does not violate Java's member access control rules.

5.2.1 Group Member Accessibility

In this section, we discuss the accessibility of group members. For simplicity, we consider three levels of access: public, protected, and private.

First, we give the following trait that adds getter methods for the public fields in group F.

```
1 trait AddGettersForPublic<group F>
2 provides {
3     pattern P<name f, T> public T @f in F {
4         public T get#@f() { return this.@f; }
5     }
6 }
```

In line 4, the program accesses group F's public fields via the **this** variable. If we allow F to be instantiated with any group expression, then such field access is not always valid. Consider the following case when AddGettersForPublic is applied to A, and is used in both class B and class C where C is a subclass of A.

```
1 class A { public int i; }
2 class B {
3     use AddGettersForPublic<A> // Error
4 }
5 class C extends A {
6     use AddGettersForPublic<A>
7 }
```

In B, a getter method for field i is generated and uses the **this** variable to access i, but i is not a member of B.

To solve the above problem, we have two options. One is to restrict the set of classes a trait can accept. The other is to refine the concept of groups into accessible groups, a category of groups that enable us to access its members via the **this** variable. We take the second option, which does not restrict the set of programs we can write. For example, it is possible for programmers to write a trait that only inspects the members of a group without accessing them. The following is the syntax for trait parameters.

trait-parameter ::= **group** *identifier*
| **abstract group** *identifier*

A trait parameter specified with **group** expects an accessible group expression; and a trait parameter specified with **abstract group** can accept any group and treat it as non-accessible.

Extending private members can be useful. We can hardly extend something for private members using inheritance-based class extension, such as MorphJ. But pattern-based traits can extend a class in place, so ideally, we would like them to work for private members as well. For example, the following program presents two traits and a class.

```
1  trait AddGettersForPrivate<group F>
2  provides {
3      pattern P<name f, T> private T @f in F {
4          public T get#@f() { return this.@f; }
5      }
6  }
7  trait AccountTrait<> {
8      group F {
9          private double balance;
10         private double creditLimit;
11     }
12     use AccountTrait<F>;
```

```
13 }
14 class Account {
15     public Account() { this.balance = 0; this.creditLimit = 1000; }
16     use AccountTrait<>;
17 }
```

Trait AddGettersForPrivate adds getters for private fields. Trait AccountTrait is an auxiliary trait for constructing class Account. It is a member provider for Account. It has fields balance and creditLimit; and it uses the former trait to create the getter method for those fields. Class Account is for creating account instances. The type system accepts the above program because balance and its getter method are in the same local scope. But if trait AddGettersForPrivate is used like trait AddGettersForPublic (i.e. generating getters for the fields in a super class), then it violates the member access control rules because the private members of a class cannot be accessed outside the class.

We need to refine the concept of accessible groups further for accessing private members. We introduce the concept of **local** groups, a category of groups that enable us to access all its members (including private ones) via the **this** variable. The following is the refined syntax for trait parameters. A trait parameter specified with **local group** expects a local group expression.

trait-parameter ::= **group** *identifier*
| **abstract group** *identifier*
| **local group** *identifier*

In summary, if a group is accessible, then we can access its non-private members; if a group is abstract, then we cannot access its members; and if a group is local, then we can access all its members. The computation of the exact group concept for a group expression is called group categorization, which is performed by the type system.

5.2.2 Modifier Patterns

Inside a pattern's header, we can specify a Java modifier to match a set of members with the same modifier. We introduce modifier patterns, enabling a pattern to match a set of members with different modifiers. In line 3 of the following traits, negative modifier **nonprivate** indicates that the pattern cannot match over private members. Modifier **nonprivate** is consistent with the accessible group concept of group F.

```
1 trait AddGettersForNonPrivate<group F>
2 provides {
3     pattern P<name f, T> nonprivate T @f in F {
4         public T get#@f() { return this.@f; }
5     }
6 }
```

In addition, we introduce the universal modifier, written as $*$, which enables a pattern to match over members with all kinds of modifiers. So, we create the following trait to generate getters for fields with all kinds of access levels.

```
1 trait AddGettersForAll<local group F>
2 provides {
3     pattern P<name f, T> * T @f in F {
4         public T get#@f() { return this.@f; }
5     }
6 }
```

Trait AddGettersForAll accepts a local group, so it can access its fields with all kinds of modifiers, which is consistent with the universal modifier.

Last but not least, we introduce the dynamic modifier, written as ?. it is a placeholder for an actual modifier pattern. During meta-evaluation, it can be substituted for some actual modifier pattern. Using the dynamic modifier, we can unify trait AddGettersForNonPrivate and trait AddGettersForAll as the following single trait.

```
1 trait AddGetters<group F>
2 provides {
3     pattern P<name f, T> ? T @f in F {
4         public T get#@f() { return this.@f; }
5     }
6 }
```

Inside AddGetters, during meta-evaluation, the ? is instantiated into the most general (least restrictive) modifier pattern that is consistent with the group concept of F's value. For instance, if AddGetters is applied to a group name, then the ? is instantiated into the universal modifier. If AddGetters is used inside a class and applied to its superclass, then the ? is instantiated into the **nonprivate** modifier.

5.3 Type-Checking of Pattern-based Traits

As we have already mentioned: for the design of a type system for a reflective metaprogramming language, there is a tension between static type safety and language expressiveness. A modular type system supports static type safety. It can catch type errors at their place of origin but it may restrict the set of metaprograms one can write. On the other hand, non-modular type systems accept a larger set of metaprograms, but some of those metaprograms may generate ill-typed code which is difficult to debug because the error messages point into the generated code instead of the source of the error in the metaprogram.

5.3.1 Static Type Safety vs. Language Expressiveness

Traits are intended to be widely used. If we do not design a modular type-checker for traits, then error messages could point to the places deep inside a trait or the places at the top of a composition hierarchy. Such kind of error messages are hard to understand. To modularly type-check a pattern-based trait, we require the trait to be fully type-checked at definition time. Moreover, at a place where the trait is applied, the error messages should never refer to the implementation of

the trait members.

However, designing a modular type system for pattern-based traits is hard. The members of a trait can be generated during meta-evaluation. In other words, the members of a trait may be result of meta-evaluation. This feature prevents us from directly adopting the modular type-checking scheme for ML modules [46] or the modular type-checking scheme for FeatherTrait [36].

MorphJ [25] presents a modular type system that guarantees static type safety of Morphing classes. The type system prevents member name conflicts by checking the disjointness between patterns (e.g. two patterns do not generate methods with the same signature if they do not subsume each other). Checking the disjointness between two patterns is complicated; the generated error messages can be difficult to understand. Moreover, the type system may prevent us from writing some useful programs. If we use Huang's approach and extend our type system with pattern disjointness checking, then the type system would definitely preclude useful programs. In addition, if we fully and statically type-check each interface satisfaction assertion in a metaprogram, then the type system would reject the set of metaprograms that try to access generated methods, which is against the purpose of interface satisfaction assertion.

We design a modular type system that is different from MorphJ's type system. Although our type system cannot prevent name conflicts before meta-evaluation, it does not impose restriction on the set of reflective patterns that a programmer can write. Our type system modularly type-checks pattern-based traits and it guarantees (1) a trait is type-checked independently from all of it applications, (2) a well-typed pattern-based trait always generates well-typed methods, which means our type system guarantees static type safety at the method level, and (3) the code generated by a well-typed pattern-based trait is always well-typed.

Please note that our type system does not guarantee that a well-typed pattern-based trait can always generate code. First, name conflicts still exist during meta-evaluation. Second, an interface assertion can be unsatisfied. Therefore, the type system will throw a runtime exception if there is an unsolved name conflict or an unsatisfied interface assertion. However, if there is no name conflict and all of the interface assertions are satisfied, then a pattern-based trait can always

generate well-typed code.

5.3.2 Nominal Typing and Structural Typing

In our type system, we give both the structural and the nominal presentations of a class. The structural presentation of a class facilitates member inspection. It not only enables us to inspect the members of a class, but also iterate over members. Moreover, the structural presentation of a class is mutable, which supports member aliasing, member exclusion, renaming, the merge of two groups of members, and so on. The nominal presentation of a class is used to determine subtyping relation and equivalence of classes. It describes a high-level concept and supports type polymorphism. For example, consider the following program:

```
1  trait T<group X>
2  provides {
3      pattern P <T extends java.util.List, U*, name m> public T @m(U) in X {
4          public Set @no_duplicate_m(U x) {
5              java.util.Set set = new java.util.HashSet();
6              set.addAll(this.@m(x));
7              return set;
8          }
9      }
10 }
```

In line 3, type parameter T is bound by interface List: we select the methods whose return type implements the List interface. Nominal subtyping checks if a class implements an interface; an interface extends another interface; a class extends another class, etc. Also in line 3, when iterating and pattern-matching against the members in X, we use the structural type of X. Even X is substituted by some class C, the nominal information of C will be compiled away. In line 5, when an instance of Set is assigned to variable set, we use nominal subtyping to check that the instance implements the Set interface. In line 6, when we access method @m via the **this** variable, instead of looking up the type of the method in the type of the **this** variable, we get the type of the method

from the structural presentation of the type of the **this** variable. Our type system calculates and passes along the structural information for the type of the **this** variable.

5.3.3 Scope-based Type Lookup

In our type system, the accessibility of trait members is checked based on the trait composition hierarchy. Every meta declaration (i.e. meta class, trait, and pattern) introduces a scope. If a pattern is nested inside a trait or nested inside another pattern, then it introduces a nested scope. Intuitively, methods defined in the same scope can access each other. A method defined in an inner scope can access the members in its outer scopes. Consider the following example:

```
1  trait FieldBackup<group X>
2  provides {
3      pattern PX<T, name m> private T @m in X {
4          private T @m#backup;
5          pattern PY<> public T get#@m() in X {
6              pattern PZ<> public void set#@m(T) in X {
7                  public void safeSet#@m(T x) {
8                      T obj = this.get#@m();
9                      this.@m#backup = obj;
10                     this.set#@m(x);
11                 }
12                 public void restore#@m() {
13                     T obj = this.@m#backup;
14                     this.safeSet#@m(obj);
15                 }
16             }
17         }
18     }
19 }
```

Trait FieldBackup provides the field backup feature. For each field, it generates a backup field (in line 4); a safeSet function (lines 7-11) that backups the field before it is set to the new value; and a restore function (lines 12-15) that restores the backup value. Please note that function restore itself calls the safeSet function, which means it backups the current state before it restores. Pattern PZ is nested inside pattern PY and pattern PZ, so it can access their members, such as the backup field @m#backup and the get method get#@m().

5.4 Formalization

In this section, we formalize pattern-based traits. Our calculus, PTFJ, is a minimal core calculus for Pattern-Based Traits for Java. It extends Featherweight Java (FJ) [30] with traits, pattern-based reflection, and groups. PTFJ supports reflective metaprogramming. It is a meta language for class member reflection and code generation. During meta-evaluation, PTFJ is translated into FJ, which is the object language.

5.4.1 FJ Syntax

For readers' convenience, we first give the syntax of Featherweight Java [30] in Figure 5.1. A FJ program consists of a fixed global class table CT plus a main expression $\mathtt{e}^o$. A class table maps class names to their corresponding class declarations. A main expression is the body of a main function in Java. We assume the metavariables B and C range over class identifiers. In FJ, the variable this is a predefined variable for self reference. The syntax $\mathtt{super(}\overline{\mathtt{f}^o}\mathtt{)}$ refers to the call to the superclass's constructor. In FJ, the methods and the fields in a class are assumed to be public and they can be accessed by other classes. FJ expressions include variables, field access expressions, method invocations, object creations, and type casts. FJ requires the fields in a class to have names distinct from those in its superclasses.

Figure 5.1: Syntax of Featherweight Java

x	variable
m^o	method name
f^o	field name
class	L^o ::= class C extends C { $\overline{F^o}$; K^o $\overline{M^o}$ }
constructor	K^o ::= C($\overline{C}$ $\overline{f^o}$){ super($\overline{f^o}$); this.$\overline{f^o}$=$\overline{f^o}$; }
field	F^o ::= C f^o
method	M^o ::= C m^o($\overline{C}$ $\overline{x}$){ return e^o; }
expressions	e^o ::= x | e^o.f^o | e^o.m^o($\overline{e^o}$) | new C($\overline{e^o}$) | (C)e^o
FJ program	p^o ::= <CT,e^o>

5.4.2 PTFJ syntax

The PTFJ calculus is based on Featherweight Java [30]. It is the minimal core syntax for Pattern-Based Traits. We show the PTFJ syntax in Figure 5.2. We omit name exclusion, aliasing, and renaming because they have been fully explained by Ducasse et al. [16] and Bettini et al. [6]. For simplicity, we omit the following syntax: field in trait body, class constructor pattern, type list parameter, and type bound over type parameter. Adding them into PTFJ will only need slight change of the current semantics. Because FJ does not have member modifiers, there are no private members. Therefore, we do not need the local group concept in PTFJ. For simplicity, PTFJ only considers accessible groups and we assume a trait parameter always accepts an accessible group.

We assume the metavariable Tr ranges over trait identifiers; Pt ranges over pattern identifiers; and G ranges over group identifiers. We require trait identifiers, pattern identifiers, and group identifiers to be disjoint.

Meta classes, as the names indicates, are meta-level declarations. They are evaluated into FJ classes at compile time. Like FJ, reserved class Object appears only in the kernel and it is the top of all meta classes.

A name l can be in a restricted concatenation form s#s, which enables the compiler to split a name and pattern-match its subpart. The concatenation of two name constants is evaluated into a new name constant by performing the string-like concatenation. Two simple names are equal if

Figure 5.2: Kernel Language Syntax of PTFJ, an extension of FJ syntax

`x`	variable
`m`	method name
`f,g`	field name
`X`	type variable
η	name variable
χ	accessible group variable
`c`	name constant
#	name concatenation operator

class	`L ::= class C extends C { F̄; K M̄ }`
constructor	`K ::= K`^{o}
field	`F ::= T f`
method	`M ::= T m(T̄ x̄){ return e; }`
expressions	`e ::= x | e.f | e.m(ē) | new C(ē) | (T)e`

Syntax that extends FJ:

types	`U,V,S,T ::= X | C`
trait	`TR ::= trait Tr<`$\overline{\chi}$`> req I prov D`
meta class	`MC ::= L uses Ē`
pattern	`PT ::= pattern Pt<`$\overline{X};\overline{\eta}$`> P in R D`
group	`GP ::= group G D`
member patterns	`P ::= F | H`
method signature	`H ::= T m(T̄)`
member signatures	`I ::= { F̄; H̄ }`
body	`D ::= { M̄; ḠP; P̄T; use Ē }`
simple names	`s ::=` η `| c`
names	`l ::= s | s#s`
trait applications	`E ::= Tr<R̄> | E satisfies { H̄ }`
group ranges	`R ::=` χ `| G | C | I`
PTFJ program	`p ::= <MCT,TTT,e>`

they are syntactically equivalent. Two name concatenation forms are equal if their first and second parts are equal as well. Field names (`f` and `g`) and method names (`m`) are both derived from names. Field names are distinct from method names.

In the syntax of ranges, we include member signature set `I`, which is range value: a group is evaluated to be a set of member signatures during meta-evaluation.

We define a PTFJ program as `<MCT,TTT,e>` where MCT is the meta class table (a mapping from meta class identifiers to their definitions), TTT is the trait table (a mapping from trait identifiers to their definitions), and `e` is the main expression of a program in Pattern-Based Traits for Java. The meta class table MCT needs to satisfy some sanity conditions that are very similar to those for the class table CT in FJ. The following are the conditions presented by Igarashi et al [30]: a class identifier should map to a class definition with the same identifier; `Object` is not mappable in MCT; when a class identifier except `Object` is used, we can always obtain its definition from MCT; and the subtype relations in MCT are acyclic. For TTT, we have the following sanity conditions: a trait identifier should map to a trait definition with the same identifier, and for any trait identifier that appears in the program, we can always obtain its definition from TTT.

5.4.3 Typing Rules

We present a type system for PTFJ. Our type system uses an incremental type-checking strategy. The static type-checking subsystem (prior to meta-evaluation) type-checks a PTFJ program thoroughly. It ensures all the expressions are well-typed, each single method definition is well-typed, and the type errors detected through dynamic checks never rise at the expression level. The static type-checking checks name conflicts. It catches the name conflicts among statically declared methods, but not the methods generated by patterns. Instead, name conflicts caused by generated methods are caught through dynamic checking.

In this section, we focus on the static type-checking part of PTFJ. First, we give some preliminary definitions for typing (see Figure 5.3). We use member types to describe the types of fields and methods. Type $\overline{\mathtt{T}} \rightarrow \mathtt{T}$, called method type or function type, is for methods: $\overline{\mathtt{T}}$ is the set

Figure 5.3: Preliminary definitions used by the PTFJ type system.

method type	$\phi \ ::= \ \overline{\mathtt{T}} \to \mathtt{T}$
member types	$\delta \ ::= \ \mathtt{T} \mid \phi$
nominal types	$\mathtt{N} \ ::= \ \mathtt{T} \mid \mathtt{thistype}$
structural type	$\sigma \ ::= \ \{\mathtt{l}_i = \delta_i\}^{i \in 1 \ldots n}$
object type	$\tau \ ::= \ \mathtt{N} \diamond \sigma$
variable binding context	$\Gamma \ ::= \ \emptyset \mid \Gamma, \mathtt{x} : \tau \mid \Gamma, \mathtt{X} \mid \Gamma, \eta \mid \mid \Gamma, \chi \mid$
member typing context	$\Delta \ ::= \ \cdot \mid \Delta[\mathtt{l} \mapsto \delta] \mid \Delta \oplus \Delta \mid \Delta \ni \Delta$
group names	$\Theta \ ::= \ \emptyset \mid \Theta \cup \{\mathtt{G}\}$

of parameter types and T is the return type. Two method types are equal if their parameter types and return types are equal respectively, that is $\overline{\mathtt{U}} \to \mathtt{U} = \overline{\mathtt{T}} \to \mathtt{T}$ if $\overline{\mathtt{U}} = \overline{\mathtt{T}}$ and $\mathtt{U} = \mathtt{T}$.

Nominal types are type variables and class names. We use them for determining subtype relations and type equality. Type variable thistype is reserved for internal use. The type system uses it to describe the type of the **this** variable when it type-checks inside traits. A structural type reveals the inner members of a class-like structure. Here, notation $\{\mathtt{l}_i = \delta_i\}^{i \in 1 \ldots n}$ is a set of bindings for member names to their member types, the same as record type (see Chapter 11 of the Piece's book [44]). We use structural types to inquire about member types. We use object types to describe both the nominal and the structural part of a class. Together they form a complete reflection of a class. We borrowed this idea from the MTJ calculus [45].

Our definition for a variable binding context is the same as the definition given in the Piece's book [44]. A variable binding context Γ is a finite sequence of bindings. It has four kinds of bindings: term variable typing binding $(\Gamma, \mathtt{x} : \tau)$, type variable binding $(\Gamma, \mathtt{X})$, name variable binding (Γ, η), and group variable binding (Γ, χ). When a variable binding is added into a context, we assume the variable has been renamed so that it is distinct from the variables bound by context. So in a context, each variable has only one binding.

A member typing context Δ is a finite mapping from names to member types. The notation $\cdot$ means an empty mapping. The notation $\Delta[\mathtt{l} \mapsto \delta]$ means extending context Δ with the mapping entry $\mathtt{l} \mapsto \delta$ if $\mathtt{l} \notin \mathit{dom}(\Delta)$; otherwise the old mapping entry for l is replaced by the new one.

The notation $\Delta \oplus \Delta$ means combining two member typing contexts into a single new context by the symmetric sum. Two contexts are merged into a new context if they have no bindings for the same name; otherwise we get an type error because of duplicate members. The notation $\Delta \Ni \Delta$ means combining two member typing contexts into a new one context by the asymmetric sum. We can always successfully perform asymmetric sum on two contexts even if they have overlapping mapping entries. For the asymmetric sum, a mapping entry in the left context overrides the mapping entry in the right if they are for the same name. A member typing context can be syntactically translated into a class structural type. We introduce the notation $\Delta \mapsto^{ty} \sigma$ to mean the syntactic translation of Δ to σ; the notation $\Delta \mapsto^{rg}$ `I` to mean the syntactic translation of Δ to range value `I`; and the subset relation over two member typing contexts, $\Delta_1 \subseteq \Delta_2$, which means $\forall \mathtt{l} \in dom(\Delta_1).\ \Delta_1(\mathtt{l}) = \delta_1 \wedge \Delta_2(\mathtt{l}) = \delta_2 \wedge \delta_1 = \delta_2$;

Figure 5.4 presents the rules of computing member typing contexts (Δ) for signatures, method, and meta-level structures that contain a set of members. Computing member typing contexts is performed under a typing environment Γ, which binds the name variables and the type variables appearing in Δ. Computing member typing contexts is necessary to obtain the structural type of a class-like structure, such as group, trait, etc. It flattens the composition hierarchy by using the symmetric sum, which enables the compiler to detect name conflicts among statically declared members. It is used in both type-checking and meta-evaluation. Next, we explain some of the rules in Figure 5.4. For computing the member typing context of a body. (i.e. $\Gamma \vdash$ `D` $\hookrightarrow \Delta$), the type system does not count the members inside patterns for two reasons. First, in most cases we cannot statically check whether the members inside a pattern will be generated. Second, the members inside a pattern are mostly method/field templates parameterized over types and names and they are not for direct use but for generating field/method instances. The rule for interface satisfaction assertion (`E satisfies` $\{\ \overline{\mathtt{H}}\ \}$) shows this language feature is not for information hiding as we can still access the members from `E`. For pattern (`pattern Pt<`$\overline{\mathtt{X}}$`;`$\overline{\eta}$`> P in R D`), it creates a condition for the code inside its body, that is pattern member `P` is a local member and can be accessed via the **this** variable. So we put the typing of `P` into the context. For a trait, the type system computes the

member typing contexts for its required members and provided members respectively. For a meta class, the type system computes the typing contexts for its own defined members and its inherited members respectively; and when flattening the member types of a base class, the rule does not perform any check for method overriding validity, instead it is performed with the typing rules.

The subtyping rules for PTFJ are the same as the subtyping rules for FJ. For readers' convenience, we show them in Figure 5.5. The subtype relation is reflexive and transitive. We could introduce the subtyping over two method types for checking the method overriding that supports covariant return types. However, to be consistent with FJ, PTFJ does not support covariant overriding, so overriding methods have the same return type as the methods they override. Therefore, we do not need to define the subtyping relations over two method types.

In Figure 5.6, we define some auxiliary functions to reveal the structural information from a nominal type (except the `thistype` variable). Function `objType` converts a nominal type into an object type and function `objTypes` converts a sequence of nominal types into a sequence of object types. A type variable's supertype is `Object` which has no members, so the structural type of a type variable is an empty set. To compute the structural type of a class is to flatten the class hierarchy and to merge the members in the hierarchy via the asymmetric sum.

After presenting the preliminary definitions, we come to the main part of static type-checking. Figure 5.7 shows the rules for well-formed types, well-formed names, well-formed member patterns, and well-formed member signatures. Like FJ, we abbreviate a sequence of judgements in our typing rules, such as $\Gamma \vdash \overline{\mathtt{T}}\ \mathtt{ok}$ for $\Gamma \vdash \mathtt{T}_1\ \mathtt{ok}, ..., \Gamma \vdash \mathtt{T}_n\ \mathtt{ok}$.

Typing rules for term expressions, trait expressions, ranges, and method definitions appear in Figure 5.8. Judgement $\Gamma \vdash \mathtt{e} : \tau$ checks if expression `e` is well-typed in context Γ. FJ defines type lookup functions to find the type of a member. Instead, PTFJ type system lookups a member typing from the structural part of an object type (see rule T-FdE and rule T-MdE). The rest of the typing rules for term expressions, such as object creation and type casts, are similar to those in FJ.

Judgement $\Theta; \Delta; \Gamma \vdash \mathtt{E}$ ok `in` `N` type-checks trait expression `E`. The current member typing

Figure 5.4: The rules of computing member typing contexts and for checking member name conflicts.

Generating member typing context for signatures: $\boxed{\Gamma \vdash \texttt{P} \hookrightarrow \Delta}$, $\boxed{\Gamma \vdash \overline{\texttt{P}} \hookrightarrow \Delta}$, and $\boxed{\Gamma \vdash \texttt{I} \hookrightarrow \Delta}$

$$\Gamma \vdash \texttt{T f} \hookrightarrow \texttt{f} : \texttt{T} \qquad \Gamma \vdash \texttt{T m}(\overline{\texttt{T}}) \hookrightarrow \texttt{m} : \overline{\texttt{T}} \to \texttt{T}$$

$$\frac{\overline{\texttt{P}} = \texttt{P}_1 ... \texttt{P}_n \qquad \forall i \in 1...n.\ \Gamma \vdash \texttt{P}_i \hookrightarrow \Delta_i}{\Gamma \vdash \overline{\texttt{P}} \hookrightarrow \bigoplus_{i=1}^{n} \Delta_i} \qquad \frac{\Gamma \vdash \overline{\texttt{F}} \hookrightarrow \Delta_F \qquad \Gamma \vdash \overline{\texttt{H}} \hookrightarrow \Delta_H}{\Gamma \vdash \{\ \overline{\texttt{F}};\ \overline{\texttt{H}}\ \} \hookrightarrow \Delta_F \oplus \Delta_H}$$

Generating member typing context for methods: $\boxed{\Gamma \vdash \texttt{M} \hookrightarrow \Delta}$ and $\boxed{\Gamma \vdash \overline{\texttt{M}} \hookrightarrow \Delta}$

$$\Gamma \vdash \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\ \texttt{return e;}\ \} \hookrightarrow \texttt{m} : \overline{\texttt{T}} \to \texttt{T} \qquad \frac{\overline{\texttt{M}} = \texttt{M}_1 ... \texttt{M}_n \qquad \forall i \in 1...n.\ \Gamma \vdash \texttt{M}_i \hookrightarrow \Delta_i}{\Gamma \vdash \overline{\texttt{M}} \hookrightarrow \bigoplus_{i=1}^{n} \Delta_i}$$

Generating member typing context for body: $\boxed{\Gamma \vdash \texttt{D} \hookrightarrow \Delta}$

$$\frac{\Gamma \vdash \overline{\texttt{M}} \hookrightarrow \Delta_M \qquad \Gamma \vdash \overline{\texttt{GP}} \hookrightarrow \Delta_G \qquad \Gamma \vdash \overline{\texttt{E}} \hookrightarrow \Delta_E}{\Gamma \vdash \{\ \overline{\texttt{M}};\ \overline{\texttt{GP}};\ \overline{\texttt{PT}};\ \texttt{use}\ \overline{\texttt{E}}\ \} \hookrightarrow \Delta_M \oplus \Delta_G \oplus \Delta_E}$$

Generating member typing context for groups: $\boxed{\Gamma \vdash \texttt{GP} \hookrightarrow \Delta}$ and $\boxed{\Gamma \vdash \overline{\texttt{GP}} \hookrightarrow \Delta}$

$$\frac{\Gamma \vdash \texttt{D} \hookrightarrow \Delta}{\Gamma \vdash \texttt{group G D} \hookrightarrow \Delta} \qquad \frac{\overline{\texttt{GP}} = \texttt{GP}_1 ... \texttt{GP}_n \qquad \forall i \in 1...n.\ \Gamma \vdash \texttt{GP}_i \hookrightarrow \Delta_i}{\Gamma \vdash \overline{\texttt{GP}} \hookrightarrow \bigoplus_{i=1}^{n} \Delta_i}$$

Generating member typing context for trait applications: $\boxed{\Gamma \vdash \texttt{E} \hookrightarrow \Delta}$ and $\boxed{\Gamma \vdash \overline{\texttt{E}} \hookrightarrow \Delta}$

$$\frac{\begin{array}{c}\mathrm{TTT}(\texttt{Tr}) = \texttt{TR} \\ \texttt{TR} \hookrightarrow \Delta_{req} \Rightarrow \Delta_{prov}\end{array}}{\Gamma \vdash \texttt{Tr<}\overline{\texttt{R}}\texttt{>} \hookrightarrow \Delta_{prov}} \qquad \frac{\Gamma \vdash \overline{\texttt{H}} \hookrightarrow \Delta_H \qquad \Gamma \vdash \texttt{E} \hookrightarrow \Delta_E}{\Gamma \vdash \texttt{E satisfies}\ \{\ \overline{\texttt{H}}\ \} \hookrightarrow \Delta_H \ni \Delta_E} \qquad \frac{\begin{array}{c}\overline{\texttt{E}} = \texttt{E}_1 ... \texttt{E}_n \\ \forall i \in 1...n.\ \Gamma \vdash \texttt{E}_i \hookrightarrow \Delta_i\end{array}}{\Gamma \vdash \overline{\texttt{E}} \hookrightarrow \bigoplus_{i=1}^{n} \Delta_i}$$

Generating member typing context for pattern: $\boxed{\Gamma \vdash \texttt{PT} \hookrightarrow \Delta}$

$$\frac{\Gamma, \overline{\texttt{X}}, \overline{\eta} \vdash \texttt{D} \hookrightarrow \Delta_D \qquad \Gamma, \overline{\texttt{X}}, \overline{\eta} \vdash \texttt{P} \hookrightarrow \Delta_P}{\Gamma \vdash \texttt{pattern Pt<}\overline{\texttt{X}}\texttt{;}\overline{\eta}\texttt{> P in R D} \hookrightarrow \Delta_D \oplus \Delta_P}$$

Generating member typing context for trait: $\boxed{\texttt{TR} \hookrightarrow \Delta \Rightarrow \Delta}$

$$\frac{\emptyset \vdash \texttt{I} \hookrightarrow \Delta_{req} \qquad \emptyset \vdash \texttt{D} \hookrightarrow \Delta_{prov}}{\texttt{trait Tr<}\overline{\chi}\texttt{> req I prov D} \hookrightarrow \Delta_{req} \Rightarrow \Delta_{prov}}$$

Generating member typing context for meta class: $\boxed{\texttt{MC} \hookrightarrow \Delta \lhd \Delta}$

$$\texttt{Object} \hookrightarrow \cdot \lhd \cdot \qquad \frac{\begin{array}{c}\emptyset \vdash \overline{\texttt{F}} \hookrightarrow \Delta_F \qquad \emptyset \vdash \overline{\texttt{M}} \hookrightarrow \Delta_M \qquad \emptyset \vdash \overline{\texttt{E}} \hookrightarrow \Delta_E \\ \Delta_{sub} = \Delta_F \oplus \Delta_M \oplus \Delta_E \qquad \mathrm{MCT}(\texttt{B}) \hookrightarrow \Delta_{base} \lhd \Delta_{bbase}\end{array}}{\texttt{class C extends B}\ \{\ \overline{\texttt{F}};\ \texttt{K}\ \overline{\texttt{M}}\ \}\ \texttt{uses}\ \overline{\texttt{E}} \hookrightarrow \Delta_{sub} \lhd (\Delta_{base} \ni \Delta_{bbase})}$$

Figure 5.5: The subtyping rules for nominal types

$$\mathtt{N} <: \mathtt{N} \qquad \frac{\mathtt{N}_1 <: \mathtt{N}_2 \quad \mathtt{N}_2 <: \mathtt{N}_3}{\mathtt{N}_1 <: \mathtt{N}_3} \qquad \frac{\texttt{class C extends B } \{...\}}{\mathtt{C} <: \mathtt{B}}$$

Figure 5.6: The auxiliary function for creating object types from nominal types

Type conversion (auxiliary functions): $\boxed{\mathit{objType}(\mathtt{T}) = \tau}$ and $\boxed{\mathit{objTypes}(\overline{\mathtt{T}}) = \overline{\tau}}$

$$\mathit{objType}(\mathtt{X}) = \mathtt{X} \diamond \{\} \qquad \frac{\mathrm{MCT}(\mathtt{C}) \hookrightarrow \Delta_{sub} \triangleleft \Delta_{base} \quad (\Delta_{sub} \ni \Delta_{base}) \mapsto^{ty} \sigma}{\mathit{objType}(\mathtt{C}) = \mathtt{C} \diamond \sigma}$$

$$\frac{\overline{\mathtt{T}} = \mathtt{T}_1, ..., \mathtt{T}_n \quad \forall i \in 1...n.\ \mathit{objType}(\mathtt{T}_i) = \tau_i \quad \overline{\tau} = \tau_1, ..., \tau_n}{\mathit{objTypes}(\overline{\mathtt{T}}) = \overline{\tau}}$$

context Δ indicates the set of members accessible by the **this** variable. The notation `in N` indicates where `E` resides: if `N` is a class name, then `E` is inside a class and if `N` is the `thistype` variable, then `E` is inside a trait/pattern. Rule T-Trp type-checks a trait application. It needs to check if Δ provides the members required by the trait. Because context Δ also includes the members in the used trait, the rule accepts the case when the trait by itself provides its required members. For example, the following is the partial definition of trait DataManaging.

```
1  trait DataManaging<>
2  requires { void sort(List l); }
3  provides {
4      void sort(List l) {
5          // the default implementation is bubble sort
6          ...
7      }
8      public dataProcess(List l) {
9          ...; this.sort(l); ...
10     }
11 }
```

The trait has a default implementation of the sort function. A user is expected to provide a more efficient implementation but it is optional. So we can use DataManaging without giving our own sort function, and the program is well-typed as in the following.

```
class DataManager { use DataManaging<>; }
```

Alternatively, we can give our own sort function, like the following program. Because the program excludes the default sort function, there is no name conflict.

```
class DataManager {
    void sort(List l) {
        // a more efficient implementation using merge sort
        ...
    }
```

```
    use DataManager<>[excludes sort]
}
```

Rule T-SAT type-checks an interface satisfaction assertion: if trait expression `E` provides a method that is also specified in the interface, then the method should have the same type as specified in the interface.

Judgement $\Theta; \Gamma \vdash$ `R` ok `in` `N` checks if a range is valid. Rule T-CNm shows that class name `C` is a valid range if `C` is a supertype of `N`. It means (1) `N` cannot be the `thistype` variable; in other words, `C` is not a valid range if it is inside a trait; and (2) `C` must be a base class of current class `N` or `N` itself. Rule T-Sig checks if a set of member signatures is a valid range. In general, a valid range represents a set of members that can be accessible by the **this** variable. So for a set of member signatures, the type system can check if those members are contained in the current member typing context.

Rule T-Md1 type-checks a method defined inside a class. Rule T-Md2 type-checks a method defined inside a trait or a pattern. They use the *objTypes* function to compute the object types for its argument types; and context Δ to construct the object type for the thistype variable. Context Δ are the typing bindings for all the members that are currently accessible via the **this** variable. It models the structural type of the thistype variable.

Figure 5.9 shows the rules for typing meta-level declarations, including pattern, group, trait, and meta class. Rule T-By type-checks a trait/pattern/group body. It uses the *gpNames* function which simply collects the identifiers for a sequence of groups. Please note the function does not collect the names of the groups nested inside. Groups do not type-check in context Θ, which means group names cannot be used inside group bodies. This avoids the infinite recursion by using group names mutually or by using itself. The following is a partial program that shows the mutual recursion case, which is rejected by our type system.

```
1 trait T1<group X> provides {
2     pattern P<T, name f> public T f in X { ... }
3 }
```

Figure 5.7: The rules for judging if types, names, member patterns, and member signatures are well-formed.

Well-formed types: $\boxed{\Gamma \vdash \texttt{T}\ \text{ok}}$

$$\text{(T-TVar)}\frac{\texttt{X} \in \Gamma}{\Gamma \vdash \texttt{X}\ \text{ok}} \qquad \text{(T-CT)}\frac{\texttt{C} \in \mathit{dom}(\text{MCT})}{\Gamma \vdash \texttt{C}\ \text{ok}}$$

Well-formed names: $\boxed{\Gamma \vdash \texttt{l}\ \text{ok}}$

$$\text{(T-NVar)}\frac{\eta \in \Gamma}{\Gamma \vdash \eta\ \text{ok}} \qquad \text{(T-Name)}\frac{}{\Gamma \vdash \texttt{c}\ \text{ok}} \qquad \text{(T-NCon)}\frac{\Gamma \vdash \texttt{s}_1\ \text{ok} \quad \Gamma \vdash \texttt{s}_2\ \text{ok}}{\Gamma \vdash \texttt{s}_1\#\texttt{s}_2\ \text{ok}}$$

Well-formed member patterns: $\boxed{\Gamma \vdash \texttt{P}\ \text{ok}}$

$$\text{(T-FdT)}\frac{\Gamma \vdash \texttt{T}\ \text{ok} \quad \Gamma \vdash \texttt{f}\ \text{ok}}{\Gamma \vdash \texttt{T f}\ \text{ok}} \qquad \text{(T-MdT)}\frac{\Gamma \vdash \overline{\texttt{T}}, \texttt{T}\ \text{ok} \quad \Gamma \vdash \texttt{m}\ \text{ok}}{\Gamma \vdash \texttt{T m(}\overline{\texttt{T}}\texttt{)}\ \text{ok}}$$

Well-formed member signatures: $\boxed{\Gamma \vdash \texttt{I}\ \text{ok}}$

$$\text{(T-SG)}\frac{\Gamma \vdash \overline{\texttt{F}}\ \text{ok} \quad \Gamma \vdash \overline{\texttt{H}}\ \text{ok}}{\Gamma \vdash \{\ \overline{\texttt{F}};\ \overline{\texttt{H}}\ \}\ \text{ok}}$$

Figure 5.8: The rules for typing term expressions, trait expressions, ranges, and method definition

---

Well-typed expressions: $\boxed{\Gamma \vdash \mathtt{e} : \tau}$

$$\text{(T-Var)}\frac{\mathtt{x} : \tau \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau} \qquad \text{(T-FdE)}\frac{\Gamma \vdash \mathtt{e} : \mathtt{N} \diamond \sigma \quad \mathtt{f} = \mathtt{T} \in \sigma}{\Gamma \vdash \mathtt{e.f} : \mathit{objType}(\mathtt{T})}$$

$$\text{(T-MdE)}\frac{\Gamma \vdash \mathtt{e} : \mathtt{N} \diamond \sigma \quad \mathtt{m} = \overline{\mathtt{T}} \to \mathtt{T} \in \sigma \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{N} \diamond \sigma} \quad \overline{\mathtt{N}} <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{e.m(\overline{e})} : \mathit{objType}(\mathtt{T})}$$

$$\text{(T-New)}\frac{\Gamma \vdash \mathtt{C}\ \text{ok} \quad \mathrm{MCT}(\mathtt{C}) = \mathtt{MC} \quad \mathtt{C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}})\{\ldots\} \in \mathtt{MC} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{N} \diamond \sigma} \quad \overline{\mathtt{N}} <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{new\ C(\overline{e})} : \mathit{objType}(\mathtt{C})}$$

$$\text{(T-Cast)}\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{N} \diamond \sigma \\ \Gamma \vdash \mathtt{T}\ \text{ok} \quad \mathtt{N} <: \mathtt{T} \vee \mathtt{T} <: \mathtt{N}\end{array}}{\Gamma \vdash \mathtt{(T)e} : \mathit{objType}(\mathtt{T})} \qquad \text{(T-SCast)}\frac{\begin{array}{cc}\Gamma \vdash \mathtt{e} : \mathtt{N} \diamond \sigma & \Gamma \vdash \mathtt{T}\ \text{ok} \\ \mathtt{T} \not<: \mathtt{N} \quad \mathtt{N} \not<: \mathtt{T} & \textit{stupid warning}\end{array}}{\Gamma \vdash \mathtt{(T)e} : \mathit{objType}(\mathtt{T})}$$

Well-typed trait applications: $\boxed{\Theta; \Delta; \Gamma \vdash \mathtt{E}\ \text{ok in}\ \mathtt{N}}$

$$\text{(T-TrP)}\frac{\begin{array}{c}\Theta; \Delta \vdash \overline{\mathtt{R}}\ \text{ok in}\ \mathtt{N} \quad \mathrm{TTT}(\mathtt{Tr}) = \mathtt{trait\ Tr<}\overline{\chi}\mathtt{>}\ \ldots \quad |\overline{\chi}| = |\overline{\mathtt{R}}| \\ \mathrm{TTT}(\mathtt{Tr}) \hookrightarrow \Delta_{req} \Rightarrow \Delta_{prov} \quad \forall \mathtt{l} \in \mathit{dom}(\Delta_{req}).\ \Delta_{req}(\mathtt{l}) = \Delta(\mathtt{l})\end{array}}{\Theta; \Delta; \Gamma \vdash \mathtt{Tr<\overline{R}>}\ \text{ok in}\ \mathtt{N}}$$

$$\text{(T-Sat)}\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash \mathtt{E}\ \text{ok in}\ \mathtt{N} \quad \Gamma \vdash \overline{\mathtt{H}}\ \text{ok} \quad \Gamma \vdash \overline{\mathtt{H}} \hookrightarrow \Delta_H \quad \Gamma \vdash \mathtt{E} \hookrightarrow \Delta_E \\ \forall \mathtt{m} \in (\mathit{dom}(\Delta_H) \cap \mathit{dom}(\Delta_E)).\ \Delta_H(\mathtt{m}) = \Delta_E(\mathtt{m})\end{array}}{\Theta; \Delta; \Gamma \vdash \mathtt{E\ satisfies\ \{\ \overline{H}\ \}}\ \text{ok in}\ \mathtt{N}}$$

Valid ranges: $\boxed{\Theta; \Delta; \Gamma \vdash \mathtt{R}\ \text{ok in}\ \mathtt{N}}$

$$\text{(T-GVar)}\frac{\chi \in \Gamma}{\Theta; \Delta; \Gamma \vdash \chi\ \text{ok in}\ \mathtt{N}} \qquad \text{(T-GNm)}\frac{\mathtt{G} \in \Theta}{\Theta; \Delta; \Gamma \vdash \mathtt{G}\ \text{ok in}\ \mathtt{N}}$$

$$\text{(T-CNm)}\frac{\Gamma \vdash \mathtt{C}\ \text{ok} \quad \mathtt{N} <: \mathtt{C}}{\Theta; \Delta; \Gamma \vdash \mathtt{C}\ \text{ok in}\ \mathtt{N}} \qquad \text{(T-Sig)}\frac{\Gamma \vdash \mathtt{I}\ \text{ok} \quad \Gamma \vdash \mathtt{I} \hookrightarrow \Delta_I \quad \Delta_I \subseteq \Delta}{\Theta; \Delta; \Gamma \vdash \mathtt{I}\ \text{ok in}\ \mathtt{N}}$$

Well-typed method: $\boxed{\Delta; \Gamma \vdash \mathtt{M}\ \text{ok in}\ \mathtt{N}}$

$$\text{(T-Md1)}\frac{\begin{array}{ccc}\Gamma \vdash \overline{\mathtt{T}}, \mathtt{T}\ \text{ok} & \Gamma \vdash \mathtt{m}\ \text{ok} & \Delta \mapsto^{ty} \sigma \quad \mathit{objTypes}(\overline{\mathtt{T}}) = \overline{\tau} \\ \Gamma, \overline{\mathtt{x}} : \overline{\tau}, \mathtt{this} : \mathtt{C} \diamond \sigma \vdash \mathtt{e} : \mathtt{N} \diamond \sigma' & & \mathtt{N} <: \mathtt{T}\end{array}}{\Delta; \Gamma \vdash \mathtt{T\ m(\overline{T}\ \overline{x})\{\ return\ e;\ \}}\ \text{ok in}\ \mathtt{C}}$$

$$\text{(T-Md2)}\frac{\begin{array}{ccc}\Gamma \vdash \overline{\mathtt{T}}, \mathtt{T}\ \text{ok} & \Gamma \vdash \mathtt{m}\ \text{ok} & \Delta \mapsto^{ty} \sigma \quad \mathit{objTypes}(\overline{\mathtt{T}}) = \overline{\tau} \\ \Gamma, \overline{\mathtt{x}} : \overline{\tau}, \mathtt{thisType}, \mathtt{this} : \mathtt{thisType} \diamond \sigma \vdash \mathtt{e} : \mathtt{N} \diamond \sigma' & & \mathtt{N} <: \mathtt{T}\end{array}}{\Delta; \Gamma \vdash \mathtt{T\ m(\overline{T}\ \overline{x})\{\ return\ e;\ \}}\ \text{ok in}\ \mathtt{thisType}}$$

```
4  trait T2<group Y> provides {
5      pattern P<T, name f> public T f in Y { ... }
6  }
7  trait T3<> provides {
8      group G1 { use T1<G2>; }
9      group G2 { use T2<G1>; }
10 }
```

Rule T-Ptn type-checks a pattern declaration. It first needs to compute the member typing context (i.e. local context $\Delta_{pattern}$) for the pattern. As shown from Figure 5.4, the member typing contexts for patterns are not computed in an outer scope. So, the outer context Δ_{outer} does not include the pattern's members. So, when the type system checks the body, the current member typing context should be the symmetric sum of the outer context and the local pattern context: $\Delta_{outer} \oplus \Delta_{pattern}$. When type-checking a group declaration (see rule T-Grp), the type system does not need to compute the local member typing context for the group because it was already computed in an outer scope. Rule T-Tr type-checks a trait declaration. A trait is at a program's top level so that the type system begins type-checking without a context. When type-checking the trait's body, required members are treated as provided, and the typings of required members are merged into the current member typing context by the asymmetric sum. Because a member can be both provided and required by a trait, the rule uses the asymmetric sum to choose the provided one. Rule T-Mc type-checks a meta class. In the rule, $\texttt{B(}\overline{\texttt{D}}\ \overline{\texttt{g}}\texttt{)\{...\}} \in \text{MCT}(\texttt{B})$ means constructor $\texttt{B(}\overline{\texttt{D}}\ \overline{\texttt{g}}\texttt{)\{...\}}$ is defined in class B. Besides typing the class body and the trait expressions, the type system also checks the validity of method overriding. As we already mentioned, Featherweight Java (without generics) does not use covariant overriding on the method's return types. So, overriding methods have the same return type as the methods they override.

Figure 5.9: The rules for typing body, pattern, groups, and meta class.

Well-typed body: $\boxed{\Delta;\Gamma \vdash \texttt{D ok in N}}$

$$\text{(T-Bdy)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash \overline{\texttt{M}}\ \text{ok in N} \qquad \Delta;\Gamma \vdash \overline{\texttt{GP}}\ \text{ok in N} \\ \mathit{gpNames}(\overline{\texttt{GP}}) = \Theta \quad \Theta;\Delta;\Gamma \vdash \overline{\texttt{PT}}\ \text{ok in N} \quad \Theta;\Delta;\Gamma \vdash \overline{\texttt{E}}\ \text{ok in N}\end{array}}{\Delta;\Gamma \vdash \{\ \overline{\texttt{M}};\ \overline{\texttt{GP}};\ \overline{\texttt{PT}};\ \texttt{use}\ \overline{\texttt{E}}\ \}\ \text{ok in N}}$$

Well-typed pattern declaration: $\boxed{\Theta;\Delta;\Gamma \vdash \texttt{PT ok in N}}$

$$\text{(T-Ptn)}\ \frac{\begin{array}{c}\Gamma \vdash \texttt{pattern Pt<}\overline{\texttt{X}}\texttt{;}\overline{\eta}\texttt{> P in R D} \hookrightarrow \Delta_{pattern} \\ \Theta;\Delta_{outer};\Gamma \vdash \texttt{R ok in N} \quad \Gamma,\overline{\texttt{X}},\overline{\eta} \vdash \texttt{P} \quad (\Delta_{outer} \oplus \Delta_{pattern});\Gamma,\overline{\texttt{X}},\overline{\eta} \vdash \texttt{D ok in N}\end{array}}{\Theta;\Delta_{outer};\Gamma \vdash \texttt{pattern Pt<}\overline{\texttt{X}}\texttt{;}\overline{\eta}\texttt{> P in R D ok in N}}$$

Well-typed group: $\boxed{\Delta;\Gamma \vdash \texttt{GP ok in N}}$

$$\text{(T-Grp)}\ \frac{\Delta;\Gamma \vdash \texttt{D ok in N}}{\Delta;\Gamma \vdash \texttt{group G D ok in N}}$$

Well-typed trait: $\boxed{\texttt{TR ok}}$

$$\text{(T-Tr)}\ \frac{\begin{array}{c}\texttt{trait Tr<}\overline{\chi}\texttt{> req }\{\ \overline{\texttt{F}};\ \overline{\texttt{H}}\ \}\ \texttt{prov D} \hookrightarrow \Delta_{req} \Rightarrow \Delta_{prov} \\ \emptyset \vdash \overline{\texttt{F}},\overline{\texttt{H}}\ \text{ok} \qquad (\Delta_{prov} \ni \Delta_{req});\overline{\chi} \vdash \texttt{D ok in thistype}\end{array}}{\texttt{trait Tr<}\overline{\chi}\texttt{> req }\{\ \overline{\texttt{F}};\ \overline{\texttt{H}}\ \}\ \texttt{prov D ok}}$$

Well-typed meta class: $\boxed{\texttt{MC ok}}$

$$\text{(T-Mc)}\ \frac{\begin{array}{c}\texttt{K}^o\ \texttt{= C(}\overline{\texttt{B}}\ \overline{\texttt{g}}\texttt{, }\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;\}} \quad \texttt{B(}\overline{\texttt{B}}\ \overline{\texttt{g}}\texttt{)\{...\}} \in \mathrm{MCT}(\texttt{B}) \\ \texttt{class C extends B }\{\ \overline{\texttt{F}};\ \texttt{K}\ \overline{\texttt{M}}\ \}\ \texttt{uses}\ \overline{\texttt{E}} \hookrightarrow \Delta_{sub} \triangleleft \Delta_{base} \\ (\Delta_{sub} \ni \Delta_{base});\emptyset \vdash \{\ \overline{\texttt{M}};\ [];\ [];\ \overline{\texttt{E}}\ \}\ \text{ok in C} \\ \forall \texttt{m} \in (\mathit{dom}(\Delta_{sub}) \cap \mathit{dom}(\Delta_{base}))\ .\ \Delta_{base}(\texttt{m}) = \Delta_{sub}(\texttt{m})\end{array}}{\texttt{class C extends B }\{\ \overline{\texttt{F}};\ \texttt{K}^o\ \overline{\texttt{M}}\ \}\ \texttt{uses}\ \overline{\texttt{E}}\ \text{ok}}$$

5.4.4 Meta-evaluation

Like basic traits, PTFJ meta-evaluation performs trait flattening [42], which flattens the trait composition hierarchy and inlines trait members into classes. Also, PTFJ meta-evaluation performs group range evaluation, pattern matching, method generation, and the following dynamic checking: name conflict checking, interface satisfaction assertion checking, and method overriding validity checking. PTFJ meta-evaluation is the translation of PTFJ programs to FJ programs.

Before discussing the evaluation rules, we first give some definitions. We define syntax $\texttt{H}^o$ as object-level method signature $\texttt{C m}^o\texttt{(}\overline{\texttt{C}}\texttt{)}$; and syntax $\texttt{I}^o$ as object-level member signature sequence $\{\ \overline{\texttt{F}^o\texttt{;}}\ \overline{\texttt{H}^o}\ \}$. We define group mapping context Σ as the mapping from group names to object-level member signature sequences. Two group mapping contexts can be combined using the symmetric sum. We define $[\overline{\texttt{I}^o}/\overline{\chi}]\texttt{D}$ as the simultaneous substitution of object-level member signature sequences for group variables in a body; $[\overline{\texttt{C}}/\overline{\texttt{X}}]\texttt{T}$ as the simultaneous substitution of class names for type variables in a type; $[\overline{\texttt{c}}/\overline{\eta}]\texttt{l}$ as the simultaneous substitution of name constants for name variables in a name (i.e. field name or method name); $[\overline{\texttt{C}}/\overline{\texttt{X}}]\texttt{D}$ as the simultaneous substitution of class names for type variables in a body; and $[\overline{\texttt{c}}/\overline{\eta}]\texttt{D}$ as the simultaneous substitution of name constants for name variables in a body.

Big step meta-evaluation rules are shown in Figure 5.10 and Figure 5.11. In Figure 5.10, rule (E-Concat) concatenates two name constants via the string-like concatenation, where we assume $+^n$ as the internal name string concatenation operator. meta-evaluation over type-level structures (see rule E-Fd and E-MSig) is the evaluation of the names appearing in those structures. In rule E-Md, $\texttt{e} \Downarrow \texttt{e}^o$ is the evaluation of expression e. We omit the rules for evaluating expressions as they are also the evaluation of the names appearing in expressions. The notion $\Sigma \vdash \texttt{R} \Downarrow \texttt{I}^o$ means range R is evaluated into object-level member signature sequence $\texttt{I}^o$ in some group mapping context Σ. Rule E-CNm evaluates class name range C, which needs to first evaluate meta class C into a FJ class. Evaluating a class name range may cause an infinite loop when the class name is used as a range inside its own class. Consider the following program.

```
1  trait AddGettersForPublic<group X>
2  provides {
3      pattern P<T, name f> public T @f in X {
4          public T get#f() { this.@f; }
5      }
6  }
7  class C {
8      public int x;
9      use AddGettersForPublic<C|{x}>
10 }
```

In the program, meta class C uses trait AddGettersForPublic to generate the getter method for its public field x. In line 9, AddGettersForPublic is applied to range C with x selected. When evaluating C, the compiler needs to evaluate the range, and then it needs to first evaluate class name C (i.e. eager evaluation of the left part for group member selection and group member exclusion). According to rule textE-CNm, class C is evaluated again, which causes an infinite loop.

In Figure 5.10, rule E-Trp evaluates a trait application and it uses evaluation rule E-Bdy to evaluate the trait body, which is given in Figure 5.11. Rule E-Sat evaluates interface satisfaction assertion `E satisfied { H }`; while judgement $\Delta_H \subseteq \Delta_E$ checks if the members from `E` satisfies the method signatures $\overline{\texttt{H}}$.

Figure 5.11 presents the evaluation rules for meta-level declarations. Rule E-Bdy evaluates the body of a meta declaration and meanwhile checks name conflicts. A body is the place where methods from different scopes are merged into one scope. In rule E-Bdy, the methods defined in the body, the methods from the groups, the methods generated by the patterns; and the methods from the trait uses are concatenated into one sequence of methods. The rule checks if there is any name conflict in the sequence by computing its member typing context. Computing member typing context for a sequence of methods uses the symmetric sum, which means a typing context exists if and only if there are no duplicate names in the sequence.

When evaluating patterns, there could be two sets, six evaluation rules in total. One set of three rules for pattern-matching over fields and the other set of three rules for pattern-matching over methods. Because the two sets of rules are similar, we omit the latter. In Figure 5.11, rule E-Pt-Empty shows the case when the fields in the range are empty, thus returning the empty method sequence. Rule E-Pt-Succ shows the case when the pattern matches the first field in the range. In this case, a set of methods is generated from the body: $[\overline{\mathtt{c}}/\overline{\eta}][\overline{\mathtt{C}}/\overline{\mathtt{X}}]\mathtt{D} \Downarrow \overline{\mathtt{M}^o}_1$. The pattern needs to pattern-match over the remaining fields in the range, which also generates a set of methods, say $\overline{\mathtt{M}^o}_2$. Because $\overline{\mathtt{M}^o}_1$ and $\overline{\mathtt{M}^o}_2$ are joined, the rule needs to check name conflicts. Rule E-Pt-Fail shows the case when the pattern fails to match the first field in the range, then it continues to pattern-match over the remainder.

Rule E-MC evaluates a meta class. It inlines all the methods generated from trait uses. Though method overriding is checked during type-checking, methods generated from patterns and then inlined into the class (either the superclass or the subclass) could violate method overriding validity. So, method overriding validity is re-checked at the site where a FJ class is generated.

In the next section, we present the PTFJ properties and prove its soundness.

5.4.5 Soundness

Our type system has the following properties.

Lemma 1 (Method Typing Equivalence)**.** *Suppose some method* $\mathtt{M}^o$ *is defined or inlined into class* C*. If* $\mathtt{M}^o$ *is well-defined via our type system:* $\Delta; \cdot \vdash \mathtt{M}^o$ ok in C*, then* $\mathtt{M}^o$ *is also well-defined via the type system of FJ.*

Lemma 2 (Type Substitution Preserves Typing)**.**

If $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{e} : \tau$ *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{e} : [\mathtt{T}/\mathtt{X}]\tau$*.*

If $\Delta; \Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{M}$ ok in N *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $[\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{M}$ ok in N*.*

Lemma 3 (Name Substitution Preserves Typing)**.**

Suppose there is no name conflict during name substitution. If $\Gamma, \eta, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma$ *and* $\Gamma \vdash \mathtt{l}$ ok*,*

Figure 5.10: meta-evaluation (Part One). The rules for evaluating names, fields, method signatures, method definitions, group ranges, and trait expressions.

Name evaluation: $\boxed{\texttt{l} \Downarrow \texttt{c}}$

$$\text{(E-Concat)}\frac{c_1 +^n c_2 = c_3}{c_2 \# c_2 \Downarrow c_3} \qquad \text{(E-Const)}\frac{}{c \Downarrow c}$$

Fields evaluation: $\boxed{\texttt{F} \Downarrow \texttt{F}^o}$

$$\text{(E-Fd)}\frac{\texttt{f} \Downarrow \texttt{f}^o}{\texttt{C f} \Downarrow \texttt{C f}^o}$$

Method signatures evaluation: $\boxed{\texttt{H} \Downarrow \texttt{H}^o}$

$$\text{(E-MSig)}\frac{\texttt{m} \Downarrow \texttt{m}^o}{\texttt{C m(}\overline{\texttt{C}}\texttt{)} \Downarrow \texttt{C m}^o\texttt{(}\overline{\texttt{C}}\texttt{)}}$$

Methods evaluation: $\boxed{\texttt{M} \Downarrow \texttt{M}^o}$

$$\text{(E-Md)}\frac{\texttt{m} \Downarrow \texttt{m}^o \quad \texttt{e} \Downarrow \texttt{e}^o}{\texttt{C m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{ return e; \}} \Downarrow \texttt{C m}^o\texttt{(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{ return e}^o\texttt{; \}}}$$

Evaluation rules for expressions ($\texttt{e} \Downarrow \texttt{e}^o$) *are omitted.*

Group ranges evaluation: $\boxed{\Sigma \vdash \texttt{R} \Downarrow \texttt{I}^o}$

$$\text{(E-GNm)}\frac{\Sigma(\texttt{G}) = \texttt{I}^o}{\Sigma \vdash \texttt{G} \Downarrow \texttt{I}^o} \qquad \text{(E-CNm)}\frac{\begin{array}{c}\text{MCT(C)} \Downarrow \texttt{L}^o \\ \texttt{L}^o \hookrightarrow \Delta \quad \Delta \mapsto^{rg} \texttt{I}^o\end{array}}{\Sigma \vdash \texttt{C} \Downarrow \texttt{I}^o} \qquad \text{(E-Sigs)}\frac{\overline{\texttt{F}} \Downarrow \overline{\texttt{F}^o} \quad \overline{\texttt{H}} \Downarrow \overline{\texttt{H}^o}}{\Sigma \vdash \texttt{\{ }\overline{\texttt{F}}\texttt{; }\overline{\texttt{H}}\texttt{ \}} \Downarrow \texttt{\{ }\overline{\texttt{F}^o}\texttt{; }\overline{\texttt{H}^o}\texttt{ \}}}$$

Trait expression evaluation: $\boxed{\Sigma \vdash \texttt{E} \Downarrow \texttt{E}^o}$

$$\text{(E-Trp)}\frac{\Sigma \vdash \overline{\texttt{R}} \Downarrow \overline{\texttt{I}^o} \quad \text{TTT(Tr)} = \texttt{trait Tr<}\overline{\chi}\texttt{> req I prov D} \quad [\overline{\texttt{I}^o}/\overline{\chi}]\texttt{D} \Downarrow \overline{\texttt{M}^o}}{\Sigma \vdash \texttt{Tr<}\overline{\texttt{R}}\texttt{>} \Downarrow \overline{\texttt{M}^o}}$$

$$\text{(E-Sat)}\frac{\Sigma \vdash \texttt{E} \Downarrow \overline{\texttt{M}^o} \quad \emptyset \vdash \overline{\texttt{M}^o} \hookrightarrow \Delta_E \quad \overline{\texttt{H}} \Downarrow \overline{\texttt{H}^o} \quad \emptyset \vdash \overline{\texttt{H}^o} \hookrightarrow \Delta_H \quad \Delta_H \subseteq \Delta_E}{\Sigma \vdash \texttt{E satisfies \{ }\overline{\texttt{H}}\texttt{ \}} \Downarrow \overline{\texttt{M}^o}}$$

Figure 5.11: meta-evaluation (Part Two). The rules for evaluating body, group, pattern, and meta class.

Body evaluation: $\boxed{\texttt{D} \Downarrow \overline{\texttt{M}^o}}$

$$\text{(E-Bdy)}\ \frac{\overline{\texttt{M}} \Downarrow \overline{\texttt{M}^o}_1 \quad \overline{\texttt{GP}} \Downarrow \overline{\texttt{M}^o}_2 \Rightarrow \Sigma \quad \Sigma \vdash \overline{\texttt{PT}} \Downarrow \overline{\texttt{M}^o}_3 \quad \Sigma \vdash \overline{\texttt{E}} \Downarrow \overline{\texttt{M}^o}_4 \quad \overline{\texttt{M}^o} = (\overline{\texttt{M}^o}_1, \overline{\texttt{M}^o}_2, \overline{\texttt{M}^o}_3, \overline{\texttt{M}^o}_4) \quad \emptyset \vdash \overline{\texttt{M}^o} \hookrightarrow \Delta}{\{\ \overline{\texttt{M}};\ \overline{\texttt{GP}};\ \overline{\texttt{PT}};\ \texttt{use}\ \overline{\texttt{E}}\ \} \Downarrow \overline{\texttt{M}^o}}$$

Group evaluation: $\boxed{\texttt{GP} \Downarrow \overline{\texttt{M}^o} \Rightarrow \Sigma}$ and $\boxed{\overline{\texttt{GP}} \Downarrow \overline{\texttt{M}^o} \Rightarrow \Sigma}$

$$\text{(E-Grp)}\ \frac{\texttt{D} \Downarrow \overline{\texttt{M}^o} \quad \emptyset \vdash \overline{\texttt{M}^o} \hookrightarrow \Delta \quad \Delta \mapsto^{gp} \texttt{I}^o}{\texttt{group G D} \Downarrow \overline{\texttt{M}^o} \Rightarrow [\texttt{G} \mapsto \texttt{I}^o]} \qquad \text{(E-Grps)}\ \frac{\overline{\texttt{GP}} = \texttt{GP}_1, ..., \texttt{GP}_n \quad \forall i \in 1...n.\ \texttt{GP}_i \Downarrow \overline{\texttt{M}^o}_i \Rightarrow \Sigma_i}{\overline{\texttt{GP}} \Downarrow (\overline{\texttt{M}^o}_1, ..., \overline{\texttt{M}^o}_n) \Rightarrow \bigoplus_{i=1}^{n} \Sigma_i}$$

Pattern evaluation: $\boxed{\Sigma \vdash \texttt{PT} \Downarrow \overline{\texttt{M}^o}}$

$$\text{(E-Pt-Empty)}\ \frac{\Sigma \vdash \texttt{R} \Downarrow \{\ []\texttt{;}\ \overline{\texttt{H}^o}\ \}}{\Sigma \vdash \texttt{pattern Pt<}\overline{\texttt{X}}\texttt{;}\overline{\eta}\texttt{> T}_p\ \texttt{f}_p\ \texttt{in R D} \Downarrow []}$$

$$\text{(E-Pt-Succ)}\ \frac{\Sigma \vdash \texttt{R} \Downarrow \{\ (\texttt{C f}^o), \overline{\texttt{F}^o}\texttt{;}\ \overline{\texttt{H}^o}\ \} \quad \exists \overline{\texttt{C}}.\ [\overline{\texttt{C}}/\overline{\texttt{X}}]\texttt{T}_p = \texttt{C} \quad \exists \overline{\texttt{c}}.\ [\overline{\texttt{c}}/\overline{\eta}]\texttt{f}_p \Downarrow \texttt{f}_p^o \wedge \texttt{f}^o = \texttt{f}_p^o \quad [\overline{\texttt{c}}/\overline{\eta}][\overline{\texttt{C}}/\overline{\texttt{X}}]\texttt{D} \Downarrow \overline{\texttt{M}^o}_1 \quad \Sigma \vdash \texttt{pattern Pt<}\overline{\texttt{X}}\texttt{;}\overline{\eta}\texttt{> T}_p\ \texttt{f}_p\ \texttt{in}\ \{\ \overline{\texttt{F}^o}\texttt{;}\ \overline{\texttt{H}^o}\ \}\ \texttt{D} \Downarrow \overline{\texttt{M}^o}_2 \quad \overline{\texttt{M}^o} = (\overline{\texttt{M}^o}_1, \overline{\texttt{M}^o}_2) \quad \emptyset \vdash \overline{\texttt{M}^o} \hookrightarrow \Delta}{\Sigma \vdash \texttt{pattern Pt<}\overline{\texttt{X}}\texttt{;}\overline{\eta}\texttt{> T}_p\ \texttt{f}_p\ \texttt{in R D} \Downarrow \overline{\texttt{M}^o}}$$

$$\text{(E-Pt-Fail)}\ \frac{\Sigma \vdash \texttt{R} \Downarrow \{\ (\texttt{C f}^o), \overline{\texttt{F}^o}\texttt{;}\ \overline{\texttt{H}^o}\ \} \quad (\nexists \overline{\texttt{C}}.\ [\overline{\texttt{C}}/\overline{\texttt{X}}]\texttt{T}_p = \texttt{C}) \vee (\nexists \overline{\texttt{c}}.\ [\overline{\texttt{c}}/\overline{\eta}]\texttt{f}_p \Downarrow \texttt{f}_p^o \wedge \texttt{f}^o = \texttt{f}_p^o) \quad \Sigma \vdash \texttt{pattern Pt<}\overline{\texttt{X}}\texttt{;}\overline{\eta}\texttt{> T}_p\ \texttt{f}_p\ \texttt{in}\ \{\ \overline{\texttt{F}^o}\texttt{;}\ \overline{\texttt{H}^o}\ \}\ \texttt{D} \Downarrow \overline{\texttt{M}^o}}{\Sigma \vdash \texttt{pattern Pt<}\overline{\texttt{X}}\texttt{;}\overline{\eta}\texttt{> T}_p\ \texttt{f}_p\ \texttt{in R D} \Downarrow \overline{\texttt{M}^o}}$$

Evaluation rules for pattern-matching over methods are omitted.

Meta class evaluation: $\boxed{\texttt{MC} \Downarrow \texttt{L}^o}$

$$\text{(E-MC)}\ \frac{\{\ \overline{\texttt{M}};\ []\texttt{;}\ []\texttt{;}\ \texttt{use}\ \overline{\texttt{E}}\ \} \Downarrow \overline{\texttt{M}^o} \quad \emptyset \vdash \overline{\texttt{M}^o} \hookrightarrow \Delta_{sub} \quad \text{MCT(B)} \Downarrow \texttt{L}_b^o \quad \texttt{L}_b^o \hookrightarrow \Delta_{base} \quad \forall \texttt{m} \in (dom(\Delta_{sub}) \cap dom(\Delta_{base}))\ .\ \Delta_{base}(\texttt{m}) = \Delta_{sub}(\texttt{m}) \quad \overline{\texttt{F}} \Downarrow \overline{\texttt{F}^o} \quad \texttt{L}^o = \texttt{class C extends B}\ \{\ \overline{\texttt{F}^o}\texttt{;}\ \texttt{K}^o\ \overline{\texttt{M}^o}\ \}}{\texttt{class C extends B}\ \{\ \overline{\texttt{F}}\texttt{;}\ \texttt{K}^o\ \overline{\texttt{M}}\ \}\ \texttt{uses}\ \overline{\texttt{E}} \Downarrow \texttt{L}^o}$$

---

then $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{e} : \mathtt{N} \diamond [\mathtt{l}/\eta]\sigma$.

Suppose name generation/substitution does not create conflicts. If $\Delta; \Gamma, \eta, \Gamma' \vdash \mathtt{M}$ ok in $\mathtt{N}$ *and* $\Gamma \vdash \mathtt{l}$ ok*, then* $[\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{M}$ ok in $\mathtt{N}$.

Lemma 4 (Type Substitution Preserves Trait Application Typing)**.** *If* $\Theta; \Delta; \Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{E}$ ok in $\mathtt{N}$ *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $\Theta; [\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{E}$ ok in $\mathtt{N}$.

Lemma 5 (Name Substitution Preserves Trait Application Typing)**.** *Suppose name substitution does not generate name conflicts. If* $\Theta; \Delta; \Gamma, \eta, \Gamma' \vdash \mathtt{E}$ ok in $\mathtt{N}$ *and* $\Gamma \vdash \mathtt{l}$ ok*, then* $\Theta; [\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{E}$ ok in $\mathtt{N}$.

Lemma 6 (Type Substitution Preserves Pattern Typing)**.**

If $\Delta; \Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{D}$ ok in `thisType` *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $[\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{D}$ ok in `thisType`.

If $\Theta; \Delta; \Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{PT}$ ok in `thisType` *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $\Theta; [\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{PT}$ ok in `thisType`.

Lemma 7 (Name Substitution Preserves Pattern Typing)**.** *Suppose name substitution does not generate name conflicts.*

If $\Delta; \Gamma, \eta, \Gamma' \vdash \mathtt{D}$ ok in `thisType` *and* $\Gamma \vdash \mathtt{l}$ ok*, then* $[\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{D}$ ok in `thisType`.

If $\Theta; \Delta; \Gamma, \eta, \Gamma' \vdash \mathtt{PT}$ ok in `thisType` *and* $\Gamma \vdash \mathtt{l}$ ok*, then* $\Theta; [\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{PT}$ ok in `thisType`.

Lemma 8 (Pattern, Group, and Trait Application Evaluation Preserves Typing)**.**

(1) If $\Delta; \Gamma \vdash \mathtt{D}$ ok in $\mathtt{N}$*, and* $\mathtt{D} \Downarrow \overline{\mathtt{M}^0}$*, then* $\Delta; \Gamma \vdash \overline{\mathtt{M}^o}$ ok in $\mathtt{N}$.

(2) If $\Theta; \Delta; \Gamma \vdash \mathtt{E}$ ok in $\mathtt{N}$*, and* $\Sigma \vdash \mathtt{E} \Downarrow \overline{\mathtt{M}^0}$*, then* $\Delta; \Gamma \vdash \overline{\mathtt{M}^o}$ ok in $\mathtt{N}$.

(3) If $\Delta; \Gamma \vdash \mathtt{GP}$ ok in $\mathtt{N}$*, and* $\mathtt{GP} \Downarrow \overline{\mathtt{M}^0} \Rightarrow \Sigma$*, then* $\Delta; \Gamma \vdash \overline{\mathtt{M}^o}$ ok in $\mathtt{N}$.

(4) If $\Theta; \Delta; \Gamma \vdash \mathtt{PT}$ ok in $\mathtt{N}$*, and* $\Sigma \vdash \mathtt{PT} \Downarrow \overline{\mathtt{M}^0}$*, then* $\Delta; \Gamma \vdash \overline{\mathtt{M}^o}$ ok in $\mathtt{N}$.

Theorem 5 (Meta Class Generates Well-formed FJ Class)**.** *If* $\mathtt{MC}$ ok *and* $\mathtt{MC} \Downarrow \mathtt{L}^o$*, then* $\mathtt{L}^o$ *is well-formed in the FJ type system.*

5.5 Summary

C++ templates support metaprogramming but they cannot iterate and inspect class members. Java supports runtime reflection but it does not support code generation. In this chapter,

we propose pattern-based traits, which provide both reflection and metaprogramming for object-oriented programming languages.

Pattern-based traits are a combination of traits and pattern-based reflection. Though traits and pattern-based reflection are not new, their integration is novel. Further, we provide language features for manipulating sets of member declarations: giving them names, manipulating their domains using set operations, and passing them as arguments to traits. The integration brings up an issue of how to design a type system that guarantees static type safety but also supports language expressiveness. We proposed a type system that guarantees generated code is always well-typed at method/statement level.

In this chapter, we formalize the language design, defining the syntax and semantics for a language named PTFJ (for Pattern-based Traits in Featherweight Java).

Chapter 6

Statement-Level Compile-Time Reflection and Metaprogramming

Java supports runtime reflection at the statement level, which has the ability to inspect and analyze class members (i.e. metadata of a class) at run time. It enables programmers to write reusable programs that are independent of certain classes. However, when runtime performance is a big concern, we propose to use compile-time reflection for writing metaprograms that generate non-reflective class/type specific code, which has lower runtime cost.

In this chapter, we present pattern-based reflection at the statement level. Statement-level pattern-based reflection is not new and some researchers have already proposed it [45, 25]. But to our knowledge, it has never been fully discussed and formalized. We find out that statement-level pattern-based reflection supports writing many useful metaprograms that class or method level pattern-based reflection cannot achieve, thus it is meaningful to propose and formalize a language with statement-level pattern-based reflection and prove that the language always generates well-typed statements.

6.1 Runtime Reflection in Java

Java reflection [21] enables us to inspect and analyze class members (i.e. metadata of a class) at run time. For example, we are given a great number of classes and asked to implement the function hasGetters. The function checks if all the fields in a class have getters (i.e. field get methods) and prints out those fields without getters. This function is useful when class objects need to be serialized (e.g. converting objects into XML data) or object-relational mapped: many

ORM tools require fields to have getters for access. It is impractical to manually write a hasGetters for each class, because each time we change the fields of a class, we also likely need to change the hasGetters for that class, therefore such implementation is not adaptable to the change of a class.

With runtime reflection, we can write a hasGetters that is applicable to any class. Following shows the partial implementation of the hasGetters.

```
1  public boolean hasGetters(Class cls) {
2     Field[] fds = cls.getFields();
3     for(int i = 0; i < fds.length; ++i) {
4        Field fd = fds[i];
5        String mn = "get"+capitalize(fd.getName());
6        Class fTy = fd.getType();
7        try {
8           Method _m = cls.getMethod(mn, new Class[0]);
9           if (! _m.getReturnType().equals(fTy)) {
10             ... /* print the message that field fd has no getter */
11          }
12       } catch (Exception e){
13          ... /* print the message that field fd has no getter */
14       }
15    }
16    ...
17 }
```

The hasGetters iterates over all the fields of a class (lines 2-3). For each field, it creates the name of the getter (line 5, function capitalize converts the first character to upper case) and searches in the class for a method that has the same name and with no parameters (line 8). If such a method is found and its return type is equal to the field's type (line 9), then the method is a getter method.

Though Java runtime reflection enables programmers to write general programs, it has three drawbacks.

First, runtime reflection is not efficient. Suppose a program uses the hasGetters method to check a class that is statically known. Each time we run the program, the function needs to iterate over all the fields and the methods in the class.

Second, Java has no specific language that is dedicated to reflection. It uses objects to store metadata. For ease of use and for expressiveness, we would like to have a meta language, which enables users to inspect and manipulate metadata more easily. For example, the meta language of Garcia's calculus [23] for type-reflective metaprogramming has abundant type-level operators and uses types as terms.

Third, Java reflection has little support for code generation. Forman's book [21] mentioned code generation via reflection, but the object-level code is written in strings, which have no type safety guarantees. We would like Java to have an object-level language that the compiler can type-check before code generation.

6.2 An Introductory Example

To overcome the drawbacks mentioned in the previous section, we present pattern-based reflection at the statement level. Following shows the `hasGetters` implemented in statement-level pattern-based reflection.

```
1  trait getterChecker<class X>
2  provides {
3     public boolean hasGetters() {
4        boolean has = false; boolean ret = true;
5        pattern <F, name f> F @f in X {
6           pattern <> public F get#@f() in X {
7              has = true;
8           }
9           if (! has) {
10             ret = false;
11             println("Field "+@f::getName()+" has no getter!");
```

```
12              } else { has = false; }
13          }
14          return ret;
15      }
16  }
```

Trait getterChecker is parameterized over the class that is to be reflected over. Inside the body of function hasGetters, the outer pattern matches each field in X (line 5) and passes the field's name and the type to the inner pattern, which searches the field's getter method in X (line 6). The symbol # is the name concatenation operator. In line 11, expression @f::getName() is a meta function call that, at compile time, returns the string of field f's name. The hasGetters in the trait can be specialized for any class and also adaptable to the change of a class.

For example, suppose we are given the following class Person:

```
1  class Person {
2      private String full_name;
3      private int age;
4      public int getAge() { return this.age; }
5  }
```

In class Person, field full_name has no getter method. When trait getterChecker is applied to class Person, the trait generates the following hasGetters function:

```
1  public boolean hasGetters() {
2      boolean has = false; boolean ret = true;
3      if (! has) {
4          ret = false;
5          println("Field "+"full_name"+" has no getter!");
6      } else { has = false; }
7      has = true;
8      if (! has) {
9          ret = false;
```

```
10          println("Field "+"age"+" has no getter!");
11      } else { has = false; }
12      return ret;
13  }
```

The generated hasGetters function is specified for class Person and it has no runtime reflection in the body. Please note that in line 7, the value of variable has is changed into **true**. So, the second println is not executed.

6.3 Language Features

In this section, we introduce the language features supporting statement-level pattern-based reflection. We start with an brief overview of pattern-based reflection. Even though it was introduced in the previous chapters, it is worthwhile to review their syntax not only for readers' convenience but also for the modification we have made to their syntax.

6.3.1 Pattern-Based Reflection

A reflective pattern at the statement level resides in a method body. It performs both reflection and code generation. The header part can iterate over a sequence of class members (i.e. fields, methods, and constructors), and pattern-match against each of them. The body contains template code that is parameterized over names and/or types. A reflective pattern generates different code instantiations from different names and types obtained using pattern-matching.

In the following, we give the syntax for reflective patterns at the statement level.

pattern-declaration ::=

pattern

<*parameters*> [*modifier-pattern*] *member-pattern*

in *range*

{ *statements* }

A pattern declaration may have a name, but it is not necessary at the statement level. Pattern

parameters include name parameters and (constrained) type parameters.

A member pattern is in the form of a field signature, a method signature, or a constructor signature. It may be prefixed with an access-level modifier pattern for pattern-matching a group of class members with certain access level(s). An access-level modifier pattern can be **public**, **private**, none for package-private, **nonprivate** for the access levels that are not private, etc.

A range represents a sequence of class members that a pattern can iterate over. It has the following syntax:

$$range ::= identifier \mid range|\{identifiers\} \mid range\backslash\{identifiers\}$$

which can refer to a class type, a class type variable (including the reserved type variable: thisType), a member-selection operation (i.e. selecting specified members from a range), or a member-exclusion operation (i.e. removing specified members from a range). Our paper [39] also proposed the sum of two ranges, which is not allowed here.

The body of a statement-level pattern is a sequence of statements. We do not allow a return statement to appear inside a pattern's body, otherwise a pattern can generate statements containing unreachable code.

As we mentioned in the introduction, statement-level reflective patterns enable us to write general methods that are applicable to different classes and also adaptable to class change. For example, when we design an extensible programming language framework, we have to use the visitor pattern for separating the syntax from its behaviors. Following is the syntax of a tiny language for integers and immutable arrays:

$$E ::= Integer \mid E + E \mid \{\overline{E}\} \mid E[E]$$

In the syntax, expression $\{\overline{E}\}$ represents immutable array definition, which is like a sequence of expressions; and $E[E]$ means array access. The below shows the Java code of the language's abstract syntax tree nodes:

```
1 interface Node { void visit(NodeVisitor v); }
2
```

```
3  class IntLit extends Node {
4      private int; public void visit(NodeVisitor v) { }
5  }
6  class Addition extends Node {
7      private Node left; private Node right;
8      public void visit(NodeVisitor v) {
9          this.left = v.visit(this.left); this.right = v.visit(this.right);
10     }
11 }
12 class Array extends Node {
13     private List fds;
14     public void visit(NodeVisitor v) {
15         List tmp = new ArrayList();
16         for(Iterator i = this.fds.iterator(); i.hasNext(); ) {
17             Object fd = i.next();
18             if (fd instanceof Node)
19                 tmp.add(v.visit((Node) fd));
20         }
21         this.fds = tmp;
22     }
23 }
24 class ArrAccess extends Node {
25     private Node arr; private Node idx;
26     public void visit(NodeVisitor v) {
27         this.arr = v.visit(this.arr); this.idx = v.visit(this.idx);
28     }
29 }
```

In the above code, we assume that abstract class NodeVisitor gives the method:

```
Node visit(Node n) { n.visit(this); return n; }
```

For each syntax tree node, we implement the visit method that visits its sub-nodes. With this visitor pattern, we can easily write a type-checker, an evaluator, and a syntax tree printer for this language (detailed implementations are omitted).

If the above language is fully extended, it will be tedious to manually write a visit method for each syntax node. Therefore, we write the following metaprogram to generate the visit methods.

```
1  trait VisitGen<>
2  provides {
3      public void visit(NodeVisitor v) {
4          pattern<F extends Node, name f> F @f in thisType {
5              this.@f = (F) v.visit(this.@f);
6          }
7          pattern<F extends Collection, name f> F @f in thisType
8          {
9              List tmp = new ArrayList();
10             for(Iterator i = this.@f.iterator(); i.hasNext(); ) {
11                 Object e = i.next();
12                 if (e instanceof Node)
13                     tmp.add(v.visit((Node) e));
14             }
15             this.@f.clear(); this.@f.addAll(tmp);
16         }
17         // insert a reflective pattern for the fields of array types
18         ...
19     }
20 }
```

In trait VisitGen and inside visit, the first pattern (lines 4-6) matches the fields which are syntax tree nodes and generates the statements for them. The second pattern (line 7-16) matches and generates statements for the fields of type Collection. In the body of the second pattern, the code traverses the collection and applies the visit method to an element if it is a syntax tree node. We restrict

the range to the type variable thisType, which will be automatically substituted for the name of the class that uses VisitGen. In the patterns, we have the code (in lines 5, 10, and 15) that may access the private fields via the **this** variable, and the thisType range guarantees the code is well-typed. For instance, when class Addition uses VisitGen, the thisType variable is substituted for Addition; a visit method is generated for Addition; and inside the visit method, it is safe to access Addition's private fields. In section 6.3.3, we give more detailed discussion about type safety.

6.3.2 Reified Generics

To support code compatibility, Java implements generics using type erasure [1] so that the specialized types are not available at runtime. For instance, the following expressions are not accepted in Java (we assume X is a well-defined type variable): X.**class**, obj **instanceof** X, **new** X(), **new** X[10], etc.

Using metaprogramming, we generate specialized code for the instantiation of generics, like C++ templates [2], so that specialized types are preserved in generated code.

For example, in the following, we give a generic equal function at the meta level, which can be instantiated into a specialized equal function for any class.

```
1  trait EqualGen<>
2  provides {
3      public boolean equals(Object obj) {
4          if (obj instanceof thisType)
5              return equals_k(this, (thisType) obj);
6          return false;
7      }
8      private boolean equals_k(thisType obj1, thisType obj2) {
9          boolean is_equal = true;
10         pattern <primitive T, name f> T @f in thisType {
11             if (obj1.@f != obj2.@f)
```

[1] see http://docs.oracle.com/javase/tutorial/java/generics/erasure.html

```
12              is_equal = false;
13          }
14          pattern <T extends Object, name f> T @f in thisType {
15              // code for comparing two objects
16              ...
17          }
18          return is_equal;
19      }
20  }
```

In line 4, the **instanceof** operator is applied to the thisType variable. Inside function equals_k, the first pattern matches and compares the fields of primitive types, while the second pattern matches and compares the fields of reference types. When the trait EqualGen is used by some concrete class C, the variable thisType is specialized into type C.

The following is an example of another use of reified generics: generating instance creation functions for the support of the factory method pattern [2] .

```
1  trait InstanceGen<class X, class S>
2  provides {
3      public S createInstance() throws InstantiationException {
4          X obj = null;
5          pattern <> public constructor() in X {
6              obj = new X();
7          }
8          if (obj instanceof S) {
9              println("An instance of "+X::getName()+" is created.");
10             return (S) obj;
11         }
12         throw new InstantiationException("...");
13     }
```

[2] see http://en.wikipedia.org/wiki/Factory_method_pattern

```
14 }
```

The trait receives two arguments: the type of a class whose instance can be created and the interface that the class implements. In line 5-6, the pattern matches a public nullary (or default) constructor for class X and creates an instance via the **new** operator if the constructor exists. In line 8, we check if the created object is an instance of type variable S. In line 9, we have the meta function call X::getName() that returns the name of X at compile time. Suppose we have class Student, which has the nullary constructor and implements interface Person. When trait InstanceGen is applied to Student and Person, it generates the following result:

```
1  public Person createInstance() throws InstantiationException {
2      Student obj = null;
3      obj = new Student();
4      if (obj instanceof Person) {
5          println("An instance of "+"Student"+" is created.");
6          return (Person) obj;
7      }
8      throw new InstantiationException("...");
9  }
```

Besides the meta function getName, we have other predefined meta functions that enable users to inspect metadata at compile time. For instance, expression X::getSimpleName() returns the simple name of type X; X::equals(Y) checks if X is equal to Y; X::isSubType(Y) checks if X is a subtype of Y; X::superClass() returns the direct superclass of X; X::isPrimitive() checks if X is a primitive type; and so on.

6.3.3 Member Accessilbility

Like pattern-based traits, a trait with statement-level pattern-based reflection also needs to check member accessibility. Suppose a pattern-based trait reflects over the members of some class A and is used by some class B. We have to discuss the relation between A and B because it has

influence over the accessibility of a member. In this section, we discuss the conditions when the members of a class can be accessed via the **this** variable.

Consider the following metaprogram, which generates a function to backup the fields in a superclass.

```
1  trait FieldBackupGen<this-class S>
2  provides {
3      public void backup() {
4          pattern <T, name f> nonprivate T @f in S {
5              pattern <> T backup#@f in thisType {
6                  this.backup#@f = this.@f;
7              }
8          }
9      }
10 }
```

The type variable S in the metaprogram cannot be instantiated with an arbtrary class. For instance, trait FieldBackupGen used as follows generates ill-typed code.

```
1  class Account { protected int balance; }
2  class AccBackup {
3      private int backupBalance;
4      use FieldBackupGen<Account>;
5  }
```

For the above, the body of the instantiated function backup is

```
this.backupBalance = this.balance
```

but field balance cannot be accessed via the **this** variable inside class AccBackup. The correct implementation is to let class AccBackup inherit from Account, so we can access Account's non-private members via the **this** variable.

In the definition of trait FieldBackupGen, we use keyword **this-class** to restrict the set of classes

that a trait can be applied to. That means that an accepted class must be a super type of the class that uses the trait. In detail, if the trait is used by class A, then an accepted class must be a super class of A or class A itself. Suppose some type variable X is prefixed with **this-class**; the subtype relationship between thisType and type variable X is that thisType <: X.

In the above code, trait FieldBackupGen's type parameter S is restricted by the **this-class**. Trait FieldBackupGen is used by class AccBackup and is applied to class Account. However, class Account is not a super type of class AccBackup, thus the trait application is rejected by the type system.

For a unrestricted type variable X, we allow users to only access its public members via an instance of X.

6.4 Calculus for the Object Language

Starting with this section, we formalize our language. First, we present the calculus for our object language, which is the modest extension of FJ (short for Featherweight Java) [30] with mutable variables and three basic kinds of statements: variable declaration, assignment, and return statement.

Figure 6.1: FJ syntax extended with statements.

x, y		term variables
m		method name
f		field name
C		type (i.e. class name)
class decl.	L ::=	class C extends C { $\overline{\texttt{C}}$ $\overline{\texttt{f}}$; K $\overline{\texttt{M}}$ }
constructor	K ::=	C($\overline{\texttt{C}}$ $\overline{\texttt{f}}$){ super($\overline{\texttt{f}}$); this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$; }
method	M ::=	C m($\overline{\texttt{C}}$ $\overline{\texttt{x}}$){ s; }
statements	s ::=	return e | C x = e; s | x = e; s
expressions	e ::=	x | e.f | e.m($\overline{\texttt{e}}$) | new C($\overline{\texttt{e}}$) | (C)e | $\mathbb{E}[\![\texttt{s}]\!]$

Figure 6.1 shows the abstract syntax of the object language. In the figure, the syntax of statements is the addition to FJ. A statement can be a return statement or a sequence of statements with a return statement at the end. The variable declaration `C x = e` declares local variable `x` of

type C and initializes x with the value of expression e. The assignment x = e assigns the value of e to x. In Java, an assignment is also an expression that gives the value of e, but in our object language, we treat an assignment only as a statement. The rest of the syntax in the figure is the same as FJ except for methods and the expression $\mathbb{E}[\![\mathtt{s}]\!]$. In the object language, the body of a method is a sequence. The expression $\mathbb{E}[\![\mathtt{s}]\!]$ does not appear in a concrete program. It wraps a statement and splices it into an expression.

Figure 6.2: Typing rules for statements.

Statement typing: $\boxed{\Gamma \vdash \mathtt{s} : \mathtt{C}}$

$$\frac{\Gamma \vdash \mathtt{x} : \mathtt{C}_1 \quad \Gamma \vdash \mathtt{e} : \mathtt{C}_0 \quad \mathtt{C}_0 <: \mathtt{C}_1 \quad \Gamma \vdash \mathtt{s} : \mathtt{C}}{\Gamma \vdash \mathtt{x = e;\ s} : \mathtt{C}}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C}_0 \quad \mathtt{C}_0 <: \mathtt{C} \quad \Gamma, \mathtt{x} : \mathtt{C} \vdash \mathtt{s} : \mathtt{C}_1}{\Gamma \vdash \mathtt{C\ x = e;\ s} : \mathtt{C}_1} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C}}{\Gamma \vdash \mathtt{return\ e} : \mathtt{C}}$$

Figure 6.2 shows the rules for typing statements. We define Γ as a typing environment, which is a finite mapping from variables to types. An environment extension has the form $\Gamma, \mathtt{x} : \mathtt{C}$, which means extending Γ with variable typing $\mathtt{x} : \mathtt{C}$ only if x does not appear in Γ. The judgments $\mathtt{C}_0 <: \mathtt{C}_1$ means class $\mathtt{C}_0$ is a subtype of class $\mathtt{C}_1$; and $\Gamma \vdash \mathtt{e} : \mathtt{C}$ means expression e has type C under the typing environment Γ. We use the subtyping rules and the typing rules for expressions from FJ's type system (see the figures Fig.1 and Fig.2 in [30]). For the additional expression $\mathbb{E}[\![\mathtt{s}]\!]$, its type should be equal to the type of statement s. See the following typing rule:

$$\frac{\Gamma \vdash \mathtt{s} : \mathtt{C}}{\Gamma \vdash \mathbb{E}[\![\mathtt{s}]\!] : \mathtt{C}}$$

We also use the typing rules from FJ to type-check methods and class declarations. For FJ's method typing rule, we need to override the rule of type-checking a method body: we use the statement typing rule to type-check the method body. Because those rules are straightforward, we do not present them in this thesis.

Figure 6.3 shows the reduction rules for statements. We define v as the value of an expression, which is new C($\overline{\mathtt{v}}$). Similar to [e'/x]e, the operation [e/x]s means the substitution of x in statement

Figure 6.3: Statement reduction rules.

Statement reduction: $\boxed{\texttt{s} \longrightarrow \texttt{s'}}$

$$\frac{}{\texttt{C x = v; s} \longrightarrow [\texttt{v/x}]\texttt{s}} \qquad \frac{}{\texttt{x = v; s} \longrightarrow [\texttt{v/x}]\texttt{s}}$$

$$\frac{\texttt{e} \longrightarrow \texttt{e'}}{\texttt{C x = e; s} \longrightarrow \texttt{C x = e'; s}}$$

$$\frac{\texttt{e} \longrightarrow \texttt{e'}}{\texttt{x = e; s} \longrightarrow \texttt{x = e'; s}} \qquad \frac{\texttt{e} \longrightarrow \texttt{e'}}{\texttt{return e} \longrightarrow \texttt{return e'}}$$

Figure 6.4: Substitution in statements.

[e'/x](return e)	=	return [e'/x]e
[e'/x](C y = e; S)	=	(C y = [e'/x]e; [e'/x]S)
[e'/x](x = e; S)	=	(x = [e'/x]e; S)
[e'/x](y = e; S)	=	(y = [e'/x]e; [e'/x]S) where $\texttt{x} \neq \texttt{y}$

s for e. Figure 6.4 shows its definition. The substitution [e'/x](C x = e; S) does not need to be defined. The substitution implies that x is declared twice in the same scope, which is precluded by the type system. In the figure, we use the reduction rules for expressions (the form $\texttt{e} \longrightarrow \texttt{e'}$) from FJ (see Figure Fig.3 in [30]). But we need to revise the computation rule for method invocation and provide the reduction rules for $\mathbb{E}[\![\texttt{s}]\!]$. Those rules are shown in Figure 6.5. Like FJ's **mbody** function, the **mbody** function in the figure returns the parameter(s) and the body of a method in a class.

Figure 6.5: Selected reduction rules for expressions.

Expression reduction: $\boxed{\texttt{e} \longrightarrow \texttt{e'}}$

$$\frac{\mathbf{mbody}(\texttt{m}, \texttt{C}) = \overline{\texttt{x}}.\texttt{s}}{\texttt{(new C(}\overline{\texttt{e}}\texttt{)).m(}\overline{\texttt{e}_0}\texttt{)} \longrightarrow \mathbb{E}[\![[\overline{\texttt{e}_0}/\overline{\texttt{x}}, \texttt{new C(}\overline{\texttt{e}}\texttt{)/this}]\texttt{s}]\!]}$$

$$\frac{\texttt{s} \longrightarrow \texttt{s'}}{\mathbb{E}[\![\texttt{s}]\!] \longrightarrow \mathbb{E}[\![\texttt{s'}]\!]} \qquad \frac{}{\mathbb{E}[\![\texttt{return v}]\!] \longrightarrow \texttt{v}}$$

... (**The rest of the reduction rules are the same as those for FJ.**)

The calculus of our object language is type safe. It inherits the properties of FJ: type preservation and progress for expressions. Besides, the properties of type preservation and progress also apply to statements. See the following properties for statements.

Lemma 9 (Substitution Preserves Typing)**.** *If* $\Gamma, \texttt{x} : \texttt{C}_0 \vdash \texttt{s} : \texttt{C}$ *and* $\Gamma \vdash \texttt{e} : \texttt{C}_1$ *where* $\texttt{C}_1 <: \texttt{C}_0$*, then* $\Gamma, \texttt{x} : \texttt{C}_0 \vdash [\texttt{e}/\texttt{x}]\texttt{s} : \texttt{C'}$ *for some* C' *such that* $\texttt{C'} <: \texttt{C}$*.*

Please note that, in Lemma 1, when type-checking [e/x]s, we do not remove the typing of x from typing context Γ, because the substitution of x in s does not always substitute all the xs in s (see the third rule in Figure 6.4). So, variable x might be still in [e/x]s.

Theorem 6 (Type Preservation)**.** *If* $\Gamma \vdash \texttt{s} : \texttt{C}$ *and* $\texttt{s} \longrightarrow \texttt{s'}$*, then* $\Gamma \vdash \texttt{s'} : \texttt{C'}$ *for some* C' *such that* $\texttt{C'} <: \texttt{C}$*.*

Theorem 7 (Progress)**.** *If* $\Gamma \vdash \texttt{s} : \texttt{C}$*, then* s *is either a statement value (i.e.* return v*) or there is some* s' *with* $\texttt{s} \longrightarrow \texttt{s'}$*.*

6.5 Calculus for the Meta Language

Our calculus for the meta language captures the new language features and presents the kernel part of the language, thus we omit some features that were fully discussed in the previous papers about traits, such as the manipulation of trait members (e.g. name exclusion, member aliasing, etc). We also omit type bounds, modifier patterns, and meta-level functions, which are considered as advanced features.

6.5.1 Kernel Syntax of the Meta Language

Figure 6.6 shows the syntax of the core meta language. In the figure, a type τ is a class name associated with the signatures of the class's members. It gives both the nominal and the structural representation of a class type, and it is the value of a range. A range variable `X` can be used as a type variable. When the range variable `X` is substituted for some τ in type `T`, `X` should be substituted for the nominal representation of τ, that is the substitution $[\texttt{C} \diamond \{\overline{\texttt{F}}\texttt{;}\texttt{Q}\texttt{;}\overline{\texttt{H}}\}/\texttt{X}]\texttt{T}$ is reduced into $[\texttt{C}/\texttt{X}]\texttt{T}$. But it performs a normal substitution when the range variable `X` is substituted for some τ in range `R`.

In the expressions, we can use the `new` operator to create an instance of a parameterized class, for instance, we can write the expression `new X(`$\overline{\texttt{e}}$`)`.

A trait `TR` is parameterized over ranges. We attach the symbol + or the symbol − to each range parameter. The parameter $\texttt{X}^{+}$ means the members (excluding constructors) of `X` can be accessed via the `this` variable. The parameter $\texttt{X}^{-}$ means the members of `X` can be only accessed via an instance of `X`.

Ranges include member selection ($\texttt{R}|\{\overline{\texttt{l}}\}$) and member exclusion ($\texttt{R}\backslash\{\overline{\texttt{l}}\}$). A range is evaluated into a range value τ, which does not appear in a concrete program. For the range of some class `C`, we compute its value by collecting the signatures of the members in `C`, including `C`'s inherited members.

Figure 6.6: Syntax of the calculus for the meta language.

x, y	term variables
X, Y	type/range variables
η	member name variable
m	method name
f	field name
C	class name
$\mathbb{C}$	reserved name constant
$\mathcal{T}$	trait name
L	object class declaration (see Figure 6.1)

(1) types, names, and member signatures:

type	τ ::= C $\diamond$ {$\overline{\text{F}}$;Q;$\overline{\text{H}}$}
nonvar. types	N ::= C | thisType
nominal types	U, T ::= X | N
member names	l ::= η | f | m
field sig.	F ::= T l
constructor sig.	Q ::= $\mathbb{C}$($\overline{\text{T}}$)
method sig.	H ::= T l($\overline{\text{T}}$)

(2) meta classes:

meta class	MC ::= L uses $\overline{\text{E}}$
meta meth.	M ::= T m($\overline{\text{T}}$ $\overline{\text{x}}$){ s; }
single stmts.	ss ::= T x = e | x = e | ps
statements	s ::= return e | ss; s
expressions	e ::= x | e.l | e.l($\overline{\text{e}}$) | new T($\overline{\text{e}}$) | (T)e

(3) traits:

trait	TR ::= trait $\mathcal{T}$<$\overline{\text{X}^o}$> req { $\overline{\text{F}}$; $\overline{\text{H}}$ } prov D
access ctrl.	o ::= $+$ | $-$
trait body	D ::= { $\overline{\text{M}}$; use $\overline{\text{E}}$ }
trait app.	E ::= $\mathcal{T}$<$\overline{\text{R}}$>

(4) pattern statements:

pattern stmt.	ps ::= pattern<$\overline{\text{X}}$,$\overline{\eta}$> P in R { ns; }
patterns	P ::= F | Q | H
ranges	R ::= X | N | R|{$\overline{\text{l}}$} | R\{$\overline{\text{l}}$} | τ
non-return stmts.	ns ::= ss | ss; ns

6.5.2 Type System

In this section, we discuss the type system of the meta language calculus. First, we give some preliminary definitions (see Figure 6.7) which are used by the type system.

Figure 6.7: Preliminary definitions for the type system.

member types	ϕ	$::= \mathtt{T} \mid \overline{\mathtt{T}} \to \mathtt{T} \mid \overline{\mathtt{T}} \to \cdot$
member names	ℓ	$::= \mathtt{l} \mid \mathbb{C}$
structure names	κ	$::= \mathtt{T} \mid \mathcal{T}$
member typings	Δ	$::= \cdot \mid \Delta[\ell \mapsto \phi] \mid \Delta \oplus \Delta$
structural typings	Θ	$::= \cdot \mid \Theta \uplus [\kappa \mapsto \Delta]$
variable bindings	Γ	$::= \emptyset \mid \Gamma, \mathtt{x} : \mathtt{T} \mid \Gamma, \mathtt{X}^o \mid \Gamma, \mathtt{thisType} \mid \Gamma, \eta$

In the figure, member types present the types for different kinds of class members: $\mathtt{T}$ for fields, $\overline{\mathtt{T}} \to \mathtt{T}$ for methods, and $\overline{\mathtt{T}} \to \cdot$ for constructors. A member name ℓ extends $\mathtt{l}$ with the reserved name $\mathbb{C}$ as the unified name for all constructors. A member typing context Δ is the mapping from names to member types. The notation $\Delta[\ell \mapsto \phi]$ means to extend Δ with the mapping from ℓ to ϕ. The notation $\Delta_1 \oplus \Delta_2$ means to merge two member typing contexts if the domains of Δ_1 and Δ_2 are disjoint; otherwise it generates a type error. A structural typing context Θ is the mapping from structure names to member typing contexts. The notation $\Theta \uplus [\kappa \mapsto \Delta]$ means to extend Θ with the new mapping from κ to Δ if $\kappa \notin \Theta$, or means $(\Theta \backslash \kappa) \uplus [\kappa \mapsto \Delta \oplus \Theta(\kappa)]$ if $\kappa \in \Theta$. A variable binding context Γ can bind a term variable to a nominal type ($\mathtt{x} : \mathtt{T}$), a range variable $\mathtt{X}^o$, the $\mathtt{thisType}$ variable, or a name variable η.

We define the function $\mathbb{N}$, which computes the nominal representation of a range: $\mathbb{N}(\mathtt{X}) = \mathtt{X}$, $\mathbb{N}(\mathtt{N}) = \mathtt{N}$, $\mathbb{N}(\mathtt{R}|\{\overline{\mathtt{l}}\}) = \mathbb{N}(\mathtt{R})$, $\mathbb{N}(\mathtt{R}\backslash\{\overline{\mathtt{l}}\}) = \mathbb{N}(\mathtt{R})$, and $\mathbb{N}(\mathtt{C} \diamond \{\ldots\}) = \mathtt{C}$. We define the function δ, which literally translates a member pattern into a single member typing context: $\cdot[\ell \mapsto \phi]$.

Before type-checking a program, the compiler computes the structural type (this is member typings) for each class and each trait in the program, thus the following two structural typing contexts are available if there are no name conflicts. First is the context Θ^{CL} that maps the names of all the classes into their member typings. The members of a class include declared members,

members imported from used traits, and members inherited from the superclasses. Second is the context Θ^{TR} that maps the names of all the traits into their member typings. The members of a trait include declared members, members imported from used traits, and members in trait requirement.

We omit the rules for type-checking ranges ($\Gamma \vdash \texttt{R}$ ok), member patterns ($\Gamma \vdash \texttt{P}$ ok), names ($\Gamma \vdash \texttt{l}$ ok), and nominal types ($\Gamma \vdash \texttt{T}$ ok). Those typing rules are straightforward. We also omit the subtyping rules ($\texttt{T} <: \texttt{T}$). We do not have bounded type parameters in the kernel calculus, therefore the subtyping rules are similar to those for FJ [30].

In Figure 6.8, the pattern statement typing has three typing rules. The first rule is for the case when the range `R`'s nominal part is some class name `C`. In this case, we extend the member typing context of `C` with the member typing generated from the pattern `P`: $\Theta \uplus [\texttt{C} \mapsto \delta(\texttt{P})]$. The first rule type-checks the following code:

```
1 C obj;
2 pattern <name f> public int @f in C {
3     println(@f+" has value "+ obj.@f);
4 }
```

By type-checking the header of the pattern (line 2), `C`'s member typing context is extended with $\texttt{f} \mapsto$ **int**, which means `C` has field `f` of type **int** within the scope of the pattern's body. So in line 3, the expression obj.@f is well-typed. The second rule is for the case when `R`'s nominal part is some range variable `X` and $\texttt{X}^{+} \in \Gamma$, thus we know `X` is a super type of `thisType` and the members (excluding constructors) of `X` should be accessible via the `this` variable. In this case, if `P` is a field or a method signature, we extend the member typing contexts of both `X` and `thisType` with $\delta(\texttt{P})$; or if `P` is a constructor signature, then only the member typing context of `X` is extended, because a constructor is not inheritable. In the following code:

```
1 trait T<this-class X>
2 provides {
3     public m() {
```

Figure 6.8: Statement and expression typing.

Pattern statement typing: $\boxed{\Gamma;\Theta \vdash \texttt{ps ok}}$

$$\frac{\mathbb{N}(\texttt{R}) = \texttt{C} \qquad \Gamma \vdash \texttt{R ok} \qquad \Gamma' = \Gamma, \overline{\texttt{X}^-}, \overline{\eta} \quad \Gamma' \vdash \texttt{P ok} \quad \Theta' = \Theta \uplus [\texttt{C} \mapsto \delta(\texttt{P})] \quad \Gamma';\Theta' \vdash \texttt{ns ok}}{\Gamma;\Theta \vdash \texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> P in R \{ ns; \} ok}}$$

$$\frac{\begin{array}{c}\mathbb{N}(\texttt{R}) = \texttt{X} \quad \texttt{X}^+ \in \Gamma \qquad \Gamma' = \Gamma, \overline{\texttt{X}^-}, \overline{\eta} \qquad \Gamma' \vdash \texttt{P ok} \qquad \delta(\texttt{P}) = \Delta \\ (\texttt{P} = \texttt{F} \text{ or } \texttt{P} = \texttt{H}) \text{ implies } \Theta' = \Theta \uplus [\texttt{X} \mapsto \Delta] \uplus [\texttt{thisType} \mapsto \Delta] \\ \texttt{P} = \texttt{Q} \text{ implies } \Theta' = \Theta \uplus [\texttt{X} \mapsto \Delta] \quad \Gamma';\Theta' \vdash \texttt{ns ok} \quad \Gamma \vdash \texttt{R ok}\end{array}}{\Gamma;\Theta \vdash \texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> P in R \{ ns; \} ok}}$$

$$\frac{\begin{array}{c}(\mathbb{N}(\texttt{R}) = \texttt{X} \text{ and } \texttt{X}^- \in \Gamma) \text{ or } \mathbb{N}(\texttt{R}) = \texttt{thisType} \\ \Gamma' = \Gamma, \overline{\texttt{X}^-}, \overline{\eta} \qquad \Gamma' \vdash \texttt{P ok} \qquad \Gamma \vdash \texttt{R ok} \\ \Theta' = \Theta \uplus [\mathbb{N}(\texttt{R}) \mapsto \delta(\texttt{P})] \qquad \Gamma';\Theta' \vdash \texttt{ns ok}\end{array}}{\Gamma;\Theta \vdash \texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> P in R \{ ns; \} ok}}$$

Statement typing: $\boxed{\Gamma;\Theta \vdash \texttt{s} : \texttt{T}}$ and $\boxed{\Gamma;\Theta \vdash \texttt{ns ok}}$

$$\frac{\Gamma;\Theta \vdash \texttt{ps ok} \quad \Gamma;\Theta \vdash \texttt{s} : \texttt{T}}{\Gamma;\Theta \vdash \texttt{ps; s} : \texttt{T}} \qquad \frac{\Gamma;\Theta \vdash \texttt{e} : \texttt{T}_0 \quad \texttt{T}_0 <: \texttt{T} \quad \Gamma, \texttt{x} : \texttt{T};\Theta \vdash \texttt{s} : \texttt{U}}{\Gamma;\Theta \vdash \texttt{T x = e; s} : \texttt{U}}$$

$$\frac{\Gamma;\Theta \vdash \texttt{x} : \texttt{T} \quad \Gamma;\Theta \vdash \texttt{e} : \texttt{T}_0 \quad \texttt{T}_0 <: \texttt{T} \quad \Gamma;\Theta \vdash \texttt{s} : \texttt{U}}{\Gamma;\Theta \vdash \texttt{x = e; s} : \texttt{U}}$$

$$\frac{\Gamma;\Theta \vdash \texttt{e} : \texttt{T}}{\Gamma;\Theta \vdash \texttt{return e} : \texttt{T}} \qquad \frac{\text{for some } \texttt{e} \quad \Gamma;\Theta \vdash \texttt{(ns; return e)} : \texttt{T}}{\Gamma;\Theta \vdash \texttt{ns ok}}$$

Expression typing: $\boxed{\Gamma;\Theta \vdash \texttt{e} : \texttt{T}}$

$$\frac{\texttt{x} : \texttt{T} \in \Gamma}{\Gamma;\Theta \vdash \texttt{x} : \texttt{T}} \qquad \frac{\Gamma;\Theta \vdash \texttt{e} : \texttt{T} \quad \Gamma \vdash \texttt{l ok} \quad \Delta = \Theta(\texttt{T})}{\Gamma;\Theta \vdash \texttt{e.l} : \Delta(\texttt{l})}$$

$$\frac{\begin{array}{c}\Gamma;\Theta \vdash \texttt{e} : \texttt{U} \quad \Gamma \vdash \texttt{l ok} \quad \Delta = \Theta(\texttt{U}) \\ \Delta(\texttt{l}) = \overline{\texttt{T}} \to \texttt{T} \quad \Gamma;\Theta \vdash \overline{\texttt{e}} : \overline{\texttt{U}} \quad \overline{\texttt{U}} <: \overline{\texttt{T}}\end{array}}{\Gamma;\Theta \vdash \texttt{e.l(}\overline{\texttt{e}}\texttt{)} : \texttt{T}}$$

$$\frac{\Gamma \vdash \texttt{T ok} \quad \Delta = \Theta(\texttt{T}) \quad \Delta(\mathbb{C}) = \overline{\texttt{T}} \to \cdot \quad \Gamma;\Theta \vdash \overline{\texttt{e}} : \overline{\texttt{U}} \quad \overline{\texttt{U}} <: \overline{\texttt{T}}}{\Gamma;\Theta \vdash \texttt{new T(}\overline{\texttt{e}}\texttt{)} : \texttt{T}}$$

$$\frac{\Gamma \vdash \texttt{T ok} \quad \Gamma;\Theta \vdash \texttt{e} : \texttt{U} \quad \texttt{U} <: \texttt{T} \text{ or } \texttt{T} <: \texttt{U}}{\Gamma;\Theta \vdash \texttt{(T)e} : \texttt{T}} \qquad \frac{\Gamma \vdash \texttt{T ok} \quad \Gamma;\Theta \vdash \texttt{e} : \texttt{U} \quad \texttt{U} \not<: \texttt{T} \text{ and } \texttt{T} \not<: \texttt{U} \quad \textit{stupid warning}}{\Gamma;\Theta \vdash \texttt{(T)e} : \texttt{T}}$$

```
4         pattern <T, name f> public T @f in X {
5             T fx = this.@f;
6         }
7         pattern <> public constructor() in X {
8             X objx = new X();
9             thisType objths = new thisType();      // error
10        }
11    }
12 }
```

The expression of field access in line 5 is well-typed; the instance creation in line 8 is well-typed, but the instance creation in line 9 is not well-typed. The third rule is for the case when `R`'s nominal part is some range variable `X` with $\texttt{X}^- \in \Gamma$, or it is the `thisType` variable. If it is `X`, then we extend `X`'s member typing context with $\delta(\texttt{P})$, otherwise we extend the `thisType`'s member typing context with $\delta(\texttt{P})$. The third rule type-checks the following code:

```
1  trait T<class X> requires { X obj; }
2  provides {
3     public m() {
4        pattern <T, name f> public T @f in X {
5            T fobj1 = obj.@f;
6            T fobj2 = this.@f;      // error
7        }
8        pattern <T, name f> public T @f in thisType {
9            T fths1 = this.@f;
10           T fths2 = obj.@f;      // error
11       }
12    }
13 }
```

Type variables `X` and `thisType` are unrelated (i.e. no subtype relations). In the above code, the first pattern only allows us to access `f` via an instance of `X`, so the field access in line 6 is not

well-typed. The second pattern only allows us to access `f` via the `this` variable, so the field access in line 10 is not well-typed.

In FJ, expression typing uses the additional functions for finding a field/method type. In contrast, because the member typing context of a class changes within different environments, the rules for typing expressions in Figure 6.8 obtain the type of a member via a member typing context.

Figure 6.9 shows the rules for typing methods, trait applications, traits, and meta classes.

There are two rules for typing a method. The first rule type-checks a method that resides in some class `C`, then the type of the `this` variable is `C` and the validity of method overriding needs to be checked. The second rule type-checks a method that resides in a trait, then the type of the `this` variable is the `thisType` variable.

For the rule of type-checking a trait application, we have function $\Gamma \vdash \overline{\mathbb{N}(\mathtt{R})} \ \text{ in } \ \mathtt{N} \Rightarrow \overline{o}$ that computes the `this` access control modifiers for a sequence of ranges. According to the definition of function $\mathbb{N}$, the value of $\mathbb{N}(\mathtt{R})$ can be either a type variable or a class name. The definition of function $\Gamma \vdash \overline{\mathbb{N}(\mathtt{R})} \ \text{ in } \ \mathtt{N} \Rightarrow \overline{o}$ is shown as follows:

$$\frac{\mathtt{X}^{o} \in \Gamma}{\Gamma \vdash \mathtt{X} \ \text{ in } \ \mathtt{N} \Rightarrow o} \qquad \frac{\mathtt{N} <: \mathtt{N}_0}{\Gamma \vdash \mathtt{N}_0 \ \text{ in } \ \mathtt{N} \Rightarrow +} \qquad \frac{\mathtt{N} \not<: \mathtt{N}_0}{\Gamma \vdash \mathtt{N}_0 \ \text{ in } \ \mathtt{N} \Rightarrow -}$$

The rule for typing a trait application checks if the modifiers of the parameters are consistent with the modifiers of the trait arguments; and also checks if the trait requirements are satisfied in the use context `N`.

For the rule of typing a trait declaration, we add the `thisType` variable into the binding context so that the `thisType` can appear inside a trait. The initial structural typing context is

$$\Theta^{CL} \uplus [\mathtt{thisType} \mapsto \Theta^{TR}(\mathcal{T})]$$

so that we can access a member via the `this` variable in a trait.

For a meta class, we do not allow an imported method to override a method in the meta class or in the meta class's superclasses. Such restriction is checked when the compiler computes the structural typing context for the meta class before type-checking. So in the rule of typing a meta class, we do not need to check if an imported method correctly overrides a method in a superclass.

Figure 6.9: Method, trait application, trait, and class typing.

Method typing: $\boxed{\Gamma; \Theta \vdash \texttt{M ok in N}}$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{\texttt{T}}, \texttt{T ok} \quad \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C}; \Theta \vdash \texttt{s} : \texttt{U} \quad \texttt{U} <: \texttt{T} \\ \texttt{class C extends } \texttt{C}_0 \texttt{ \{ ... \} uses ...} \\ (\Theta^{CL}(\texttt{C}_0))(\texttt{m}) = \overline{\texttt{U}} \to \texttt{U} \text{ implies } \texttt{U} = \texttt{T} \text{ and } \overline{\texttt{U}} = \overline{\texttt{T}}\end{array}}{\Gamma; \Theta \vdash \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ s; \}} \text{ ok in } \texttt{C}}$$

$$\frac{\Gamma \vdash \overline{\texttt{T}}, \texttt{T ok} \quad \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{thisType}; \Theta \vdash \texttt{s} : \texttt{U} \quad \texttt{U} <: \texttt{T}}{\Gamma; \Theta \vdash \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ s; \}} \text{ ok in } \texttt{thisType}}$$

Trait application typing: $\boxed{\Gamma; \Theta \vdash \texttt{E ok in N}}$

$$\frac{\begin{array}{c}\texttt{trait } \mathcal{T}\texttt{<}\overline{\texttt{X}^o}\texttt{>} \texttt{ req \{ } \overline{\texttt{F}}\texttt{; } \overline{\texttt{H}} \texttt{ \} prov \{ ... \}} \\ \Gamma \vdash \overline{\texttt{R}} \text{ ok} \qquad \Gamma \vdash \overline{\mathbb{N}(\texttt{R})} \text{ in } \texttt{N} \Rightarrow \overline{o} \qquad \Theta(\texttt{N}) = \Delta \\ \text{for each } \texttt{T f} \text{ in } \overline{\texttt{F}} : \quad \Delta(\texttt{f}) = \texttt{U} \text{ and } \texttt{U} = \texttt{T} \\ \text{for each } \texttt{T m}(\overline{\texttt{T}}) \text{ in } \overline{\texttt{H}} : \quad \Delta(\texttt{m}) = \overline{\texttt{U}} \to \texttt{U} \text{ and } \texttt{U} = \texttt{T} \text{ and } \overline{\texttt{U}} = \overline{\texttt{T}}\end{array}}{\Gamma; \Theta \vdash \mathcal{T}\texttt{<}\overline{\texttt{R}}\texttt{>} \text{ ok in } \texttt{N}}$$

Trait typing: $\boxed{\texttt{TR ok}}$

$$\frac{\begin{array}{c}\Gamma = \overline{\texttt{X}^o}, \texttt{thisType} \qquad \Gamma \vdash \overline{\texttt{F}} \text{ ok} \qquad \Gamma \vdash \overline{\texttt{H}} \text{ ok} \\ \Theta = \Theta^{CL} \uplus [\texttt{thisType} \mapsto \Theta^{TR}(\mathcal{T})] \\ \Gamma; \Theta \vdash \overline{\texttt{M}} \text{ ok in } \texttt{thisType} \quad \Gamma; \Theta \vdash \overline{\texttt{E}} \text{ ok in } \texttt{thisType}\end{array}}{\texttt{trait } \mathcal{T}\texttt{<}\overline{\texttt{X}^o}\texttt{> req \{ } \overline{\texttt{F}}\texttt{; } \overline{\texttt{H}} \texttt{ \} prov \{ } \overline{\texttt{M}}\texttt{; use } \overline{\texttt{E}} \texttt{ \} ok}}$$

Meta class typing: $\boxed{\texttt{MC ok}}$

$$\frac{\begin{array}{c}\texttt{K} = \texttt{C}(\overline{\texttt{C}_0}\ \overline{\texttt{f}_0}\texttt{, } \overline{\texttt{C}}\ \overline{\texttt{f}})\texttt{\{super(}\overline{\texttt{f}_0}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;\}} \\ \mathit{fields}(\texttt{C}_0) = \overline{\texttt{C}_0}\ \overline{\texttt{f}_0} \qquad \emptyset; \Theta^{CL} \vdash \overline{\texttt{M}} \text{ ok in } \texttt{C} \qquad \emptyset; \Theta^{CL} \vdash \overline{\texttt{E}} \text{ ok in } \texttt{C}\end{array}}{\texttt{class C extends } \texttt{C}_0 \texttt{ \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}} \texttt{ \} uses } \overline{\texttt{E}} \text{ ok}}$$

6.5.3 Meta-evaluation

The meta-evaluation of a metaprogram includes (1) pattern-matching and code generation, and (2) trait flattening (i.e. methods from traits are inlined into classes). Because many previous papers, including our previous chapter, have fully discussed the process of trait flattening [42, 45, 36, 18], we here focus on the former. Because pattern-matching and code generation is performed at the statement level while trait flattening is performed at the member level, there is no side effect if we discuss the evaluate rules for those separately.

A statement-level pattern is evaluated into a sequence of statements. In Figure 6.10, we present part of the reduction rules for patterns. The rules are for the pattern-matching of fields. Those rules can be applied to the pattern-matching of methods or constructors with some slight modification. In the figure, the first rule is for the case when the pattern matches a field (it can be implemented by the unification algorithm), then it generates a sequence of instantiated statements. If the local variables in a generated statement have name conflicts with other local variables, we perform α renaming to substitute those local variable for fresh ones. The second rule is for the case when the match fails, then the pattern continues to pattern-match the rest fields. The third rule is for the case when there is no field, then the pattern generates an empty statement sequence.

Figure 6.10: Pattern-matching and code generation using a field pattern.

$\texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{(\texttt{T}_0\ \texttt{f}_0)\texttt{,}\overline{\texttt{F}}\texttt{;Q;}\overline{\texttt{H}}\}\ \texttt{\{ ns; \}} \longrightarrow$
$[\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{ns; (pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{\overline{\texttt{F}}\texttt{;Q;}\overline{\texttt{H}}\}\ \texttt{\{ ns; \})}$
where $\exists\overline{\texttt{T}}.\exists\overline{\texttt{l}}$ such that $[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{T} = \texttt{T}_0$ and $[\overline{\texttt{l}}/\overline{\eta}]\texttt{f} = \texttt{f}_0$

$\texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{(\texttt{T}_0\ \texttt{f}_0)\texttt{,}\overline{\texttt{F}}\texttt{;Q;}\overline{\texttt{H}}\}\ \texttt{\{ ns; \}} \longrightarrow$
$\texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{\overline{\texttt{F}}\texttt{;Q;}\overline{\texttt{H}}\}\ \texttt{\{ ns; \}}$
where $\nexists\overline{\texttt{T}}.\nexists\overline{\texttt{l}}$ such that $[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{T} = \texttt{T}_0$ and $[\overline{\texttt{l}}/\overline{\eta}]\texttt{f} = \texttt{f}_0$

$\texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{\texttt{[]}\texttt{;Q;}\overline{\texttt{H}}\}\ \texttt{\{ ns; \}} \longrightarrow \texttt{[]}$

6.5.4 Soundness

In this section, we give the properties of our meta language. Our purpose is to show that a metaprogram always generates a piece of well-typed code.

Lemma 10 (Staged Type Preservation of Statements)**.** *For some meta-level non-pattern statement* $\mathtt{s}$ *(i.e. statement without patterns), if* $\Gamma;\Theta \vdash \mathtt{s} : \mathtt{C}$ *where* $TyVars(\Gamma) = \emptyset$*, then when applying object-level type-checking to* $\mathtt{s}$*, we have* $\Gamma' \vdash \mathtt{s} : \mathtt{C}$ *for some object-level typing environment* Γ'*.*

Definition 1 (Class Name Alias $\tilde{\mathtt{N}}$)**.** *An alias of a class name, written* $\tilde{\mathtt{N}}$*, is a distinct class name. A class name can have multiple name alias.*

(1) An alias of a class name preserves the nominal subtyping relations of that class.

(2) For any class alias $\tilde{\mathtt{N}}$*,* $\tilde{\mathtt{N}} \notin \mathrm{dom}(\Theta^{CL})$*.*

(3) For some class name alias $\tilde{\mathtt{N}}$ *and some* Θ*, if* $\tilde{\mathtt{N}}$ *does not appear in the domain of* Θ*, then* $\Theta(\tilde{\mathtt{N}}) = \Theta^{CL}(\mathtt{N})$*; otherwise because there exists some* Δ *such that* $\tilde{\mathtt{N}} \mapsto \Delta \in \Theta$*,* $\Theta(\tilde{\mathtt{N}}) = \Delta$*.*

Lemma 11 (Type Substitution Preserves Typing)**.**

i) Suppose $\mathbf{this} : \mathtt{N}_0 \in \Gamma, \Gamma'$*;* $\tilde{\mathtt{N}}$ *is an alias of* $\mathtt{N}$*; and* $\tilde{\mathtt{N}} \notin \mathrm{dom}(\Theta)$*. If* $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{e} : \mathtt{U}$*,* $\Gamma \vdash \mathtt{N}$ ok*, and* $\Gamma \vdash \mathbb{N}(\mathtt{N})$ in $\mathtt{N}_0 \Rightarrow o$*, then* $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{e} : [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U}$*.*

ii) Suppose $\mathbf{this} : \mathtt{N}_0 \in \Gamma, \Gamma'$*;* $\tilde{\mathtt{N}}$ *is an alias of* $\mathtt{N}$*; and* $\tilde{\mathtt{N}} \notin \mathrm{dom}(\Theta)$*. If* $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{ns}$ ok*,* $\Gamma \vdash \mathtt{N}$ ok*, and* $\Gamma \vdash \mathbb{N}(\mathtt{N})$ in $\mathtt{N}_0 \Rightarrow o$*, then* $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{ns}$ ok*.*

Lemma 12 (Name Substitution Preserves Typing)**.**

i) If $\Gamma, \eta, \Gamma'; \Theta \vdash \mathtt{e} : \mathtt{U}$*, and* $\Gamma \vdash \mathtt{l}$ ok*, then* $\Gamma, \Gamma'; [\mathtt{l}/\eta]\Theta \vdash [\mathtt{l}/\eta]\mathtt{e} : \mathtt{U}$*.*

ii) If $\Gamma, \eta, \Gamma'; \Theta \vdash \mathtt{ns}$ ok*, and* $\Gamma \vdash \mathtt{l}$ ok*, then* $\Gamma, \Gamma'; [\mathtt{l}/\eta]\Theta \vdash [\mathtt{l}/\eta]\mathtt{ns}$ ok*.*

Lemma 13 (Statement Concatenation Preserves Typing)**.** *Suppose function lvar collects all the local variables in a sequence of statements. If* $\Gamma;\Theta \vdash \mathtt{ns}_1$ ok*,* $\Gamma;\Theta \vdash \mathtt{ns}_2$ ok*, and* $lvars(\mathtt{ns}_1) \cap lvars(\mathtt{ns}_2) = \emptyset$*, then* $\Gamma;\Theta \vdash \mathtt{ns}_1\,\mathtt{;ns}_2$ ok*.*

Theorem 8 (Type Preservation for Patterns)**.** *If* $\Gamma;\Theta \vdash \mathtt{ps}$ ok *and* $\mathtt{ps} \longrightarrow \mathtt{ns}$*, then* $\Gamma;\Theta \vdash \mathtt{ns}$ ok

Theorem 9 (Progress for Patterns)**.** *If* $\emptyset;\Theta \vdash \mathtt{ps}$ ok *and* $\mathtt{ps}$ *is in the form of* $\mathtt{pattern}\langle\overline{\mathtt{X}},\overline{\eta}\rangle\ \mathtt{P}\ \mathtt{in}\ \mathtt{R}\ \{\ \mathtt{ns}_0\ \}$*, then there is some* $\mathtt{ns}$ *with* $\mathtt{ps} \longrightarrow \mathtt{ns}$ *(suppose* $\mathtt{ns}$ *includes empty statements).*

6.6 Summary

Java reflection enables us to write reusable programs that are independent of certain classes. However, when runtime performance is a big concern, we propose to use compile-time reflection for writing metaprograms that generate non-reflective class/type specific code, which has lower runtime cost.

In this chapter, we introduce pattern-based reflection at the statement level, which supports statement-level code generation and reuse. We present reified generics, which enable metaprograms to access and manipulate type variables during meta-evaluation. We also introduce meta-level functions, which enable users to inspect metadata at the compile time.

We formalize our language, presenting the calculus and the type systems for both the object language and the meta language. We show that the meta language always generates well-typed statements.

Chapter 7

PtjORM: Object-Relational Mapping Library

Object-relational mapping (ORM for short) is the ability of mapping classes and objects from/to tables and records in a relational database. An implementation of ORM normally needs the frequent use of reflection over class members. Some Java ORM libraries, such as Hibernate [1] , use Java run-time reflection [2] , which produces some overhead of inspecting over class members at run-time. In the following, we quote Java's documentation [3] on reflection:

> "Performance overhead: because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications."

Though Hibernate supports bytecode-level reflection optimization [4] (using CGLib [5] or Javassist [6]) to speed up the performance of object construction and field access, it cannot completely avoid the runtime overhead of reflection.

This chapter introduces an object-relational mapping library, called PtjORM, which is a real-world application of Pattern-based Traits. PtjORM is written in Pattern-based Trait for Java. In

[1] http://www.hibernate.org
[2] http://docs.jboss.org/ejb3/app-server/HibernateAnnotations/api/org/hibernate/reflection/package-use.html
[3] http://docs.oracle.com/javase/tutorial/reflect/index.html
[4] http://docs.jboss.org/hibernate/orm/3.5/javadoc/org/hibernate/bytecode/package-summary.html
[5] http://cglib.sourceforge.net
[6] http://www.jboss.org/javassist

terms of functionality, current version of PtjORM is lightweight. It is not as powerful and versatile as Hibernate. However, PtjORM supports compile-time reflective metaprogramming, thus avoiding the overhead of runtime reflection. Suppose we need to map the objects of some class, PtjORM, at compile-time, reflects over the class and generates the object-relational mapping code specific to that class. The generated code is Java source code, which is easy for users to modify or debug if necessary.

7.1 Object-Relational Mapping

Like many other ORM software products, PtjORM supports property (field) mapping, association mapping, and inheritance mapping. To begin with, we give the following classes (Resume, Course, Person, Student, and Faculty) to illustrate the mappings. Figure 7.1 shows the result of mapping these classes into MySQL tables. In the figure, P.K. is short for primary key and F.K. is short for foreign key. The records in the tables shows the object-relational mapping of the objects: one faculty, two students, two courses, and one resume were mapped and saved into the tables. The annotated arrows in the figure shows different kinds of association relations among the classes. We will explain class associations in Section 7.1.

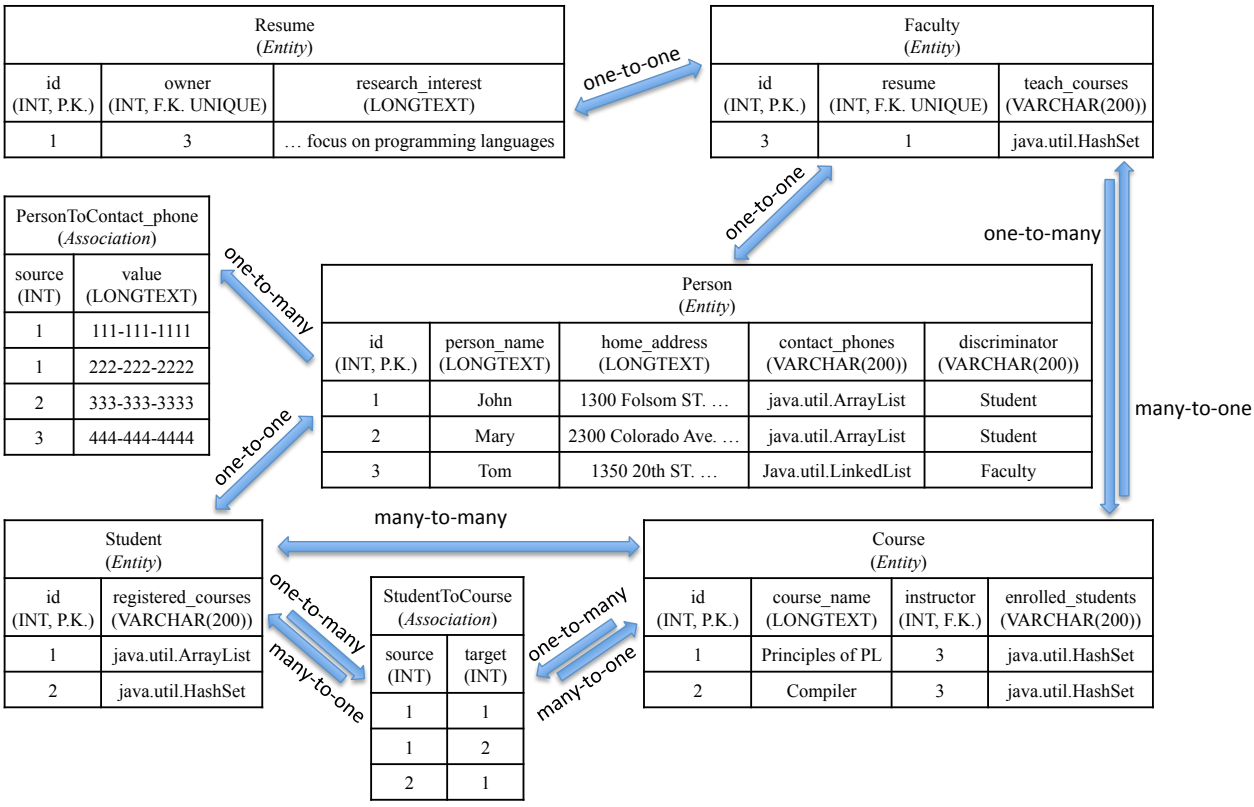
```
1  class Resume {
2      private int id; // identifier
3      private Faculty owner; // onetoone
4      private String research_interest; // property
5      ....
6  }
7  class Course {
8      private int id; // identifier
9      private String course_name; // property
10     private Faculty instructor; // manytoone
11     private java.util.Collection<Student> enrolled_students; // manytomany
12     ...
```

```
13 }
14 class Person {
15     private int id; // identifier
16     private String person_name; // property
17     private String home_address; // property
18     private java.util.List<String> contact_phones; // onetomany (unidirectional)
19     ...
20 }
21 class Student extends Person {
22     private java.util.Collection<Course> registered_courses; // manytomany
23     ...
24 }
25 class Faculty extends Person {
26     private Resume resume; // onetoone
27     private java.util.Collection<Course> teach_courses; // onetomany
28     ...
29 }
```

7.1.1 Property Mapping

The basic ORM is to map the identifier and the property fields of a class. Each object-relational mappable class should have an identifier (e.g. field id in class Person), which maps to a primary-key column of a table. Different objects of the same class should have different identifier values. A property field (e.g. field home_address in class Person) of a class also maps to a column of a table. The type (including Java primitive types) of an identifier or a property field can be directly mapped to the datatype of a column. For instance, primitive type **byte** can be mapped to TINYINT in MySQL; **long** can be mapped to BIGINT in MySQL; and String can be mapped to LONGTEXT in MySQL.

Figure 7.1: The MySQL tables mapped from the classes in Section 7.1. In the figure, P.K. is short for primary key and F.K. is short for foreign key. The records in the tables were mapped from the following objects: one faculty, two students, two courses, and one resume. The annotated arrows shows the association relations among the classes.



7.1.2 Association Mapping

PtjORM supports the mapping of four kinds of associations: one-to-one, many-to-one, one-to-many, and many-to-many. For the above classes, field owner in class Resume and field resume in class Faculty have a one-to-one association, which means each resume has only one owner and different resumes have different owners. Field instructor in class Course and the corresponding field teach_courses in class Faculty has a one-to-many relation, which means one course can be taught by only one faculty but a faculty can teach more than one course. Field enrolled_students in class Course and its corresponding field registered_courses in class Student has a many-to-many relation, which means a course can enroll many students and a student can register many courses. All of the relations above are bidirectional, but for field contact_phones in class Person, it has a one-to-many relation with phone numbers of type String, but this association is unidirectional because String is not defined as a mappable class entity and there is no corresponding field in class String. For one-to-many unidirectional associations and for many-to-many associations (both unidirectional and bidirectional) associations, we need to create association tables. For example in Figure 7.1, table PersonToContact_phones is the association table that maps persons' identifiers to their phone numbers, and table StudentToCourse is the association table that maps students' identifiers from/to courses' identifiers (i.e. the mapping is bidirectional).

7.1.3 Inheritance Mapping

Hibernate provides three basic inheritance mapping strategies [7] : table per hierarchy, table per subclass, and table per concrete class. With flexible user-level inheritance mapping configuration, Hibernate enables seven inheritance mapping strategies derived from the above three. PtjORM currently supports two kinds of inheritance mapping strategies: table per subclass and table per concrete class. In the future, we plan to extend the inheritance mapping of PtjORM with the table per hierarchy strategy.

For table per hierarchy, only one table need to be created. The columns in the table is the

[7] http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html/ch10.html

union of the fields in Person, Student, and Faculty, plus a discriminator column which specifies the concrete class of an object saved.

For table per subclass, a table is created for each class. In the table of a subclass, its columns do not include the fields inherited from the base class. The table of a base class needs a discriminator column. We use table per subclass strategy in Figure 7.1. Table Student and table Faculty only have the columns for their own defined fields and they share table Person for saving the values of their inherited fields.

For table per concrete class, class hierarchy is flattened and class inheritance relation breaks. The table of a subclass contains all the columns of the table of the base class and there is no one-to-one association between the two tables. So, in this strategy, table Faculty and table Student would each have all the columns of table Person. Neither would have a one-to-one association to Person.

7.2 Object-Relational Mapping Configuration

At the implementation level, The PtjORM library is constructed in traits. So, it enables us to use trait applications for constructing ORM classes and configuring object-relational mappings.

In PtjORM, trait ORMMain is the library's main entry. We use it to construct ORM classes. The following is the header of trait ORMMain.

```
1 trait ORMMain<
2       nonabstract class X,
3       class id, class property,
4       class oneToOne, class manyToOne, class oneToMany, class manyToMany,
5       group config
6 >
```

It has eight parameters. The first one takes a non-abstract class which is object-relational mappable. The second one takes the identifier field of the class. The third one takes the property field(s). The next four parameters take the field(s) with different kind of associations, and the last one takes some

detailed association mapping and/or inheritance mapping configuration. Here, a parameter taking a set of fields means it takes the object-relational mappable class with those fields selected (using the field selection operator). A parameter taking some object-relational mapping configuration means it takes a group of trait applications, where the traits are defined specifically for configuring different kinds of object-relational mappings.

In detail, PtjORM provides the following traits to configure association mappings: OneToOne, ManyToOne, OneToManyEntity, OneToManyNonEntity, ManyToManyController, and ManyToManyControlled, and it provides the following traits to configure inheritance mapping: HasSubClass and HasBaseClass. For example, for mapping class Person (defined in Section 7.1), we present the following meta class PersonORM and trait PersonORMTrait.

```
1  public class PersonORM implements PtjORM.ObjectRelational {
2      use PersonORMTrait<>;
3  }
4  trait PersonORMTrait<> {
5      group Config {
6          use HasSubClass<>;
7          use OneToManyNonEntity<Person|{contact_phones}, String>;
8      };
9      use ORMMain<
10         Person,
11         Person|{id}, Person|{person_name, home_address},
12         Person|{}, Person|{}, Person|{contact_phones}, Person|{},
13         Config
14     >;
15 }
```

Meta class PersonORM is an ORM class for class Person. It provides the functions for mapping class Person and its instances. Trait PersonORMTrait is an ORM trait. It is used to construct class PersonORM.

The reason for using an ORM trait is tricky. We need to specify mapping configuration in a **group** constructor, which wraps a set of trait applications, but a **group** constructor cannot be defined in a class. So, we use a ORM trait, as an intermediate step, to define the **group** and pass it to trait ORMMain.

In trait PersonORMTrait, group Config wraps a set of ORM specifications. Because class Person is a base class, we use trait HasSubClass. It indicates to the program to add column discriminator into table Person when the table is created. Field contact_phones has an one-to-many association but its associated elements are non-entities. So, we use trait OneToManyNonEntity. It indicates the program to create an association table for field contact_phones. The second parameter of trait OneToManyNonEntity takes type String, which is the type of the associated elements. Of course, the type of the associated elements can be automatically inferred if it is not specified by users.

Next, we give the ORM class and the ORM trait for class Student, where the ORM class is the class for mapping class Student and its instances.

```
1  public class StudentORM implements PtjORM.ObjectRelational {
2      use StudentORMTrait<>;
3  }
4  trait StudentORMTrait<> {
5      group Config {
6          use HasBaseClass<Person>;
7          use ManyToManyController<Student|{registered_courses}, Course|{enrolled_students}>;
8      };
9      use ORMMain<
10         Student,
11         Student|{id}, Student|{},
12         Student|{}, Student|{}, Student|{}, Student|{registered_courses},
13         Config
14     >;
15 }
```

In trait StudentORMTrait, group Config wraps the ORM specifications for Student. We use trait HasBaseClass to specify that Student is a subclass of Person. It requires the tool to use the table per subclass strategy for inheritance mapping. We use trait ManyToManyController to specify the many-to-many association between Student and Course. Here, class Student is designated as the controller of this many-to-many association, which means class Student is responsible for creating and maintaining the association table.

7.3 Reflective Metaprogramming for PtjORM

Reflective Metaprogramming facilitates object-relational mapping. From the software development perspective, we are able to decompose the project into modules and decompose modules into unites of different functionalities, which are written in traits. Assembling the unites into the whole project is simple, just using the symmetric sum of traits. Pattern-based reflection provides an expressive way of inspecting class members, and with a clean syntax. From the software performance perspective, metaprogramming enables our tool to generate code that does not contain the reflection of class members, which helps in improving the runtime performance of ORM.

7.3.1 PtjORM: Built of Traits

The PtjORM library is modularly constructed. The current version of PtjORM (version 2.0) is composed of 63 traits, which totally implement 205 functions. In the library, each trait represents a unit of single functionality. Although many of the traits have required methods, their definitions are independent of each other. Trait ORMMain is the only trait responsible for connecting and combining the other traits together. Member name conflicts and method signature inconsistency (i.e. inconsistency between the signature of an implemented method and the signature of that method requirement) are solved in ORMMain.

Without using traits, we could not achieve such fine-grained modularity. Suppose we did not use traits, then we would have written the library in classes, and the methods in those classes would have been composed using object composition and method delegation, which produces mutual

dependency and indirectness of method calls. For instance, suppose class ORMMainC is the main class to integrate the classes of different functionality, class ORMClassMappingC is the class specific for mapping classes into database tables, and class TableNameManagerC specific for table name management. The following are the partial definitions of class ORMMainC, ORMClassMappingC, and TableNameManagerC.

```
1  public class ORMMainC {
2      private ORMClassMappingC clsmap; private TableNameManagerC tnmanager;
3      ...
4      public ORMMainC() {
5          this.clsmap = new ORMClassMappingC(this); // object composition
6          this.tnmanager = new TableNameManagerC(this); // object composition
7          ...
8      }
9      public String getTableName() { return this.tnmanager.getTableName(); } // method delegation
10     public void map() { this.clsmap.map(); } // method delegation
11     ...
12 }
13 public class ORMClassMappingC {
14     private ORMMainC main;
15     public ORMClassMappingC(ORMMainC main) { this.main = main; }
16     public void map() {
17         ....
18         this.main.getTableName(); // indirect call of getTableName defined in TableNameManagerC
19         ...
20     }
21     ...
22 }
23 public class TableNameManagerC {
24     private ORMMainC main; private String tn;
```

```
25      public TableNameManagerC(ORMMainC main) { this.main = main; }
26      public String getTableName() { return this.tn; } // original definition of getTableName
27      ...
28  }
```

Class ORMClassMappingC needs to use method getTableName from class TableNameManagerC and TableNameManagerC needs to use some methods from other classes. To combine the methods in ORMClassMappingC and TableNameManagerC, we create their instances (see line 5 and line 6) and define their method delegates (see line 9 and line 10) in class ORMMainC. Because ORMClassMappingC and TableNameManagerC need to use the methods from other classes, we have to create the mutual recursion by passing the **this** variable (the instance of ORMMainC) to them (see line 5 and line 6). Mutual recursion incurs mutual dependency. Besides, the call of method getTableName produces indirectness: it first calls the delegate method of getTableName in ORMMainC, which then forwards the call to the original method getTableName in TableNameManager.

7.3.2 PtjORM: Compile-Time Reflection

The PtjORM library uses compile-time reflection to iterate over class members. The following is a portion of code used by the function for object saving.

```
1  trait StoreManyToMany<nonabstract class X, class manyToMany>
2  provides {
3      public void storeManyToMany(X objx, ...) {
4          pattern P_manymany <T, name f> * T @f in manyToMany {
5              // case T is a subclass of java.util.Collection
6              pattern P_collection <S extends java.util.Collection> public S get#@f() in X {
7                  S col = objx.get#@f(); ...
8              }
9              // case T is an array type
10             pattern P_array <S extends Object> public S[] get#@f() in X {
11                 S[] arr = objx.get#@f(); ...
```

```
12              }
13          }
14      }
15  }
```

Trait StoreManyToMany is responsible for saving the fields with many-to-many associations. The header of pattern P_manymany in line 3 iterates over the fields in manyToMany. Within P_manymany, pattern P_collection matches a field if its type is a subclass of Java's Collection interface; and pattern P_array matches a field if its type is an array type. In line 6 and line 10, the program invokes the getter methods via the non-reflective approach. So, we apply trait StoreManyToMany to class Student in Section 7.1 with the arguments of Student and Student|{registered_courses}. At compile time it will generate the following method specific for Student.

```
1  public void storeManyToMany(Student objx, ...) {
2      { // the scope generated by pattern P_manymany
3          { // the scope generated by pattern P_collection
4              java.util.Collection<Course> col = objx.getRegistered_courses(); ...
5          }
6      }
7  }
```

In line 4, the compiler instantiates the call of the getter method for field registered_courses.

The PtjORM library also uses compile-time reflection for class instance creation. For a parameterized non-abstract class type variable X, without compile-time reflection, an instance of class X needs to be created using the Java runtime reflection like the following.

```
1  trait ObjectGeneration<nonabstract class X>
2  provides {
3      public X createEmptyInstance() throws java.lang.Exception {
4          return (X) X.class.newInstance();
5      }
6  }
```

The call of the newInstance method involves the runtime check and finding a nullary constructor for an instance of X. Method newInstance may throw four kinds of exceptions (including IllegalAccessException, InstantiationException, ExceptionInInitializerError, and SecurityException). This means a Java VM needs to perform the runtime check in many aspects. When we need to obtain a large set of objects from the database, then there will be a great number of calls to the newInstance method, thus will compromise the runtime performance.

In contrast, PtjORM uses compile-time reflection. The following shows the non-reflection version of object creation in PtjORM.

```
1  trait ObjectGeneration<nonabstract class X>
2  provides {
3      public X createEmptyInstance() throws ptjorm.exceptions.ObjectCreationError {
4          X objx = null;
5          pattern P <name f> public @f() in X { objx = new X(); }
6          if (objx == null)
7              throw new ptjorm.exceptions.ObjectCreationError(X::getName() +
8                  " has no public nullary constructor.")
9          return objx;
10     }
11 }
```

In line 5, pattern P matches a public nullary constructor in non-abstract class X. If a public nullary constructor exists, then the program creates an object of X using the Java **new** operator, in a non-reflective way.

7.4 Evaluation

The runtime performance of an ORM tool can be influenced by many factors. Besides the type of reflection, they also include the mode of saving/fetching objects (saving mode such as cascade saving; and fetching mode such as eager fetching, lazy fetching, batch fetching, etc) , which could vary the amount of communication with a database management system. Here, the amount of

communication is measured by the number of executed SQL statements. So, instead of evaluating the feature of compile-time reflection solely, we evaluate the overall runtime performance of our PtjORM library.

We evaluated the PtjORM library by testing it with a benchmark based on a Java implementation of the 007 benchmark [11]. The 007 benchmark was originally implemented in C++. It tests the performance of persistence for a CAD (computer assisted design) application. Ibrahim and Cook provided its Java implementation [29] to test the performance of their autofetch (automatic query turning) strategy for ORM. We modified their Java implementation of the 007 benchmark. We used it to compare the runtime performance of saving and loading objects for the following four ORM tools: PtjORM (version 2.0), Hibernate (version 4.1), Ebean (version 2.7.7), and EclipseLink (version 2.4.1).

Hibernate, Ebean, and EclipseLink are all noncommercial, open-source, and well maintained ORM tools. By default, they all use some Java bytecode generation and manipulation (runtime) tool for reflection optimization. Even in that case, we are still interested to see whether PtjORM has competitive runtime performance.

To be fair, we tested and compared the above ORM tools by trying to use very similar configurations. For object saving, we use the cascade saving mode: when saving an object, a ORM tool also saves all the objects associated with that object. For object fetching, we use the eager fetching mode because the current version of PtjORM only supports eager fetching. When fetching an object, eager fetching enables a ORM tool to pre-fetch the associations and also the objects associated with that object even if they are not needed. Eager fetching is not efficient for fetching a large set of data but it avoids the N+1 selects problem. For association fetching, we additionally use batch fetching, which enables a ORM tool to use one single SQL query to fetch a set of objects. However for inheritance mapping, PtjORM used a strategy different from the other three. PtjORM used the table per subclass strategy while the others used the table per hierarchy strategy (PtjORM currently does not support table per hierarchy). So, the persistence tables created by PtjORM will have more records.

During the benchmark testing, we recorded the number of SQL statements executed by each ORM tool during each single test. It helps us see the difference in performance when they execute the similar number of SQL statements. Also, we can test if the number of SQL statements determines the runtime performance of a ORM tool.

The 007 benchmark we used can generate the databases of 4 different sizes: tiny, small, medium, and large. We did not perform the benchmark test with a large-size database. Considering its large volume, we think it is impractical and unnecessary to fetch all the objects at once. So, we used the databases of the first three sizes. The tiny one has 996 records; the small one has 11,465 records, and the medium one has 75,779 records.

We ran the 007 benchmark on a laptop with a 2.2 GHz, Intel® Core™ i7 processor (4 cores) and 8GB memory. The MySQL database was installed on a desktop with a 1.80GHz Intel® Pentium™ E2160 dual processor and 1GB memory. The desktop and the laptop were connected in the same local network.

7.4.1 Object Saving Evaluation

First, we give table 7.1, which shows the number of the SQL statements executed during the object saving tests (please note that the number of generated SQL statements is independent of databases). While the others used the table per hierarchy strategy, PtjORM used the table per subclass strategy for inheritance mapping and it generated the largest number of SQL statements.

The ORM tools were tested with MySQL database. The test results about the object-saving performance is show in Table 7.2 shows the results of the tests with MySQL database. The test results show that even with the largest number of executed SQL statements, PtjORM achieved the best object-saving performance. Approximately, PtjORM is 20% faster than Hibernate, 24% faster than EclipseLink, and 15% faster than Ebean.

Table 7.1: The number of SQL statements executed by each ORM tool for object saving.

DB size / ORM tools	tiny	small	medium
Ebean	1111	11.7K	75.9K
Eclipselink	1071	11.5K	75.7K
Hibernate	1005	11.6K	75.6K
PtjORM	1125	12.6K	76.7K

Table 7.2: The object-saving performance of the ORM tools tested with a MySQL database. The numbers in the table are used time for saving objects. Time unit ms is short for millisecond.

DB size / ORM tools	tiny	small	medium
Ebean	955ms	9610ms	62476ms
Eclipselink	1120ms	10084ms	64357ms
Hibernate	1169ms	9817ms	60178ms
PtjORM	823ms	8258ms	50365ms

7.4.2 Object Fetching Evaluation

We use the batch fetching mode for object fetching. For fetching a tiny database, we use batch sizes: 4, 8, 12, 40, 70, and 100. For fetching a small database, we use batch sizes: 40, 100, 160, 400, 700, and 1000. For fetching a medium database, we use batch sizes: 200, 600, 1000, 2000, 3500, and 5000. The tables in Table 7.3 show the number of SQL statements executed by each ORM tool for fetching the databases of three different sizes. From the tables, we note that PtjORM generates the largest number of SQL statements when the batch size is small. With the increase of batch size, the number of SQL statements generated by PtjROM drops dramatically. When fetching a large number of objects, we suggest PtjORM users to assign a large batch size for efficiency. For EclipseLink and Hibernate, increasing the batch size does not have a big influence on the number of SQL statements.

The tables in Table 7.4 show the time used by each ORM tool for fetching the databases of three different sizes. We learned that Ebean has the best performance and one reason is that it generates the least number of SQL statements. For fetching a tiny database, PtjORM is 2%−70% faster than Hibernate, -2%−70% faster than EclipseLink. When fetching a small database, except for the batch size of 40, PtjORM is 8%−58% faster than Hibernate; and PtjORM is 60% faster than EclipseLink in average. When fetching a medium database, PtjORM does not perform well with small batch sizes, but with larger batch sizes, its performance exceeds the performance of Hibernate and EclipseLink. With a small batch size, because PtjORM generates a greater number of SQL statements, the cost of executing the SQL statements and commuting with the server outweighs the saving from compile-time reflection. But when the batch size is equal or larger than 3500, its performance exceeds the performance of Hibernate and EclipseLink. In summary, PtjORM has the second best performance for fetching objects. Of course, from the benchmark test results, we feel the need to improve PtjORM by reducing the number of SQL statements that PtjORM generates.

Table 7.3: The number of SQL statements executed by each ORM tool for fetching the databases of three sizes

tools \ batch size	4	8	12	40	70	100
Ebean	203	118	91	50	41	37
Eclipselink	159	130	125	149	161	160
Hibernate	228	185	153	142	142	140
PtjORM	458	284	208	79	63	54

tools \ batch size	40	100	160	400	700	1000
Ebean	171	83	65	48	44	41
Eclipselink	431	314	303	290	288	286
Hibernate	428	395	368	335	358	374
PtjORM	976	469	287	141	109	74

tools \ size	200	600	1000	2000	3500	5000
Ebean	160	75	58	47	43	41
Eclipselink	454	404	395	389	394	393
Hibernate	615	485	518	527	526	523
PtjORM	1673	648	404	295	178	126

Table 7.4: Time used for fetching objects from the databases of three sizes

tools \ batch size	4	8	12	40	70	100
Ebean	345ms	254ms	217ms	159ms	148ms	140ms
Eclipselink	514ms	484ms	478ms	564ms	632ms	622ms
Hibernate	639ms	599ms	608ms	591ms	625ms	586ms
PtjORM	627ms	458ms	353ms	224ms	195ms	186ms

tools \ batch size	40	100	160	400	700	1000
Ebean	677ms	541ms	509ms	458ms	461ms	456ms
Eclipselink	7964ms	2465ms	2479ms	2390ms	2440ms	2391ms
Hibernate	1505ms	1463ms	1423ms	1365ms	1405ms	1421ms
PtjORM	2167ms	1352ms	1103ms	826ms	697ms	601ms

tools \ batch size	200	600	1000	2000	3500	5000
Ebean	2473ms	2287ms	2299ms	2241ms	2281ms	2285ms
Eclipselink	8337ms	8287ms	8165ms	7704ms	7746ms	7706ms
Hibernate	4970ms	4648ms	4743ms	4952ms	4952ms	5139ms
PtjORM	18365ms	9869ms	6802ms	5310ms	4453ms	4253ms

7.5 Summary

In this chapter, we start with an introduction to object-relational mapping. Next we present PtjORM, an object-relational mapping tool written in pattern-based traits. PtjORM is designed and developed with the composition of small functional and modular units (written in traits). PtjORM uses compile-time reflection, which avoids the overhead of Java's runtime reflection. Our benchmark tests show that it has competitive runtime performance compared to some of the mainstream Java ORM tools, such as Hibernate, EclipseLink, and Ebean.

Chapter 8

Related Work

Our related work includes the programming language features for writing generic and reusable components. We group our related work into three categories. Composition refers to the related work that supports writing modular programs, composing them and solving conflicts in a flexible way. Reflection refers to the related work that supports type inspection and analysis. Metaprogramming refers to the related work that supports writing code templates and performing partial evaluation and optimization.

8.1 Composition

There has been a line of work about the improvement of code reuse through the use and the extension of the trait concept. FeatherTrait Java (FTJ) [36] extends Featherweight Java with traits. FTJ does not reflect over classes but its type system is modular. Chai [55] is another language that extends Java with traits. Chai allows traits to be more generally used: a trait not only performs a building block for a class but also creates a subtype relation with the class, which means the name of the trait is also a type that an object of the class can be type-casted to. Moreover, at the site of a trait application, the used trait can be dynamically substituted by another trait if they implement the same interface. The type system of Chai is not modular: when the traits are flattened and merged into classes, they are type-checked for generating well-typed code.

Featherweight Jigsaw [35] applies the concept of traits into classes and therefore, classes are also building blocks with a set of composition operators used for traits. Besides name-level

operations (e.g. name exclusion) that are already present in the concept of traits, Jigsaw introduces member modifiers to solve name conflicts.

Our work is inspired by Reppy and Turon's work: metaprogramming with traits [45]. They introduced user-customizable traits (i.e. trait functions) that are parametrized over types, values, and (method and field) names. Trait functions have limited reflection. Reppy and Turon proposed pattern-matching for trait functions in their technical report [60], but the syntax is not as expressive as the pattern-based reflection implemented Huang and Smaragdakis [25]. Trait functions do not provide iteration over the members of a class, users have to manually write a trait application for each member in a class.

In the field of functional programming, MixML [46] extends ML modules with imports and outports (like the require and the provide clauses in a trait) for writing modular programs and for flexible composition. MixML does not have name ambiguity issue because it does not flatten a module that is composed by other modules. Instead, it keeps the composition hierarchy. So, when we need to access an imported member, we need to specify the path leading to that member if there are duplicate members derived from different imported modules.

8.2 Reflection

Programming languages such as Genoupe [15], SafeGen [28], CTR [17], and MorphJ [26, 25] support reflection. Genoupe introduces a type system for reflective program generators, which offers an particular high degree of static safety for reflection. However, the type system does not guarantee that generated code is always well-typed because it uses the properties that may depend on run-time values. SafeGen uses first-order logic formulae to express patterns that can generate code and these formulae are used by a theorem prover as pre-conditions to check the safety of generated code. Therefore, the type system of SafeGen is undecidable. CTR introduces transforms, which support pattern-based compile-time reflection and its type system is designed with high degree of modular static type safety.

MorphJ refines the type system of CTR with a modular type system. Considering its power

for static reflection, we believe that MorphJ is the calculus closest relative to ours. However, one reflective power of MorphJ that our system does not have is negative nested patterns, which can be used to prevent name conflicts. MorphJ has a modular type system and supports full static type safety. So its type system can report precise error messages and there is no need for type-checking during meta evaluation. Our type-checking for pattern-based traits is also modular, but would inevitably produce type errors that point to generated code. However, incremental type-checking enables the type system to report such errors as early as possible and therefore save programers' efforts to trace the errors back to source programs. Of course, our type system has its own merit: it can accept some useful but unsafe metaprograms that MorphJ's type system rejects.

Kiczals et. al. proposed aspect-oriented programming (AOP) [33] for the separation of cross-cutting concerns. One language implementation of AOP is AspectJ [32]. AspectJ weaves advice code into original application code at the Java bytecode level, so it does not generate source code. AspectJ inserts code flexibly, allowing users to define different point cuts for insertion. Pattern-based traits support writing programs that describe cross-cutting concerns, but they enable us to merely add wrapper code for members. Current stable version of AspectJ, that is AspectJ 5, supports Java generics, but it does not allow the use of type variables and name variables for pattern-matching. Concerning about the type system, AspectJ does not guarantee static type safety: a binding between a pointcut and an advice can cause type errors at runtime [14].

8.3 Metaprogramming

Some functional programming languages, such as MetaML [58], MetaOCaml, and Template Haskell [48], enable metaprogramming by providing multiple stages of computation, where earlier stages can manipulate code for late stages. These languages use explicit stage annotations for the support of code manipulation at expression level. Our programming language supports (two-staged) metaprogramming: pattern-matching, code generation, and trait flattening are performed at compile time while generated program is evaluated at run time.

MetaML and MetaOCaml do not support type reflection and generation, thus enabling their

type systems to guarantee static type safety of metaprograms and their residual code. In contrast, C++ templates and Template Haskell do not guarantee static type safety of residual code. Therefore, type errors may arise during meta evaluation. During meta evaluation, C++ templates type-check generated code near the end of meta evaluation; and Template Haskell type-checks when a splice of code is spliced. For some type errors raised at runtime, their type system cannot catch them as early as our incremental type-checking.

Some programming languages use dependent types to describe programs whose types are dependent on terms. Cayenne [5] supports the manipulation of types as normal data. It has dependent types, which contain the functions that generate types, but usually those functions have to be manually written. Typing a Cayenne program may involve computation, which may overlap with the computation of the program. In contrast, incremental type-checking takes advantage of meta evaluation such that type-checking and meta-evaluation interleave. Some other dependently typed programming languages, such as DML [62], ATS [1] , and Concoqtion [20], have to use some external proof assistant tools (i.e. theorem provers) for type-checking.

[1] http://www.ats-lang.org

Chapter 9

Conclusion and Future Work

In general, for this thesis, we discussed reflective metaprogramming; introduced incremental type-checking for checking type-reflective metaprograms; proposed pattern-based traits, a language feature supporting reflective metarpogramming for objects; and presented an application of pattern-based traits: PtjORM, an object-relational mapping tool with compile-time reflection.

Reflective metaprogramming is a programming paradigm for writing reusable and generic software components. Because type-reflective metaprograms have types that may depends on the result of meta evaluation, applying the MetaML-style modular type-checking approach to type-check those programs could compromise the flexibility of metaprogramming. We design a new type-checking approach, incremental type checking, that can incrementally type check code fragments as they are created and spliced together during meta computation. Incremental type-checking can detect type errors in a type-reflective metaprogram as early as possible without sacrificing flexibility.

For object-oriented programming languages, pattern-based reflection has the ability to iterate over class members and inspect their signatures. Traits offers the symmetric sum, a more expressive form of composition than inheritance. In this thesis, we investigate the combination of pattern-based reflection with traits, which is implemented into a new language feature called pattern-based traits. A pattern-based trait contains reflectors and template code. It uses reflector to obtain a class member's meta information, which are used to instantiate template code and the result can be merged into other classes or traits by the composition power of traits. Further, this thesis proposed the language features for manipulating sets of member declarations: giving them names,

manipulating their domains using set operations, and passing them as arguments to traits.

Derived from pattern-based traits, this thesis presents statement-level compile time reflection and metaprogramming, which enables a metaprogram to generate statements. In addition, this thesis introduced reified generics for pattern-based traits, which enables a pattern to iterate over any class when traits are instantiated.

Our application of pattern-based traits is PtjORM, which is an object-relational mapping tool. PtjORM uses compile-time reflection and our benchmark tests show that it has competitive runtime performance compared to the mainstream Java ORM tools.

Listed below is our future work.

(1) We plan to investigate the relation between incremental type-checking and gradual typing [53]. A brief comparison is presented in our paper [40], but it is not complete. We would like to see if there is a correspondence between the two.

(2) Currently, our incremental type-checking is implemented in an experimental functional programming language based on Garcia's calculus. We plan to extend a general-purposed functional programming with type reflection, and then implement incremental type-checking for that language.

Bibliography

[1] The boost graph library: user guide and reference manual. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] David Abrahams and Aleksey Gurtovoy. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional, 2004.

[3] Andrei Alexandrescu. Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification. Technical report, Sun Microsystems, Inc., 2007.

[5] Lennart Augustsson. Cayenne?a language with dependent types. In Proceedings of the third ACM SIGPLAN international conference on Functional programming, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM.

[6] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing software product lines using traits. In Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, pages 2096–2102, New York, NY, USA, 2010. ACM.

[7] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. A prototypical java-like language with records and traits. In Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ '10, pages 129–138, New York, NY, USA, 2010. ACM.

[8] Alan H. Borning and Daniel H. H. lngalls. Multiple inheritance in smalltalk-80. In Proceedings of the National Conference on Artificial Intelligence, 1982.

[9] Gilad Bracha and William Cook. Mixin-based inheritance. In Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM.

[10] Luca Cardelli. A semantics of multiple inheritance. In Proc. of the international symposium on Semantics of data types, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[11] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. In Proceedings of the 1993 ACM SIGMOD international conference on Management of data, SIGMOD '93, pages 12–21, New York, NY, USA, 1993. ACM.

[12] Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[13] Ole-Johan Dahl, BJørn Myhrhaug, and Kristen Nygaard. Some features of the simula 67 language. In Proceedings of the second conference on Applications of simulations, pages 29–31. Winter Simulation Conference, 1968.

[14] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. Strongaspectj: flexible and safe pointcut/advice bindings. In Proceedings of the 7th international conference on Aspect-oriented software development, AOSD '08, pages 60–71, New York, NY, USA, 2008. ACM.

[15] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A type system for reflective program generators. In Proceedings of the 4th international conference on Generative Programming and Component Engineering, GPCE'05, pages 327–341, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst., 28(2):331–388, March 2006.

[17] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In Proceedings of the 5th international conference on Generative programming and component engineering, GPCE '06, pages 275–284, New York, NY, USA, 2006. ACM.

[18] Kathleen Fisher. A typed calculus of traits. In Workshop on Foundations of Object-oriented Programming, 2004.

[19] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '98, pages 171–183, New York, NY, USA, 1998. ACM.

[20] Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM '07, pages 112–121, New York, NY, USA, 2007. ACM.

[21] Ira R. Forman and Nate Forman. Java Reflection in Action (In Action series). Manning Publications Co., Greenwich, CT, USA, 2004.

[22] Ronald Garcia. Static Computation and Reflection. PhD thesis, Indiana University, September 2008.

[23] Ronald Garcia and Andrew Lumsdaine. Toward foundations for type-reflective metaprogramming. In Proceedings of the eighth international conference on Generative programming and component engineering, GPCE '09, pages 25–34, New York, NY, USA, 2009. ACM.

[24] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM.

[25] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. In Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, PLDI '08, pages 79–89, New York, NY, USA, 2008. ACM.

[26] Shan Shan Huang and Yannis Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. ACM Trans. Program. Lang. Syst., 33(2):6:1–6:44, February 2011.

[27] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In Appear: Proc. of the European Conf. on Object-Oriented Programming (ECOOP), LNCS. Springer-Verlag, 2007.

[28] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with safegen. Sci. Comput. Program., 76(5):376–391, May 2011.

[29] Ali Ibrahim and William R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In Proceedings of the 20th European conference on Object-Oriented Programming, ECOOP'06, pages 50–73, Berlin, Heidelberg, 2006. Springer-Verlag.

[30] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst., 23(3):396–450, May 2001.

[31] Nicolai M. Josuttis. The C++ standard library: a tutorial and reference. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[32] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[33] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In ECOOP, pages 220–242, 1997.

[34] Gregor Kiczales and Jim Des Rivieres. The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, USA, 1991.

[35] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight jigsaw: A minimal core calculus for modular composition of classes. In Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, pages 244–268, Berlin, Heidelberg, 2009. Springer-Verlag.

[36] Luigi Liquori and Arnaud Spiwack. Feathertrait: A modest extension of featherweight java. ACM Trans. Program. Lang. Syst., 30(2):11:1–11:32, March 2008.

[37] M. D. McIlroy. Mass-produced software components. Proc. NATO Conf. on Software Engineering, Garmisch, Germany, 1968.

[38] Scott Meyers. Effective STL: 50 specific ways to improve your use of the standard template library. Addison-Wesley Longman Ltd., Essex, UK, UK, 2001.

[39] Weiyu Miao and Jeremy Siek. Pattern-based traits. In Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, pages 1729–1736, New York, NY, USA, 2012. ACM.

[40] Weiyu Miao and Jeremy G. Siek. Incremental type-checking for type-reflective metaprograms. In Proceedings of the ninth international conference on Generative programming and component engineering, GPCE '10, pages 167–176, New York, NY, USA, 2010. ACM.

[41] David A. Moon. Object-oriented programming with flavors. In Conference proceedings on Object-oriented programming systems, languages and applications, OOPLSA '86, pages 1–8, New York, NY, USA, 1986. ACM.

[42] Oscar Nierstrasz, Stphane Ducasse, and Nathanael Schrli. Flattening traits. Journal of Object Technology, 5:66–90, 2006.

[43] Martin Odersky and al. An overview of the scala programming language. Technical report, EPFL Lausanne, Switzerland, 2004.

[44] Benjamin C. Pierce. Types and programming languages. MIT Press, Cambridge, MA, USA, 2002.

[45] John Reppy and Aaron Turon. Metaprogramming with traits. In Proceedings of the 21st European conference on Object-Oriented Programming, ECOOP'07, pages 373–398, Berlin, Heidelberg, 2007. Springer-Verlag.

[46] Andreas Rossberg and Derek Dreyer. Mixinúp the ml module system. ACM Trans. Program. Lang. Syst., 35(1):2:1–2:84, April 2013.

[47] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In Proc. European Conference on Object-Oriented Programming, pages 248–274. Springer, 2003.

[48] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM.

[49] Jeremy Siek. The c++0x concepts effort. Generic and Indexed Programming, Lecture Notes in Computer Science, 7470:175–216, 2012.

[50] Jeremy Siek and Walid Taha. Gradual typing for objects. ECOOP 2007–Object-Oriented Programming, pages 2–27, 2007.

[51] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In In International Symposium on Computing in Object-Oriented Parallel Environments, pages 59–70, 1998.

[52] Jeremy G. Siek and Andrew Lumsdaine. Essential language support for generic programming. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI '05, pages 73–84, New York, NY, USA, 2005. ACM.

[53] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. Scheme and Functional Programming Workshop, pages 81–92, 2006.

[54] Jeremy G Siek and Philip Wadler. Threesomes, with and without blame. In ACM SIGPLAN Notices, volume 45, pages 365–376. ACM, 2010.

[55] Charles Smith and Sophia Drossopoulou. Chai: Traits for java-like languages. In Proceedings of the European conference on Object-oriented programming, pages 453–478, 2005.

[56] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In Conference proceedings on Object-oriented programming systems, languages and applications, OOPLSA '86, pages 38–45, New York, NY, USA, 1986. ACM.

[57] Bjarne Stroustrup. Multiple inheritance for C++. In Computing Systems, 1999.

[58] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM '97, pages 203–217, New York, NY, USA, 1997. ACM.

[59] Walid Mohamed Taha. Multistage programming: its theory and applications. PhD thesis, 1999. AAI9949870.

[60] Aaron Turon. Metaprogramming with traits. honors thesis, forthcoming as a university of chicago technical report, 2007.

[61] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.

Appendix A

Proof of the Correctness of the Meta-theory in This Thesis

A.1 Incremental Type-Checking

Theorem 10 (Progress)**.**

*(1) If e is closed and $\Gamma; \Psi \vdash e : \tau$, then e is a proper object code, or there is some e' such that $e \longmapsto e'$, or $e \longmapsto$ **error**.*

*(2) If e^m is closed and $\Gamma; \Psi \vdash e^m : \tau^m$, then e^m is a meta value, or there is some $e^{m\prime}$ such that $e^m \longmapsto e^{m\prime}$, or $e^m \longmapsto$ **error**.*

Proof. Straightforward induction on typing derivations.

We utilize the Theorem: Unique Decomposition presented in [23] to show that each non-proper object expression can be uniquely decomposed into a context and a redex. Similarly, each non-value meta expression can also be uniquely decomposed into a context and a redex.

(1) Case $\Gamma; \Psi \vdash \lambda x : e^m.e : \tau_1 \to \tau_2$.

e^m can be decomposed into either $E^m[r^m]$ or $E[r]$ where r^m and r are redex and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Next we focus on the meta evaluation context and the object evaluation context adopts the same mechanism. By the inversion of T-Abs1, we have P1 : $\Gamma; \Psi \vdash e^m$: **type** and P2 : $\Gamma, x : \tau_1; \Psi \vdash e : \tau_2$. From P1, we have $\Gamma; \Psi \vdash E^m[r^m]$: **type**. By induction hypothesis, we have P3 : $E^m[r^m] \longmapsto E^m[c^m]$ or P4 : $E^m[r^m] \longmapsto$ **error**. Suppose $t = \lambda x : e^m.e$. For case P3, we have $t \longmapsto t'$ where $t' = \lambda x : E^m[c^m].e$. For case P4, we have $E^m[r^m]$ is not well-typed, and then $t \longmapsto$ **error**.

(2) Case $\Gamma; \Psi \vdash \lambda x : \tau^s.e : \tau^s \to \tau$.

If e is a proper code, then $\lambda x : \tau^s.e$ is also a proper code. If e is not a proper code, then e can be decomposed into either $E^m[r^m]$ or $E[r]$ where r^m and r are redex and suppose we have the reductions rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Next we focus on the meta evaluation context and the object evaluation context adopts the same mechanism. By the inversion of T-Abs2, we have P1 : $\Gamma, x : \tau^s; \Psi \vdash e : \tau$. From P1, we have $\Gamma, x : \tau^s; \Psi \vdash E^m[r^m] : \tau$. By induction hypothesis, we have P2 : $E^m[r^m] \longmapsto E^m[c^m]$ or P3 : $E^m[r^m] \longmapsto \textbf{error}$. Suppose $t = \lambda x : \tau^s.e$. For case P2, we have $t \longmapsto t'$ where $t' = \lambda x : \tau^s.E^m[r^m].e$. For case P3, we have $E^m[r^m]$ is not well-typed, and then $t \longmapsto \textbf{error}$.

(3) Case $\Gamma; \Psi \vdash \sim e^m : [\overline{\tau}/\overline{\alpha}]\tau_1$.

By the inversion of T-Esp, we have P1 : $\Gamma; \Psi \vdash e^m : \exists\overline{\alpha}.(\textbf{code}\ \tau_1)$. If e^m is a value, then according to the Canonical Form lemma (omitted in this paper), e^m is in the form of $\prec e^o \succ$ for some e^o. According to the reduction rule for splice, we have $\sim \prec e^o \succ \longmapsto e^o$. If e^m is not a value, then e^m can be decomposed into either $E^m[r^m]$ or $E[r]$ where r^m and r are redex and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Next we focus on the meta evaluation context and the object evaluation context adopts the same mechanism. From P1, we have $\Gamma; \Psi \vdash E^m[r^m] : \exists\overline{\alpha}.(\textbf{code}\ \tau_1)$. By induction hypothesis, we have P2 : $E^m[r^m] \longmapsto E^m[c^m]$ or P3 : $E^m[r^m] \longmapsto \textbf{error}$. Suppose $t = \sim e^m$. For case P2, we have $t \longmapsto t'$ where $t' = \sim E^m[c^m]$. For case P3, we have $E^m[r^m]$ is not well-typed, and then $t \longmapsto \textbf{error}$.

(4) Case $\Gamma; \Psi \vdash \textbf{typeof}\ e^m : \textbf{type}$.

By the inversion of M-Typeof, we have P1 : $\Gamma; \Psi \vdash e^m : \exists\overline{\alpha}.(\textbf{code}\ \tau)$. If e^m is a value, then according to the Canonical Form lemma, e^m is in the form of $\prec e^o \succ$ for some e^o. According to the reduction rule for typeof, we have $\textbf{typeof} \prec e^o \succ \longmapsto \tau'$ for some τ'. If e^m is not a value, then e^m can be decomposed into either $E^m[r^m]$ or $E[r]$ where r^m and r are redex and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Next we focus on the meta evaluation context and the object evaluation context adopts the same mechanism. From P1, we have $\Gamma; \Psi \vdash E^m[r^m] : \exists\overline{\alpha}.(\textbf{code}\ \tau)$. By induction hypothesis, we have P2 : $E^m[r^m] \longrightarrow E^m[c^m]$ or P3 : $E^m[r^m] \longrightarrow \textbf{error}$. Suppose

$t = \textbf{typeof}\ e^m$. For case P2, we have $t \longmapsto t'$ where $t' = \textbf{typeof}\ E^m[c^m]$. For case P3, we have $E^m[r^m]$ is not well-typed, then $t \longmapsto \textbf{error}$.

(5) Case $\Gamma; \Psi \vdash \prec e \succ : \exists \overline{\alpha}.(\textbf{code}\ \tau)$.

By the inversion of M-Code, we have P1 : $\Gamma, \overline{\alpha}; \Psi \vdash e : \tau$. If e is a proper code, then $\prec e \succ$ is a meta value. If e is not a proper code, then e can be decomposed into $E^m[r^m]$ or $E[r]$ where r^m and r are redex and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Next we focus on the meta evaluation context and the object evaluation context adopts the same mechanism. From P1, we have $\Gamma, \overline{\alpha}; \Psi \vdash E^m[r^m] : \tau$. By induction hypothesis, we have P2 : $E^m[r^m] \longrightarrow E^m[c^m]$ or P3 : $E^m[r^m] \longrightarrow \textbf{error}$. Suppose $t = \prec e \succ$. For case P2, we have $t \longmapsto t'$ where $t' = \prec E^m[c^m] \succ$. For case P3, we have $E^m[r^m]$ is not well-typed, then $t \longmapsto \textbf{error}$.

(6) Case $\Gamma; \Psi \vdash e_1^m\ e_2^m : \tau_3^m$.

By the inversion of M-App, we have P1 : $\Gamma; \Psi \vdash e_1^m : \tau_1^m \to \tau_3^m$ and P2 : $\Gamma; \Psi \vdash e_2^m : \tau_2^m$ where $\tau_1^m \sim \tau_2^m$. If e_1^m is not a value, then it can be decomposed into $E^m[r^m]$ or $E[r]$ where r^m and r are redex and and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Next we focus on the meta evaluation context and the object evaluation context adopts the same mechanism. From P1, we have $\Gamma; \Psi \vdash E^m[r^m] : \tau_1^m \to \tau_2^m$. By induction hypothesis, we have P2 : $E^m[r^m] \longrightarrow E^m[c^m]$ or P3 : $E^m[r^m] \longrightarrow \textbf{error}$. Suppose $t = e_1^m\ e_2^m$. For case P2, we have $t \longmapsto t'$ where $t' = E^m[c^m]\ e_2^m$. For case P3, we have $E^m[r^m]$ is not well-typed, then $t \longmapsto \textbf{error}$. If e_1^m is a value but e_2^m is not a value, then the proof is similar to the previous case. If e_1^m and e_2^m are both values, then according to the Canonical Form lemma, e_1^m has the form $\lambda x : \tau^m.e_{11}^m$. So, we have the reduction rule $(\lambda x : \tau^m.e_{11}^m)e_2^m \longmapsto [e_2^m/x]e_{11}^m$.

(7) Case $\Gamma; \Psi \vdash \textbf{if}\ e_1^m\ \textbf{then}\ e_2^m\ \textbf{else}\ e_3^m : \tau^m$.

Similar to case (6). □

Lemma 14 (Transitivity)**.** *If $\tau_1^m <:_n \tau_2^m$ and $\tau_2^m <:_n \tau_3^m$, then $\tau_1^m <:_n \tau_3^m$*

Proof. Induction on naive subtyping rules. □

Lemma 15. *If* $\tau_1^m <:_n \tau_2^m$, $\tau_3^m <:_n \tau_4^m$, $\tau_{13}^m = \tau_1^m \vee_n \tau_3^m$, *and* $\tau_{24}^m = \tau_2^m \vee_n \tau_4^m$, *then* $\tau_{13}^m <:_n \tau_{24}^m$

Proof. Prove by the transitivity property of naive subtyping. □

Theorem 11 (Preservation)**.**

(1) If $\Gamma; \Psi \vdash e : \tau$ *and* $e \longmapsto e'$, *then* $\Gamma; \Psi \vdash e' : \tau'$.

(2) If $\Gamma; \Psi \vdash e^m : \tau^m$ *and* $e^m \longmapsto e^{m\prime}$, *then* $\Gamma; \Psi \vdash e^{m\prime} : \tau^{m\prime}$.

Proof. Straightforward induction on typing derivations.

(1) Case $\Gamma; \Psi \vdash \lambda x : e^m.e : \tau_1 \to \tau_2$.

Suppose $t = \lambda x : e^m.e$, which is not a value. If e^m is not a value, then it can be decomposed into $E^m[r^m]$ or $E[r]$ where r^m and r are redex and and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Here we focus on the meta evaluation context while the object evaluation context adopts the same mechanism. From premise that $t \longmapsto t'$, we have $t' = \lambda x : E^m[c^m].e$. By the inversion of T-Abs1, we have P1 : $\Gamma; \Psi \vdash e^m$: **type**, that is $\Gamma; \Psi \vdash E^m[r^m]$: **type**, and P2 : $\Gamma, x : \tau_1; \Psi \vdash e : \tau_2$. By induction hypothesis, from P1 we have P3 : $\Gamma; \Psi \vdash E^m[c^m]$: **type**. By T-Abs1, from P2 and P3 we have $\Gamma; \Psi \vdash \lambda x : E^m[c^m].e : \tau_1 \to \tau_2$, that is $\Gamma; \Psi \vdash t' : \tau_1 \to \tau_2$.

(2) Case $\Gamma; \Psi \vdash \lambda x : \tau^s.e : \tau^s \to \tau$.

Suppose $t = \lambda x : \tau^s.e$, which is not a value, then e is not a proper code and it can be decomposed into $E^m[r^m]$ or $E[r]$ where r^m and r are redex and and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Here we focus on the meta evaluation context while the object evaluation context adopts the same mechanism. From premise that $t \longmapsto t'$, we have $t' = \lambda x : \tau^s.E^m[c^m]$. By the inversion of T-Abs2, we have P1 : $\Gamma, x : \tau^s; \Psi \vdash e : \tau$ and by induction hypothesis, we have $\Gamma, x : \tau^s; \Psi \vdash E^m[c^m] : \phi(\tau)$. By Abs2, we have $\Gamma; \Psi \vdash \lambda x : \tau^s.E^m[c^m] : \tau^s \to \phi(\tau)$, that is $\Gamma; \Psi \vdash t' : \phi(\tau^s \to \tau)$.

(3) Case $\Gamma; \Psi \vdash \sim e^m : [\overline{\tau}/\overline{\alpha}]\tau_1$.

Suppose $t = \sim e^m$, which is not a value, then e^m is not a value and it can be decomposed into $E^m[r^m]$ or $E[r]$ where r^m and r are redex and and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow$

c. Here we focus on the meta evaluation context while the object evaluation context adopts the same mechanism. By the inversion of T-Esp, we have P1 : $\Gamma; \Psi \vdash e^m : \exists\overline{\alpha}.(\mathbf{code}\ \tau_1)$. By induction hypothesis, we have P2 : $\Gamma; \Psi \vdash E^m[c^m] : \exists\overline{\beta}.(\mathbf{code}\ \tau_2)$ where $\exists\overline{\beta}.(\mathbf{code}\ \tau_2) <:_n \exists\overline{\alpha}.(\mathbf{code}\ \tau_1)$ and $\tau_2 = \phi(\tau_1)$. By T-Esp, we have $\Gamma; \Psi \vdash \sim e^m : [\overline{\tau}/\overline{\beta}]\tau_2$, that is $\Gamma; \Psi \vdash \sim e^m : [\overline{\tau}/\overline{\beta}](\phi(\tau_1))$, that is $\phi([\overline{\tau}/\overline{\beta}]\tau_1)$.

(4) Case $\Gamma; \Psi \vdash \mathbf{typeof}\ e^m : \mathbf{type}$.

Omitted.

(5) Case $\Gamma; \Psi \vdash \prec e \succ : \exists\overline{\alpha}.(\mathbf{code}\ \tau)$.

Suppose $t = \prec e \succ$, which is not a value, then e^m is not a value and it can be decomposed into $E^m[r^m]$ or $E[r]$ where r^m and r are redex and and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Here we focus on the meta evaluation context while the object evaluation context adopts the same mechanism. By the inversion of M-Code, we have $\Gamma, \overline{\alpha}; \Psi \vdash e : \tau$. By induction hypothesis, we have $\Gamma, \overline{\alpha}; \Psi \vdash E^m[c^m] : \phi(\tau)$. By M-Code, we have $\Gamma; \Psi \vdash \prec e \succ : \exists\overline{\alpha}.(\mathbf{code}\ \phi(\tau))$ where $\exists\overline{\alpha}.(\mathbf{code}\ \phi(\tau)) <:_n \exists\overline{\alpha}.(\mathbf{code}\ \tau)$.

(6) Case $\Gamma; \Psi \vdash e_1^m\ e_2^m : \tau_3^m$.

Suppose $t = e_1^m\ e_2^m$, which is not a value, then case one : e_1^m is not value and it can be decomposed into $E^m[r^m]$ or $E[r]$ where r^m and r are redex and and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Here we focus on the meta evaluation context while the object evaluation context adopts the same mechanism. By the inversion of M-App, we have $\Gamma; \Psi \vdash e_1^m : \tau_1^m \to \tau_3^m$. By induction hypothesis, we have $\Gamma; \Psi \vdash E^m[c^m] : \tau_4^m \to \tau_5^m$ where $\tau_4^m <:_n \tau_1^m$, $\tau_5^m <:_n \tau_3^m$, and $\tau_4^m \sim \tau_2^m$ (inversion of the simple-step evaluation). So, by M-App we have $\Gamma; \Psi \vdash E^m[c^m]\ e_2^m : \tau_5^m$.

(7) Case $\Gamma; \Psi \vdash \mathbf{if}\ e_1^m\ \mathbf{then}\ e_2^m\ \mathbf{else}\ e_3^m : \tau^m$.

Suppose $t = \mathbf{if}\ e_1^m\ \mathbf{then}\ e_2^m\ \mathbf{else}\ e_3^m$, which is not a value, for the case when e_1^m is not a value, it is easy to prove. Case two : e_2^m is not a value, it can be decomposed into $E^m[r^m]$ or $E[r]$ where r^m and r are redex and and suppose we have the reduction rules $r^m \longrightarrow c^m$ and $r \longrightarrow c$. Here we focus on the meta evaluation context while the object evaluation context adopts the same mechanism. By the inversion of M-If, we have $\Gamma; \Psi \vdash e_2^m : \tau_2^m$ and $\Gamma; \Psi \vdash e_3^m : \tau_3^m$. By induction

hypothesis, we have $\Gamma; \Psi \vdash E^m[c^m] : \tau_2^{m\prime}$ where $\tau_2^{m\prime} <:_n \tau_2^m$. From the above lemma, we have $(\tau_2^{m\prime} \vee_n \tau_3^m) <:_n (\tau_2^m \vee_n \tau_3^m)$. □

Theorem 12 (Type Safety)**.**

(1) If $\Gamma; \Psi \vdash e : \tau$ *and* $e \longmapsto^* e'$*, then* $\Gamma; \Psi \vdash e' : \tau'$*, and* e' *is a proper object code, or* $\exists e''.e' \longmapsto e''$*, or* $e' \longmapsto$ **error**.

(2) If $\Gamma; \Psi \vdash e^m : \tau^m$ *and* $e^m \longmapsto^* e^{m\prime}$*, then* $\Gamma; \Psi \vdash e^{m\prime} : \tau^{m\prime}$*, and* $e^{m\prime}$ *is a meta value, or* $\exists e^{m\prime\prime}.e^{m\prime} \longmapsto e^{m\prime\prime}$*, or* $e^{m\prime} \longmapsto$ **error**.

Proof. Using Progress and Preservation. □

A.2 Unification

Lemma 16 (Substitution Preserves $\hookrightarrow$)**.** *For some* θ *and* $[\sigma/\alpha]$ *where* $\alpha \notin vars(\sigma)$*, if* $\theta \hookrightarrow C$*, then* $[\sigma/\alpha] \circ \theta \hookrightarrow [\sigma/\alpha]C \cup \{\alpha = \sigma\}$.

Proof. From $\theta \hookrightarrow C$, we have $[\sigma/\alpha]; \theta \hookrightarrow C \cup \{\alpha = \sigma\}$. From $\theta \hookrightarrow C$, we have $[\sigma/\alpha]\theta \hookrightarrow [\sigma/\alpha]C$; further we have $[\sigma/\alpha]; [\sigma/\alpha](\theta) \hookrightarrow [\sigma/\alpha]C \cup \{\alpha = \sigma\}$. From the definition of $\circ$, we obtain that $[\sigma/\alpha]; [\sigma/\alpha](\theta)$ is equal to $[\sigma/\alpha] \circ \theta$. So $[\sigma/\alpha] \circ \theta \hookrightarrow [\sigma/\alpha]C \cup \{\alpha = \sigma\}$. Done. □

Theorem 13 (Correctness of Unification Function *unify_eq*)**.** *For any constraint set* C *and substitution* θ*, suppose* C *has no type consistency constraints and* $\theta \hookrightarrow C'$*. We have* $unify(C) \circ \theta \hookrightarrow unify\_eq(C, C')$.

Proof. (1) if $C = \{\}$, then $unify(\{\}) \circ \theta = \theta$, and $unify\_eq(\{\}, C') = C'$; so $unify(\{\}) \circ \theta \hookrightarrow unify\_eq(\{\}, C')$.

(2) Suppose $unify(C) \circ \theta \hookrightarrow unify\_eq(C, C')$. We need to prove $unify(C \cup \{\sigma = \rho\}) \circ \theta \hookrightarrow unify\_eq(C \cup \{\sigma = \rho\}, C')$. Prove by cases on $\sigma = \rho$: (2.1) If σ is equal to ρ, then according to the definition of *unify*, $unify(C \cup \{\sigma = \rho\}) \circ \theta = unify(C) \circ \theta$; and according to the definition of *unify_eq*, $unify\_eq(C \cup \{\sigma = \rho\}, C') = unify\_eq(C, C')$, therefore $unify(C \cup \{\sigma = \rho\}) \circ \theta \hookrightarrow unify\_eq(C \cup \{\sigma =$

$\rho\}, C')$. (2.2) If $\sigma = \sigma_1 \to \sigma_2$ and $\rho = \rho_1 \to \rho_2$, then then according to the definition of *unify*, $\mathit{unify}(C \cup \{\sigma_1 \to \sigma_2 = \rho_1 \to \rho_2\}) \circ \theta = \mathit{unify}(C \cup \{\sigma_1 = \rho_1, \sigma_2 = \rho_2\}) \circ \theta$; and according to the definition of *unify_eq*, $\mathit{unify\_eq}(C \cup \{\sigma_1 \to \sigma_2 = \rho_1 \to \rho_2\}, C') = \mathit{unify\_eq}(C \cup \{\sigma_1 = \rho_1, \sigma_2 = \rho_2\}, C')$. From the hypothesis that $\mathit{unify}(C) \circ \theta \hookrightarrow \mathit{unify\_eq}(C, C')$, we have $\mathit{unify}(C \cup \{\sigma_1 = \rho_1, \sigma_2 = \rho_2\}) \circ \theta \hookrightarrow \mathit{unify\_eq}(C \cup \{\sigma_1 = \rho_1, \sigma_2 = \rho_2\}, C')$. (2.3) If $\sigma = \alpha$ and $\alpha \notin \mathit{vars}(\rho)$, then according to the definition of *unify*, $\mathit{unify}(C \cup \{\alpha = \rho\}) \circ \theta = \mathit{unify}(C) \circ [\rho/\alpha] \circ \theta$; and according to the definition of *unify_eq*, $\mathit{unify\_eq}(C \cup \{\alpha = \rho\}\}, C') = \mathit{unify\_eq}(C, [\rho/\alpha]C' \cup \{\alpha = \rho\})$. From the lemma (Substitution Preserves $\hookrightarrow$), we have $[\rho/\alpha] \circ \theta \hookrightarrow [\rho/\alpha]C' \cup \{\alpha = \rho\}$. Therefore, $\mathit{unify}(C) \circ [\rho/\alpha] \circ \theta \hookrightarrow \mathit{unify\_eq}(C, [\rho/\alpha]C' \cup \{\alpha = \rho\})$. (2.4) If $\rho = \alpha$, the case is similar to case (2.3). Done. □

A.3 Pattern-based Traits

Lemma 17 (Expression Typing Equivalence)**.** *Suppose* $\Gamma \vdash^{\mathrm{FJ}} \mathtt{e}^o : \mathtt{C}$ *is expression typing in FJ. Suppose some expression* $\mathtt{e}^o$ *is inside class* C*. If* $\mathtt{e}^o$ *is well-typed via our type system:* $\Gamma \vdash \mathtt{e}^o : \tau$*, then* $\mathtt{e}^o$ *is also well-typed via the type system of FJ, that is there exists some* Γ' *such that* $\Gamma' = [\![\Gamma]\!]$ *and* $\Gamma' \vdash^{\mathrm{FJ}} \mathtt{e}^o : [\![\tau]\!]$*, where notation* $[\![\cdot]\!]$ *converts an object type into the nominal part of the object type, or converts the object types inside a* Γ *into the nominal parts of those object types.*

Proof. The only difference between the expression typing rules in FJ and our expression typing rules is class member type lookup. The type system of FJ uses function *mtype* and function *fields* to lookup the type of a method and the types of fields respectively. In our type system, we lookup the type of a class member via the structural type of a class. According to Rule T-Mc in Figure 5.9, the structural type of a class is calculated by $\Delta_{sub} \ni \Delta_{base}$. (i.e. it merges the members in the super classes and in the current class; and a method in the current class overrides a method in a superclass if the two methods have the same name.) Suppose the structural type of C is σ. It is straightforward that $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T}$ if and only if $\mathtt{m} : \overline{\mathtt{T}} \to \mathtt{T} \in \sigma$. It is also straightforward that $\mathit{fields}(\mathtt{C}) = \mathtt{T}_1\ \mathtt{f}_1, ..., \mathtt{T}_n\ \mathtt{f}_n$ if and only if for each $i \in 1, ..., n$ such that $\mathtt{f}_i : \mathtt{T}_i \in \sigma$. Except the

member type lookup, our expression typing rules are similar to the expression typing rules for FJ. Done. □

Lemma 18 (Method Typing Equivalence)**.** *Suppose some method* $\mathtt{M}^o$ *is defined or inlined into class* C*. If* $\mathtt{M}^o$ *is well-defined via our type system:* $\Delta;\cdot \vdash \mathtt{M}^o$ ok in C*, then* $\mathtt{M}^o$ *is also well-defined via the type system of FJ.*

Proof. In FJ, the rule for typing method is defined like the following:

$$\frac{\begin{array}{c}\overline{\mathtt{c}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash^{\mathrm{FJ}} \mathtt{e} : \mathtt{C}_1 \qquad\qquad \mathtt{C}_1 <: \mathtt{C}_0 \\ \mathtt{class\ C\ extends\ D\ \{\ ...\ \}} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0, \text{ then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}\ \}\ \mathtt{OK\ IN\ C}}$$

Our rule of typing method is Rule T-Md2 in Figure 5.8. So, if $\Delta;\cdot \vdash \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}$ ok in C, then by the inversion of Rule T-Md2, we have $\overline{\mathtt{x}} : \overline{\mathtt{C} \diamond \sigma}, \mathtt{this} : \mathtt{C} \diamond \sigma_0 \vdash \mathtt{C}_1 \diamond \sigma_1$ and $\mathtt{C}_1 <: \mathtt{C}_0$. By lemma 17, from $\overline{\mathtt{x}} : \overline{\mathtt{C} \diamond \sigma}, \mathtt{this} : \mathtt{C} \diamond \sigma_0 \vdash \mathtt{C}_1 \diamond \sigma_1$, we have $\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{C}_1$. Because method overriding is checked in Rule T-Mc in Figure 5.9. So, by the FJ's typing rule of method, we have $\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}\ \mathtt{OK\ IN\ C}$. Done. □

Definition 2 (Type Substitution over Object Types)**.** *Suppose* $\mathtt{N} \diamond \sigma$ *is well-formed under the typing environment* $\Gamma, \mathtt{X}, \Gamma'$*, and suppose* $\Gamma \vdash \mathtt{T}$ ok*.* $[\mathtt{T}/\mathtt{X}](\mathtt{N} \diamond \sigma) = \mathtt{N}' \diamond \sigma'$ *such that* $[\mathtt{T}/\mathtt{X}]\mathtt{N} = \mathtt{N}'$ *and* $[\mathtt{T}/\mathtt{X}]\sigma \subseteq \sigma'$*.*

Lemma 19 (Type Substitution Preserves Subtyping)**.** *Suppose* N *and* $\mathtt{N}'$ *are well-formed under the typing environment* $\Gamma, \mathtt{X}, \Gamma'$*, and suppose* $\Gamma \vdash \mathtt{T}$ ok*. If* $\mathtt{N} <: \mathtt{N}'$*, then* $[\mathtt{T}/\mathtt{X}]\mathtt{N} <: [\mathtt{T}/\mathtt{X}]\mathtt{N}'$*.*

Proof. Straightforward induction on subtyping derivations. □

Lemma 20 (Type Substitution Preserves Expression Typing)**.** *If* $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{e} : \tau$ *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{e} : [\mathtt{T}/\mathtt{X}]\tau$*.*

Proof. Straightforward induction on typing derivations.

Case $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{x} : \tau$. By the inversion of Rule T-Var in Figure 5.8, we have $\mathtt{x} : \tau \in \Gamma, \mathtt{X}, \Gamma'$. Therefore, we have $\mathtt{x} : [\mathtt{T}/\mathtt{X}]\tau \in \Gamma; [\mathtt{T}/\mathtt{X}]\Gamma'$. By Rule T-Var, we have $\Gamma; [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{x} : [\mathtt{T}/\mathtt{X}]\tau$.

Case $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{e.f} : \mathtt{U} \diamond \sigma$. By the inversion of Rule T-FdE in Figure 5.8 and by the definition of *objType*, we have $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_0$ and $\mathtt{f} = \mathtt{U} \in \sigma_0$. From $\mathtt{f} = \mathtt{U} \in \sigma_0$, we have $\mathtt{f} = [\mathtt{T}/\mathtt{X}]\mathtt{U} \in [\mathtt{T}/\mathtt{X}]\sigma_0$. From $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_0$ and by the induction hypothesis, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{e} : ([\mathtt{T}/\mathtt{X}]\mathtt{N}) \diamond \sigma_1$ where $[\mathtt{T}/\mathtt{X}]\sigma_0 \subseteq \sigma_1$. By Rule T-FdE, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}](\mathtt{e.f}) : \mathit{objType}([\mathtt{T}/\mathtt{X}]\mathtt{U})$.

Case $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{U} \diamond \sigma$. By the inversion of Rule T-MdE, we have $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_0$; $\mathtt{m} = \overline{\mathtt{S}} \to \mathtt{U} \in \sigma_0$; $\Gamma, \mathtt{X}, \Gamma' \vdash \overline{\mathtt{e}} : \overline{\mathtt{N}' \diamond \sigma'}$; and $\overline{\mathtt{N}'} <: \overline{\mathtt{S}}$. By the induction hypothesis, from $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_0$, we have $\Gamma; [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]e : [\mathtt{T}/\mathtt{X}](\mathtt{N}) \diamond \sigma_1$ where $[\mathtt{T}/\mathtt{X}]\sigma_0 \subseteq \sigma_1$. From $\mathtt{m} = \overline{\mathtt{S}} \to \mathtt{U} \in \sigma_0$, we have $\mathtt{m} = \overline{[\mathtt{T}/\mathtt{X}]\mathtt{S}} \to [\mathtt{T}/\mathtt{X}]\mathtt{U} \in [\mathtt{T}/\mathtt{X}]\sigma_0$. By the induction hypothesis, from $\Gamma, \mathtt{X}, \Gamma' \vdash \overline{\mathtt{e}} : \overline{\mathtt{N}' \diamond \sigma'}$, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{e}} : \overline{[\mathtt{T}/\mathtt{X}]\mathtt{N}' \diamond \sigma''}$ where for each σ''_i, we have $[\mathtt{T}/\mathtt{X}]\sigma'_i \subseteq \sigma''_i$. By lemma 19, from $\overline{\mathtt{N}'} <: \overline{\mathtt{S}}$, we have $\overline{[\mathtt{T}/\mathtt{X}]\mathtt{N}'} <: \overline{[\mathtt{T}/\mathtt{X}]\mathtt{S}}$. By Rule T-MdE, we finally have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}](\mathtt{e.m}(\overline{\mathtt{e}})) : \mathit{objType}([\mathtt{T}/\mathtt{X}]\mathtt{U})$.

Case $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C} \diamond \sigma$. By the inversion of Rule T-New, we have $\mathtt{C}$ is well-defined; $\mathtt{C}(\overline{\mathtt{U}}\ \mathtt{f})\ \{\ldots\} \in \mathtt{MC}$; $\Gamma, \mathtt{X}, \Gamma' \vdash \overline{\mathtt{e}} : \overline{\mathtt{N} \diamond \sigma'}$; and $\overline{\mathtt{N}} <: \overline{\mathtt{U}}$. By the induction hypothesis, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{e}} : \overline{[\mathtt{T}/\mathtt{X}](\mathtt{N}) \diamond \sigma''}$ where for each σ''_i, we have $[\mathtt{T}/\mathtt{X}]\sigma'_i \subseteq \sigma''_i$. By lemma 19, from $\overline{\mathtt{N}} <: \overline{\mathtt{T}}$, we have $\overline{[\mathtt{T}/\mathtt{X}]\mathtt{N}} <: \overline{[\mathtt{T}/\mathtt{X}]\mathtt{U}}$. By Rule T-New, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}](\mathtt{new\ C}(\overline{\mathtt{e}})) : \mathit{objType}([\mathtt{T}/\mathtt{X}]\mathtt{C})$.

Case $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{(T)e} : \tau$. Similar to the second case.

Done. □

Definition 3 (Name Substitution over Δ)**.** *Suppose there is no name conflict during name substitution. Suppose* $\Delta = [\mathtt{l}_i \mapsto \delta_i]^{i \in 1 \ldots n}$. *For some* $\mathtt{l}_0$ *and* η, $[\mathtt{l}_0/\eta]\Delta = [[\mathtt{l}_0/\eta](\mathtt{l}_i) \mapsto \delta_i]^{i \in 1 \ldots n}$.

Definition 4 (Name Substitution over Object Types)**.** *Suppose there is no name conflict during name substitution. Suppose* $\sigma = \{\mathtt{l}_i \mapsto \delta_i\}^{i \in 1 \ldots n}$; $\mathtt{N} \diamond \sigma$ *is well-formed under the typing environment* Γ, η, Γ'; *and* $\Gamma \vdash \mathtt{l}$ ok. $[\mathtt{l}/\eta](\mathtt{N} \diamond \sigma) = \mathtt{N} \diamond \{[\mathtt{l}/\eta](\mathtt{l}_i) \mapsto \delta_i\}^{i \in 1 \ldots n}$.

Lemma 21 (Name Substitution Preserves Expression Typing)**.** *Suppose there is no name conflict during name substitution. If* $\Gamma, \eta, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma$ *and* $\Gamma \vdash \mathtt{l}$ ok, *then* $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{e} : \mathtt{N} \diamond [\mathtt{l}/\eta]\sigma$.

Proof. Straightforward induction on typing derivations.

Case $\Gamma, \eta, \Gamma' \vdash \mathtt{x} : \tau$. By the inversion of Rule T-Var in Figure 5.8, we have $\mathtt{x} : \tau \in \Gamma, \eta, \Gamma'$. Therefore, we have $\mathtt{x} : [\mathtt{l}/\eta]\tau \in \Gamma; [\mathtt{l}/\eta]\Gamma'$. By Rule T-Var, we have $\Gamma; [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{x} : [\mathtt{l}/\eta]\tau$.

Case $\Gamma, \eta, \Gamma' \vdash \mathtt{e.f} : \mathtt{U} \diamond \sigma$. By the inversion of Rule T-FdE, we have $\Gamma, \eta, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_0$ and $\mathtt{f} = \mathtt{U} \in \sigma_0$. From $\mathtt{f} = \mathtt{U} \in \sigma_0$, we have $([\mathtt{l}/\eta]\mathtt{f}) = \mathtt{U} \in [\mathtt{l}/\eta]\sigma_0$. By the induction hypothesis, from $\Gamma, \eta, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_0$, we have $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{e} : \mathtt{N} \diamond [\mathtt{l}/\eta]\sigma_0$. By Rule T-FdE, we have $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{T}/\eta](\mathtt{e.f}) : \mathtt{U} \diamond \sigma$. Therefore, $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{T}/\eta](\mathtt{e.f}) : \mathtt{U} \diamond [\mathtt{T}/\eta]\sigma$.

Case $\Gamma, \eta, \Gamma' \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{U} \diamond \sigma$. By the inversion of Rule T-MdE, we have $\Gamma, \eta, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_0$; $\mathtt{m} = \overline{\mathtt{S}} \to \mathtt{U} \in \sigma_0$; $\Gamma, \eta, \Gamma' \vdash \overline{\mathtt{e}} : \overline{\mathtt{N}' \diamond \sigma'}$; and $\overline{\mathtt{N}'} <: \overline{\mathtt{S}}$. By the induction hypothesis, from $\Gamma, \eta, \Gamma' \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_0$, we have $\Gamma; [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]e : \mathtt{N} \diamond [\mathtt{l}/\eta]\sigma_0$. From $\mathtt{m} = \overline{\mathtt{S}} \to \mathtt{U} \in \sigma_0$, we have $([\mathtt{l}/\eta]\mathtt{m}) = \mathtt{S} \to [\mathtt{l}/\eta]\mathtt{U} \in [\mathtt{l}/\eta]\sigma_0$. By the induction hypothesis, from $\Gamma, \eta, \Gamma' \vdash \overline{\mathtt{e}} : \overline{\mathtt{N}' \diamond \sigma'}$, we have $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash \overline{[\mathtt{l}/\eta]\mathtt{e}} : \overline{\mathtt{N}' \diamond [\mathtt{l}/\eta]\sigma'}$. By Rule T-MdE, we have $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{U} \diamond \sigma$. Therefore, $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{U} \diamond [\mathtt{l}/\eta]\sigma$.

Case $\Gamma, \eta, \Gamma' \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C} \diamond \sigma$. By the inversion of Rule T-New, we have $\mathtt{C}$ is well-defined; $\mathtt{C}(\overline{\mathtt{U}}\ \mathtt{f})\ \{\ldots\} \in \mathtt{MC}$; $\Gamma, \eta, \Gamma' \vdash \overline{\mathtt{e}} : \overline{\mathtt{N} \diamond \sigma'}$; and $\overline{\mathtt{N}} <: \overline{\mathtt{U}}$. By the induction hypothesis, we have $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash \overline{[\mathtt{l}/\eta]\mathtt{e}} : \overline{\mathtt{N} \diamond [\mathtt{l}/\eta]\sigma'}$. By Rule T-New, we have $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta](\mathtt{new\ C}(\overline{\mathtt{e}})) : \mathtt{C} \diamond \sigma$. Therefore, $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta](\mathtt{new\ C}(\overline{\mathtt{e}})) : \mathtt{C} \diamond [\mathtt{l}/\eta]\sigma$.

Case $\Gamma, \eta, \Gamma' \vdash \mathtt{(T)e} : \tau$. Similar to the second case.

Done. □

Lemma 22 (Type Substitution Preserves Method Typing)**.** *If* $\Delta; \Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{M}$ ok in $\mathtt{N}$ *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $[\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{M}$ ok in $\mathtt{N}$.

Proof. If $\Delta; \Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{U\ m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}$ ok in $\mathtt{thisType}$, then by the inversion of Rule T-Md1, we have P1: $\Gamma, \mathtt{X}, \Gamma' \vdash \overline{\mathtt{U}}, \mathtt{U}$ ok; P2 : $\Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{m}$ ok; P3 : $\Gamma, \mathtt{X}, \Gamma', \overline{\mathtt{x}} : \overline{\mathtt{U} \diamond \sigma}, \mathtt{thisType}, \mathtt{this} : \mathtt{thisType} \diamond \sigma_0 \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_1$; and P4 : $\mathtt{N} <: \mathtt{U}$. From P1, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{U}}, [\mathtt{T}/\mathtt{X}]\mathtt{U}$ ok. (It is easy to prove that type substitution preserves well-formed types.) From P2, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash \mathtt{m}$ ok. By lemma 20, from P3, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma', \overline{\mathtt{x}} : \overline{[\mathtt{T}/\mathtt{X}]\mathtt{U} \diamond \sigma'}, \mathtt{thisType}, \mathtt{this} : \mathtt{thisType} \diamond \sigma_0{}' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{e} : [\mathtt{T}/\mathtt{X}]\mathtt{N} \diamond \sigma_1{}'$, where $\overline{[\mathtt{T}/\mathtt{X}]\sigma} \subseteq \overline{\sigma'}$, $[\mathtt{T}/\mathtt{X}]\sigma_0 \subseteq \sigma_0{}'$, and $[\mathtt{T}/\mathtt{X}]\sigma_1 \subseteq \sigma_1{}'$. By

lemma 19, from P4 we have $[\mathtt{T}/\mathtt{X}]\mathtt{N} <: [\mathtt{T}/\mathtt{X}]\mathtt{U}$. Therefore, by Rule T-Md1, we have $\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash$ `[T/X](U m(`$\overline{\mathtt{U}}$ $\overline{\mathtt{x}}$`){ return e; })` ok in `thisType`. □

Lemma 23 (Name Substitution Preserves Method Typing)**.** *Suppose name generation/substitution does not create conflicts. If* $\Delta; \Gamma, \eta, \Gamma' \vdash \mathtt{M}$ ok in $\mathtt{N}$ *and* $\Gamma \vdash \mathtt{l}$ ok*, then* $[\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{M}$ ok in $\mathtt{N}$.

Proof. If $\Delta; \Gamma, \eta, \Gamma' \vdash$ `U m(`$\overline{\mathtt{U}}$ $\overline{\mathtt{x}}$`){ return e; }` ok in `thisType`, then by the inversion of Rule T-Md1, we have P1: $\Gamma, \eta, \Gamma' \vdash \overline{\mathtt{U}}, \mathtt{U}$ ok; P2 : $\Gamma, \eta, \Gamma' \vdash \mathtt{m}$ ok; P3 : $\Gamma, \eta, \Gamma', \overline{\mathtt{x}} : \overline{\mathtt{U} \diamond \sigma}, \mathtt{thisType}, \mathtt{this} :$ $\mathtt{thisType} \diamond \sigma_0 \vdash \mathtt{e} : \mathtt{N} \diamond \sigma_1$; and P4 : $\mathtt{N} <: \mathtt{U}$. From P1, we have $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash \overline{\mathtt{U}}, \mathtt{U}$ ok. From P2, we have $\Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{m}$ ok. (It is easy to prove that name substitution preserves well-formed names.) By lemma 21, from P3 we have $\Gamma, [\mathtt{l}/\eta]\Gamma', \overline{\mathtt{x}} : \overline{\mathtt{U} \diamond [\mathtt{l}/\eta]\sigma}, \mathtt{thisType}, \mathtt{this} : \mathtt{thisType} \diamond [\mathtt{l}/\eta]\sigma_0 \vdash$ $[\mathtt{l}/\eta]\mathtt{e} : \mathtt{N} \diamond [\mathtt{l}/\eta]\sigma_1$. Because $\Delta \mapsto ty\sigma_0$, $[\mathtt{l}/\eta]\Delta \mapsto^{ty} [\mathtt{l}/\eta]\sigma_0$. Therefore, by Rule T-Md1, we have $[\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]$(`U m(`$\overline{\mathtt{U}}$ $\overline{\mathtt{x}}$`){ return e; }`) ok in `thisType`. Done □

Lemma 24 (Type Substitution Preserves Range Typing)**.** *If* $\Theta; \Delta; \Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{R}$ ok in $\mathtt{N}$ *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $\Theta; [\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{R}$ ok in $\mathtt{N}$.

Proof. Straightforward induction on range typing derivations. □

Lemma 25 (Name Substitution Preserves Range Typing)**.** *If* $\Theta; \Delta; \Gamma, \eta, \Gamma' \vdash \mathtt{R}$ ok in $\mathtt{N}$ *and* $\Gamma \vdash \mathtt{l}$ ok*, then* $\Theta; [\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{R}$ ok in $\mathtt{N}$.

Proof. Straightforward induction on range typing derivations. □

Lemma 26 (Type Substitution Preserves Trait Application Typing)**.** *If* $\Theta; \Delta; \Gamma, \mathtt{X}, \Gamma' \vdash \mathtt{E}$ ok in $\mathtt{N}$ *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $\Theta; [\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{E}$ ok in $\mathtt{N}$.

Proof. Straightforward induction on trait application typing derivations in Figure 5.8.

Case $\Theta; \Delta; \Gamma, \mathtt{X}, \Gamma' \vdash$ `Tr<`$\overline{\mathtt{R}}$`>` ok in `N`. By the inversion of Rule T-TrP, we have P1:

$\forall \mathtt{l} \in dom(\Delta_{req}).\ \Delta_{req}(\mathtt{l}) = \Delta(\mathtt{l})$;

and P2: $\Theta;\Delta;\Gamma,\mathtt{X},\Gamma' \vdash \overline{\mathtt{R}}$ ok `in` N. By P1, we have that $\forall \mathtt{l} \in dom(\Delta_{req})$. $\mathtt{X} \notin TVars(\Delta(\mathtt{l}))$. Therefore, $\forall \mathtt{l} \in dom(\Delta_{req})$. $\Delta_{req}(\mathtt{l}) = ([\mathtt{T}/\mathtt{X}]\Delta)(\mathtt{l})$. From P2, by lemma 25, we have $\Theta;\Delta;\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{R}}$ ok `in` N. Finally, by Rule T-TrP, we have $\Theta;[\mathtt{T}/\mathtt{X}]\Delta;\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}](\mathtt{Tr<}\overline{\mathtt{R}}\mathtt{>})$ ok in N.

Case $\Theta;\Delta;\Gamma,\mathtt{X},\Gamma' \vdash$ `E satisfies { H }` ok in N. By the inversion of Rule T-Sat, we have P1: $\Theta;\Delta;\Gamma,\mathtt{X},\Gamma' \vdash \mathtt{E}$ ok `in` N; P2: $\forall \mathtt{m} \in (dom(\Delta_H) \cap dom(\Delta_E))$. $\Delta_H(\mathtt{m}) = \Delta_E(\mathtt{m})$; and P3: $\Gamma,\mathtt{X},\Gamma' \vdash \overline{\mathtt{H}}$ ok. From the rule of computing member typing context for trait application (Figure 5.4), we know $\mathtt{X} \notin TVars(\Delta_E)$. From P2, we have $\forall \mathtt{m} \in (dom(\Delta_H) \cap dom(\Delta_E))$. $\mathtt{X} \notin TVars(\Delta_H(\mathtt{m}))$. Therefore, for $\Gamma,\Gamma' \vdash\vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{H}} \hookrightarrow \Delta_H{}'$, we have $\forall \mathtt{m} \in (dom(\Delta_H{}') \cap dom(\Delta_E))$. $\Delta_H{}'(\mathtt{m}) = \Delta_E(\mathtt{m})$. By the introduction hypothesis, from P1, we have $\Theta;[\mathtt{T}/\mathtt{X}]\Delta;\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{E}$ ok in N. From P3, it is straightforward that $\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{H}}$ ok. Finally, by Rule T-Sat, we have $\Theta;[\mathtt{T}/\mathtt{X}]\Delta;\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]$(`E satisfies { H }`) ok in N.

Done. □

Lemma 27 (Name Substitution Preserves Trait Application Typing)**.** *Suppose name substitution does not generate name conflicts. If* $\Theta;\Delta;\Gamma,\eta,\Gamma' \vdash \mathtt{E}$ ok in N *and* $\Gamma \vdash \mathtt{l}$ ok*, then* $\Theta;[\mathtt{l}/\eta]\Delta;\Gamma,[\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]\mathtt{E}$ ok in N.

Proof. Straightforward induction on trait application typing derivations in Figure 5.8. The proof is similar to lemma 26. □

Lemma 28 (Type Substitution Preserves Pattern Typing)**.**

If $\Delta;\Gamma,\mathtt{X},\Gamma' \vdash \mathtt{D}$ ok in `thisType` *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $[\mathtt{T}/\mathtt{X}]\Delta;\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{D}$ ok in `thisType`.

If $\Theta;\Delta;\Gamma,\mathtt{X},\Gamma' \vdash \mathtt{PT}$ ok in `thisType` *and* $\Gamma \vdash \mathtt{T}$ ok*, then* $\Theta;[\mathtt{T}/\mathtt{X}]\Delta;\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{PT}$ ok in `thisType`.

Proof. If $\Delta;\Gamma,\mathtt{X},\Gamma' \vdash$ `{ M; GP; PT; use E }` ok in `thisType`, then by the inversion of Rule T-Bdy in Figure 5.9, we have P1: $\Delta;\Gamma,\mathtt{X},\Gamma' \vdash \overline{\mathtt{M}}$ ok in `thisType`; P2: $\Delta;\Gamma,\mathtt{X},\Gamma' \vdash \overline{\mathtt{GP}}$ ok in `thisType`; P3: $\Theta;\Delta;\Gamma,\mathtt{X},\Gamma' \vdash \overline{\mathtt{PT}}$ ok in `thisType`; and P4: $\Theta;\Delta;\Gamma,\mathtt{X},\Gamma' \vdash \overline{\mathtt{E}}$ ok in `thisType`. By lemma 22, from P1, we have $[\mathtt{T}/\mathtt{X}]\Delta;\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{M}}$ ok `thisType`. From P2, we have $[\mathtt{T}/\mathtt{X}]\Delta;\Gamma,[\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{GP}}$ ok in `thisType` (It is easy to prove that type substitution preserves group typing). By the

induction hypothesis, from P3 we have $\Theta; [\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{PT}}$ ok in `thisType`. By lemma 26, from P4 we have $\Theta; [\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash \overline{[\mathtt{T}/\mathtt{X}]\mathtt{E}}$ ok in `thisType`. Therefore, by Rule T-Bdy, we have $[\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash$ `[T/X]({ `$\overline{\mathtt{M}}$`; `$\overline{\mathtt{GP}}$`; `$\overline{\mathtt{PT}}$`; use `$\overline{\mathtt{E}}$ `})` ok in `thisType`.

If $\Theta; \Delta; \Gamma, \mathtt{X}, \Gamma' \vdash$ `pattern Pt<`$\overline{\mathtt{Y}}$`;`$\overline{\eta}$`> P in R D` ok `in N`, then by the inversion of Rule T-Ptn, we have P1: $\Gamma, \mathtt{X}, \Gamma' \vdash$ `pattern Pt<`$\overline{\mathtt{X}}$`;`$\overline{\eta}$`> P in R D` $\hookrightarrow \Delta_0$; P2: $\Theta; \Delta; \Gamma, \mathtt{X}, \Gamma' \vdash$ `R` ok `in thisType`; P3: $\Gamma, \mathtt{X}, \Gamma', \overline{\mathtt{Y}}, \overline{\eta} \vdash$ `P`; and P4: $(\Delta \oplus \Delta_0); \Gamma, \mathtt{X}, \Gamma', \overline{\mathtt{Y}}, \overline{\eta} \vdash$ `D` ok `in thisType`. From P1, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash$ $[\mathtt{T}/\mathtt{X}]$(`pattern Pt<`$\overline{\mathtt{X}}$`;`$\overline{\eta}$`> P in R D`) $\hookrightarrow [\mathtt{T}/\mathtt{X}]\Delta_0$ (according the definition of computing member typing context for patterns in Figure 5.4). By lemma 24, From P2 we have $\Theta; [\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash$ $[\mathtt{T}/\mathtt{X}]$`R` ok `in thisType`. From P3, we have $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma', \overline{\mathtt{Y}}, \overline{\eta} \vdash [\mathtt{T}/\mathtt{X}]$`P`. By the induction hypothesis, from P4 we have $[\mathtt{T}/\mathtt{X}](\Delta \oplus \Delta_0); \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma', \overline{\mathtt{Y}}, \overline{\eta} \vdash [\mathtt{T}/\mathtt{X}]$`D` ok `in thisType`. Therefore, by Rule T-Ptn we have $\Theta; [\mathtt{T}/\mathtt{X}]\Delta; \Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]$(`pattern Pt<`$\overline{\mathtt{Y}}$`;`$\overline{\eta}$`> P in R D`) ok `in N`.

Done. □

Lemma 29 (Name Substitution Preserves Pattern Typing)**.** *Suppose name substitution does not generate name conflicts.*

If $\Delta; \Gamma, \eta, \Gamma' \vdash$ `D` ok in `thisType` *and* $\Gamma \vdash$ `l` ok*, then* $[\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]$`D` ok in `thisType`*.*

If $\Theta; \Delta; \Gamma, \eta, \Gamma' \vdash$ `PT` ok in `thisType` *and* $\Gamma \vdash$ `l` ok*, then* $\Theta; [\mathtt{l}/\eta]\Delta; \Gamma, [\mathtt{l}/\eta]\Gamma' \vdash [\mathtt{l}/\eta]$`PT` ok in `thisType`*.*

Proof. The proof is similar to the proof of lemma 20. □

Definition 5.

If $\mathtt{f} = \mathtt{T} \in \sigma$ *and* $\mathtt{f} \Downarrow \mathtt{f}'$*, then* $\mathtt{f}' = \mathtt{T} \in \sigma$*.*

If $\mathtt{m} = \overline{\mathtt{T}} \to \mathtt{T} \in \sigma$ *and* $\mathtt{m} \Downarrow \mathtt{m}'$*, then* $\mathtt{m}' = \overline{\mathtt{T}} \to \mathtt{T} \in \sigma$*.*

Lemma 30 (Expression Evaluation Preserves Expression Typing)**.** *If* $\Gamma \vdash \mathtt{e} : \tau$ *and* $\mathtt{e} \Downarrow \mathtt{e}^0$*, then* $\Gamma \vdash \mathtt{e}^0 : \tau$*.*

Proof. The evaluation of an expression is the evaluation of names inside the expression (i.e. name concatenations are evaluated into name constants). Accord to definition 5, a member typing lookup

has the same result when the name of the member is evaluated. So, it is straightforward that $\Gamma \vdash \mathtt{e}^0 : \tau$. □

Lemma 31 (Method Evaluation Preserves Method Typing)**.** *If* $\Delta; \Gamma \vdash \mathtt{M}$ ok in $\mathtt{N}$ *and* $\mathtt{M} \Downarrow \mathtt{M}^0$*, then* $\Delta; \Gamma \vdash \mathtt{M}^0$ ok in $\mathtt{N}$*.*

Proof. Similar to the evaluation of an expression, the evaluation of a method evaluates the names inside the method (i.e. name concatenations are evaluated into name constants). Accord to definition 5 and lemma 30, it is straightforward that $\Delta; \Gamma \vdash \mathtt{M}^0$ ok in $\mathtt{N}$. □

Lemma 32 (Pattern, Group, and Trait Application Evaluation Preserves Typing)**.**

(1) If $\Delta; \Gamma \vdash \mathtt{D}$ ok in $\mathtt{N}$*, and* $\mathtt{D} \Downarrow \overline{\mathtt{M}^0}$*, then* $\Delta; \Gamma \vdash \overline{\mathtt{M}^o}$ ok in $\mathtt{N}$*.*

(2) If $\Theta; \Delta; \Gamma \vdash \mathtt{E}$ ok in $\mathtt{N}$*, and* $\Sigma \vdash \mathtt{E} \Downarrow \overline{\mathtt{M}^0}$*, then* $\Delta; \Gamma \vdash \overline{\mathtt{M}^o}$ ok in $\mathtt{N}$*.*

(3) If $\Delta; \Gamma \vdash \mathtt{GP}$ ok in $\mathtt{N}$*, and* $\mathtt{GP} \Downarrow \overline{\mathtt{M}^0} \Rightarrow \Sigma$*, then* $\Delta; \Gamma \vdash \overline{\mathtt{M}^o}$ ok in $\mathtt{N}$*.*

(4) If $\Theta; \Delta; \Gamma \vdash \mathtt{PT}$ ok in $\mathtt{N}$*, and* $\Sigma \vdash \mathtt{PT} \Downarrow \overline{\mathtt{M}^0}$*, then* $\Delta; \Gamma \vdash \overline{\mathtt{M}^o}$ ok in $\mathtt{N}$*.*

Proof. For (1), by the inversion of Rule T-Bdy in Figure 5.9, from the proposition:

$\Delta; \Gamma \vdash \{\ \overline{\mathtt{M}};\ \overline{\mathtt{GP}};\ \overline{\mathtt{PT}};\ \mathtt{use}\ \overline{\mathtt{E}}\ \}$ ok in $\mathtt{N}$,

we have $\Delta; \Gamma \vdash \overline{\mathtt{M}}$ ok `in` $\mathtt{N}$; $\Delta; \Gamma \vdash \overline{\mathtt{GP}}$ ok `in` $\mathtt{N}$; $\Theta; \Delta; \Gamma \vdash \overline{\mathtt{PT}}$ ok `in` $\mathtt{N}$; and $\Theta; \Delta; \Gamma \vdash \overline{\mathtt{E}}$ ok `in` $\mathtt{N}$. By the inversion of Rule E-Bdy in Figure 5.11, from $\mathtt{D} \Downarrow \overline{\mathtt{M}^0}$, we have $\overline{\mathtt{M}} \Downarrow \overline{\mathtt{M}^o}_1$; $\overline{\mathtt{GP}} \Downarrow \overline{\mathtt{M}^o}_2 \Rightarrow \Sigma$; and $\Sigma \vdash \overline{\mathtt{E}} \Downarrow \overline{\mathtt{M}^o}_4$. By the induction hypothesis, we have $\Delta; \Gamma \vdash \overline{\mathtt{M}^0}_1$ ok in $\mathtt{N}$; $\Delta; \Gamma \vdash \overline{\mathtt{M}^0}_2$ ok in $\mathtt{N}$; $\Delta; \Gamma \vdash \overline{\mathtt{M}^0}_3$ ok in $\mathtt{N}$; and $\Delta; \Gamma \vdash \overline{\mathtt{M}^0}_4$ ok in $\mathtt{N}$. From Rule E-Bdy, we know there is no name conflicts and the result is the concatenation of the generated methods: $\overline{\mathtt{M}^o}_1, \overline{\mathtt{M}^o}_2, \overline{\mathtt{M}^o}_3, \overline{\mathtt{M}^o}_4$, which is well-typed. For (2), the evaluation of a trait application is the evaluation of trait body. In the trait body, there are methods, groups, and patterns. By lemma 31, the evaluation a method generates a well-formed method. By the induction hypothesis, the evaluation of a pattern or a group generates a list of well-formed methods. Therefore, it is straightforward that the evaluation of a trait application generates a list of well-formed methods.

For (3), the evaluation of a group is the evaluation of the group body. By the induction hypothesis,

we know the evaluation of a body returns a list of well-formed methods.

For (4), Straightforward induction on pattern evaluation derivations in Figure 5.11.

Case $\Sigma \vdash$ `pattern Pt<`$\overline{\texttt{X}}$`;`$\overline{\eta}$`> T`$_p$ `f`$_p$ `in R D` $\Downarrow []$, then the empty list of methods is well-formed.

Case $\Sigma \vdash$ `pattern Pt<`$\overline{\texttt{X}}$`;`$\overline{\eta}$`> T`$_p$ `f`$_p$ `in R D` $\Downarrow \overline{\texttt{M}^o}$. (4.1) By Rule E-Pt-Succ, we have P1: $[\overline{\texttt{c}}/\overline{\eta}][\overline{\texttt{C}}/\overline{\texttt{X}}]\texttt{D} \Downarrow \overline{\texttt{M}^o}_1$ (for some $\overline{\texttt{c}}$ and $\overline{\texttt{C}}$); and P2: $\Sigma \vdash$ `pattern Pt<`$\overline{\texttt{X}}$`;`$\overline{\eta}$`> T`$_p$ `f`$_p$ `in R' D` $\Downarrow \overline{\texttt{M}^o}_2$ for some `R'`. By lemma 28 and lemma 29, from P1 we have $\Delta;\Gamma \vdash \overline{\texttt{M}^o}_1$ ok in `N`. By the induction hypothesis, from P2 we have $\Delta;\Gamma \vdash \overline{\texttt{M}^o}_2$ ok in `N`. Therefore, $\Delta;\Gamma \vdash \overline{\texttt{M}^o}$ ok in `N`. Or (4.2) By Rule E-Pt-Fail, we have $\Sigma \vdash$ `pattern Pt<`$\overline{\texttt{X}}$`;`$\overline{\eta}$`> T`$_p$ `f`$_p$ `in R' D` $\Downarrow \overline{\texttt{M}^o}$. By the induction hypothesis, we also have $\Delta;\Gamma \vdash \overline{\texttt{M}^o}$ ok in `N`.

Done. □

Theorem 14 (Meta Class Generates Well-formed FJ Class)**.** *If* `MC` ok *and* `MC` $\Downarrow$ `L`^{o}*, then* `L`^{o} *is well-formed in the FJ type system.*

Proof. By lemma 32, we know trait applications generate well-formed methods. By lemma 18, we know generated methods are well-formed in the FJ type system. In Rule E-MC, we check method overriding, making sure that any generated method can validly override a method in the base class. Therefore, `L`^{o} is well-formed in the FJ type system. □

A.4 Statement-Level Reflective Metaprogramming

Lemma 33 (Staged Type Preservation of Statements)**.** *For some meta-level non-pattern statement* `s` *(i.e. statement without patterns), if* $\Gamma;\Theta \vdash$ `s : C` *where* $TyVars(\Gamma) = \emptyset$*, then when applying object-level type-checking to* `s`*, we have* $\Gamma' \vdash$ `s : C` *for some object-level typing environment* Γ'*.*

Proof. Because our object language uses the type system of Featherweight Java (FJ) to check the object expressions. We first prove that our type-checking of the meta expressions is equivalent to the type system of FJ.

FJ uses functions *fields* and *mtype* to lookup the type of a member in a class. Instead, we lookup the type of a member in Θ^{CL} and Θ^{TR}. According to the definition of Θ^{CL}, for type lookup

that happens in class C, (1) if $\Theta^{CL}(\mathsf{C})(\mathbb{C}) = \overline{\mathsf{T}} \to \cdot$, then *fields*$(\mathsf{C}) = \overline{\mathsf{T}}$; (2) if $\Theta^{CL}(\mathsf{C})(\mathsf{m}) = \overline{\mathsf{U}} \to \mathsf{U}$, then *mtype*$(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{U}} \to \mathsf{U}$; and (3) if $\Theta^{CL}(\mathsf{C})(\mathsf{f}) = \mathsf{T}$ and *fields*$(\mathsf{C}) = \overline{\mathsf{T}}$, then $\mathsf{T} \in \overline{\mathsf{T}}$.

Suppose class C uses some trait $\mathcal{T}$. Because Θ^{CL} includes the members imported from trait $\mathcal{T}$, so we have $\Theta^{TR}(\mathcal{T}) \subseteq \Theta^{CL}(\mathsf{C})$ (assume there is no name conflict statically, otherwise $\Theta^{CL}(\mathsf{C})$ is not defined). So, for type lookup that happens in trait $\mathcal{T}$, there are two cases.

Case One: when we lookup the type of a member defined in some other class D, then it is similar to type lookup that happens in class D (see the previous paragraph).

Case Two: when we lookup the type of a member defined in trait $\mathcal{T}$ (e.g. access a member via the **this** variable), we have Θ that is equal to $\Theta^{CL} \uplus [\mathbf{thistype} \mapsto \Theta^{TR}(\mathcal{T})]$. Therefore, we have (1) if $\Theta(\mathbf{thistype})(\mathsf{m}) = \overline{\mathsf{U}} \to \mathsf{U}$, then *mtype*$(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{U}} \to \mathsf{U}$; and (2) if $\Theta(\mathbf{thistype})(\mathsf{f}) = \mathsf{T}$ and *fields*$(\mathsf{C}) = \overline{\mathsf{T}}$, then $\mathsf{T} \in \overline{\mathsf{T}}$. (i.e. during type-checking, we check that a field required by trait $\mathcal{T}$ must be provided by class C.)

The expression typing rules are presented in Figure 6.8. Compared to the expression typing rules in FJ, there is only one difference, which is that we use type lookup approach that is different from FJ. Because we have proved that our type lookup approach can be reduced to the type lookup approach in FJ, we then have the property that if $\Gamma; \Theta \vdash \mathsf{e} : \mathsf{T}$ (using our typing rules), then there exists some Γ' in FJ, such that $\Gamma' \vdash \mathsf{e} : \mathsf{T}$ (using FJ typing rules).

Second, we prove that if the statements in the meta language does not include patterns, then the statement typing in the meta language is equivalent to the statement typing in the object language. Because there is no pattern, thus no non-return statement. For the statement typing rules in Figure 6.8, we only need to select the typing rules for statement $\mathsf{T}\ \mathsf{x} = \mathsf{e}; \mathsf{s}$, statement $\mathsf{x} = \mathsf{e}; \mathsf{s}$, and statement $\mathsf{return}\ \mathsf{e}$. Comparing these rules to the rules presented in Figure 6.2, it is obvious for readers to know that they are equivalent.

Done. □

Definition 6 (Class Name Alias $\tilde{\mathtt{N}}$)**.** *An alias of a class name, written* $\tilde{\mathtt{N}}$*, is a distinct class name. A class name can have multiple name alias.*

(1) An alias of a class name preserves the nominal subtyping relations of that class.

(2) For any class alias $\tilde{\mathtt{N}}$, $\tilde{\mathtt{N}} \notin \mathrm{dom}(\Theta^{CL})$.

(3) For some class name alias $\tilde{\mathtt{N}}$ *and some* Θ*, if* $\tilde{\mathtt{N}}$ *does not appear in the domain of* Θ*, then* $\Theta(\tilde{\mathtt{N}}) = \Theta^{CL}(\mathtt{N})$*; otherwise because there exists some* Δ *such that* $\tilde{\mathtt{N}} \mapsto \Delta \in \Theta$, $\Theta(\tilde{\mathtt{N}}) = \Delta$.

Lemma 34 (Type Substitution Preserves Subtyping)**.** *If* $\mathtt{U} <: \mathtt{T}$*, then* $[\mathtt{N}/\mathtt{X}]\mathtt{U} <: [\mathtt{N}/\mathtt{X}]\mathtt{T}$.

Proof. (1) If $\mathtt{U} = \mathtt{X}$, then $\mathtt{T} = \mathtt{X}$. (i.e. the only subtype of a type variable is itself.) So, $[\mathtt{N}/\mathtt{X}]\mathtt{X} <: [\mathtt{N}/\mathtt{X}]\mathtt{X}$. (2) If $\mathtt{U}$ is some class name, then $\mathtt{T}$ must be also some class name, then $[\mathtt{N}/\mathtt{X}]\mathtt{U} = \mathtt{U}$ and $[\mathtt{N}/\mathtt{X}]\mathtt{T} = \mathtt{T}$. Therefore $[\mathtt{N}/\mathtt{X}]\mathtt{U} <: [\mathtt{N}/\mathtt{X}]\mathtt{T}$. Done. □

Definition 7 (Type Substitution over Θ)**.** *Suppose* $\Theta = [\kappa_i \mapsto \Delta_i]^{\,i\in 1\ldots n}$.

(1) If $\mathtt{X} \notin \mathrm{dom}(\Theta)$,

then $[\tilde{\mathtt{N}}/\mathtt{X}]\Theta = [\tilde{\mathtt{N}} \mapsto \Theta(\mathtt{N})] \uplus [\kappa_i \mapsto [\tilde{\mathtt{N}}/\mathtt{X}]\Delta_i]^{\,i\in 1\ldots n}$

(2) If $\mathtt{X} \in \mathrm{dom}(\Theta)$,

then $[\tilde{\mathtt{N}}/\mathtt{X}]\Theta = [\tilde{\mathtt{N}}/\mathtt{X}](\Theta\backslash\mathtt{X}) \uplus [\tilde{\mathtt{N}} \mapsto [\tilde{\mathtt{N}}/\mathtt{X}](\Theta(\mathtt{X}))]$

Definition 8 (Name Substitution over Θ)**.** *Suppose* $\Theta = [\kappa_i \mapsto \Delta_i]^{\,i\in 1\ldots n}$.

$[\mathtt{l}/\eta]\Theta = [\kappa_i \mapsto [\mathtt{l}/\eta]\Delta_i]^{\,i\in 1\ldots n}$ *if for each* Δ_i*, for any* $\ell_1, \ell_2 \in \mathrm{dom}(\Delta_i)$, $[\mathtt{l}/\eta]\ell_1 \neq [\mathtt{l}/\eta]\ell_2$*. Otherwise, generates a name conflict error.*

Lemma 35 (Name Substitution over Θ Does Not Generate Name Conflicts)**.** *Suppose* $\Theta = [\kappa_i \mapsto \Delta_i]^{\,i\in 1\ldots n}$.

There always exists some Θ' *such that* $[\mathtt{l}/\eta]\Theta = \Theta'$.

Proof. Accord to the definition of Θ, for each Δ_i, for any $\ell_1, \ell_2 \in \mathrm{dom}(\Delta_i)$, $\ell_1 \neq \ell_2$. Proposition $\ell_1 \neq \ell_2$ means there does not exist any substitution $[\overline{\mathtt{l}_0}/\overline{\eta_0}]$ such that $[\overline{\mathtt{l}_0}/\overline{\eta_0}]\ell_1 = [\overline{\mathtt{l}_0}/\overline{\eta_0}]\ell_2$. Therefore, for some substitution $[\mathtt{l}/\eta]$, $[\mathtt{l}/\eta]\Theta$ does not generate any name conflict. □

Lemma 36 (Type Substitution Preserves Well-formed Types)**.** *If* $\Gamma, \mathtt{X}^o, \Gamma' \vdash \mathtt{U} : \mathrm{ok}$, $\Gamma \vdash \mathtt{T}$ ok*, and* $\Gamma \vdash \mathbb{N}(\mathtt{T})$ in $\mathtt{N} \Rightarrow o$*, then* $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{U}$ ok.

Proof. Prove by induction on the rules for checking well-formed types. □

Lemma 37 (Expression-Level Type Substitution Preserves Typing)**.** *Suppose* **this** : $\mathtt{N}_0 \in \Gamma, \Gamma'$*;* $\tilde{\mathtt{N}}$ *is an alias of* $\mathtt{N}$*; and* $\tilde{\mathtt{N}} \notin \mathrm{dom}(\Theta)$*. If* $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{e} : \mathtt{U}$*,* $\Gamma \vdash \mathtt{N}$ ok*, and* $\Gamma \vdash \mathbb{N}(\mathtt{N})$ in $\mathtt{N}_0 \Rightarrow o$*, then* $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{e} : [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U}$*.*

Proof. Prove by induction on the expression typing rules.

(1) Case $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{x} : \mathtt{U}$. By the inversion of the typing rule for variables and by the induction hypothesis, we have $\mathtt{x} : \mathtt{U} \in \Gamma, \mathtt{X}^o, \Gamma'$. Obviously, we have $\mathtt{x} : [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U} \in \Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'$ (please note that if $\mathtt{x} : \mathtt{U} \in \Gamma$, then $[\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U} = \mathtt{U}$). By the typing rule for variables, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash \mathtt{x} : [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U}$.

(2) Case $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{e.l} : \Delta(\mathtt{l})$. By the inversion of the typing rule for member access, we have $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{e} : \mathtt{U}$ and $\Delta = \Theta(\mathtt{U})$. **(i)** If $\mathtt{U} = \mathtt{X}$, then by the induction hypothesis, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{e} : \tilde{\mathtt{N}}$. Because $\mathtt{X} \in \mathrm{dom}(\Theta)$, by the definition of type substitution over Θ, we have $[\tilde{\mathtt{N}}/\mathtt{X}](\Theta(\mathtt{X})) = ([\tilde{\mathtt{N}}/\mathtt{X}]\Theta)(\tilde{\mathtt{N}})$; thus having $[\tilde{\mathtt{N}}/\mathtt{X}](\Theta(\mathtt{X})(\mathtt{l})) = ([\tilde{\mathtt{N}}/\mathtt{X}]\Theta)(\tilde{\mathtt{N}})(\mathtt{l})$. Finally, by the typing rule for member access, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{e.l} : [\tilde{\mathtt{N}}/\mathtt{X}](\Theta(\mathtt{X})(\mathtt{l}))$. **(ii)** If $\mathtt{U} \neq \mathtt{X}$, then by the induction hypothesis, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{e} : \mathtt{U}$. By the definition of type substitution over Θ, we have $[\tilde{\mathtt{N}}/\mathtt{X}](\Theta(\mathtt{U})) = ([\tilde{\mathtt{N}}/\mathtt{X}]\Theta)(\mathtt{U})$; thus having $[\tilde{\mathtt{N}}/\mathtt{X}](\Theta(\mathtt{U})(\mathtt{l})) = ([\tilde{\mathtt{N}}/\mathtt{X}]\Theta)(\mathtt{U})(\mathtt{l})$. Finally, by the typing rule for member access, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{e.l} : [\tilde{\mathtt{N}}/\mathtt{X}](\Theta(\mathtt{U})(\mathtt{l}))$.

(3) Similar to Case (2).

(4) Case $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{new\ U(\overline{e})} : \mathtt{U}$. By the inversion of the typing rule for object creation and by the induction hypothesis, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash \overline{[\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{e}} : \overline{[\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U}}$; $\Delta = \Theta(\mathtt{U})$; $\Delta(\mathbb{C}) = \overline{\mathtt{T}} \to \cdot$; $\Gamma \vdash \mathtt{U}$ ok; and $\overline{\mathtt{U}} <: \overline{\mathtt{T}}$. **(i)** If $\mathtt{U} = \mathtt{X}$, then $([\tilde{\mathtt{N}}/\mathtt{X}]\Theta)(\tilde{\mathtt{N}})(\mathbb{C}) = \overline{[\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{T}} \to \cdot$. **(ii)** If $\mathtt{U} \neq \mathtt{X}$, then $([\tilde{\mathtt{N}}/\mathtt{X}]\Theta)(\mathtt{U})(\mathbb{C}) = \overline{[\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{T}} \to \cdot$. From both of the cases, we have $([\tilde{\mathtt{N}}/\mathtt{X}]\Theta)([\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U})(\mathbb{C}) = \overline{[\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{T}} \to \cdot$. Because $\overline{\mathtt{U}} <: \overline{\mathtt{T}}$, we have $\overline{[\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U}} <: \overline{[\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{T}}$. By lemma 36, from $\Gamma \vdash \mathtt{U}$ ok, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma' \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U}$ ok. Therefore, by the typing rule for object creation, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}](\mathtt{new\ U(\overline{e})}) : [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{U}$.

(5) The rules for type casting: $\Gamma; \Theta \vdash \mathtt{(U)e} : \mathtt{U}$ are easy to prove. Similar to case (4). □

Lemma 38 (Expression-Level Name Substitution Preserves Typing)**.** *If* $\Gamma, \eta, \Gamma'; \Theta \vdash \mathtt{e} : \mathtt{U}$*, and* $\Gamma \vdash \mathtt{l}$ ok*, then* $\Gamma, \Gamma'; [\mathtt{l}/\eta]\Theta \vdash [\mathtt{l}/\eta]\mathtt{e} : \mathtt{U}$*.*

Proof. Proof sketch: when type-checking an expression, we lookup member types in Θ. Before name substitution, for some κ and ℓ, suppose $\Theta(\kappa)(\ell) = \phi$, then according to the definition of name substitution over Θ, we still have $\Theta(\kappa)([\mathtt{l}/\eta]\ell) = \phi$. So, it is easy to conclude that $\Gamma, \Gamma'; [\mathtt{l}/\eta]\Theta \vdash [\mathtt{l}/\eta]\mathtt{e} : \mathtt{U}$. □

Lemma 39 (Type Substitution Preserves Well-formed Ranges)**.** *If* $\Gamma, \mathtt{X}^o, \Gamma' \vdash \mathtt{R} : \text{ok}$*,* $\Gamma \vdash \mathtt{T}$ ok*, and* $\Gamma \vdash \mathbb{N}(\mathtt{T})$ in $\mathtt{N} \Rightarrow o$*, then* $\Gamma, [\mathtt{T}/\mathtt{X}]\Gamma' \vdash [\mathtt{T}/\mathtt{X}]\mathtt{R}$ ok*.*

Proof. Prove by induction on the rules for checking well-formed ranges. □

Lemma 40 (Statement-Level Type Substitution Preserves Typing)**.** *Suppose* $\mathbf{this} : \mathtt{N}_0 \in \Gamma, \Gamma'$*;* $\tilde{\mathtt{N}}$ *is an alias of* $\mathtt{N}$*; and* $\tilde{\mathtt{N}} \notin \text{dom}(\Theta)$*. If* $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{ns}$ ok*,* $\Gamma \vdash \mathtt{N}$ ok*, and* $\Gamma \vdash \mathbb{N}(\mathtt{N})$ in $\mathtt{N}_0 \Rightarrow o$*, then* $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{ns}$ ok*.*

Proof. First, we prove that type substitution preserves well-formed pattern statements. We prove by induction on the typing rules for pattern statements in Figure 6.8. For some pattern statement ps, that is pattern<$\overline{\mathtt{X}}$,$\overline{\eta}$> P in R { ns; }, suppose $\Gamma, \mathtt{X}^o, \Gamma'; \Theta \vdash \mathtt{ps}$ ok.

(1) Case when $\mathbb{N}(\mathtt{R}) = \mathtt{C}$. By the inversion of the first rule for typing pattern statements, we have $\Gamma, \mathtt{X}^o, \Gamma' \vdash \mathtt{R}$ ok; $(\Gamma, \mathtt{X}^o, \Gamma', \overline{\mathtt{X}^-}, \overline{\eta}) \vdash \mathtt{P}$ ok; $\Theta' = \Theta \uplus [\mathtt{C} \mapsto \delta(\mathtt{P})]$; and $(\Gamma, \mathtt{X}^o, \Gamma', \overline{\mathtt{X}^-}, \overline{\eta}); \Theta' \vdash \mathtt{ns}$ ok. By lemma 39, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma' \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{R}$ ok. By lemma 36, we have $(\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma', \overline{\mathtt{X}^-}, \overline{\eta}) \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{P}$ ok. By the induction hypothesis, we have $(\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma', \overline{\mathtt{X}^-}, \overline{\eta}); [\tilde{\mathtt{N}}/\mathtt{X}]\Theta' \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{ns}$ ok. From $\Theta' = \Theta \uplus [\mathtt{C} \mapsto \delta(\mathtt{P})]$, we have $[\tilde{\mathtt{N}}/\mathtt{X}]\Theta' = ([\tilde{\mathtt{N}}/\mathtt{X}]\Theta) \uplus [\mathtt{C} \mapsto \delta([\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{P})]$. Therefore, by the first rule for typing patten statements, we have $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{ps}$ ok.

(2) Case when $\mathbb{N}(\mathtt{R}) = \mathtt{X}_0$ and ${\mathtt{X}_0}^+ \in \Gamma$. **(i)** If $\Gamma \vdash \mathbb{N}(\mathtt{N})$ in $\mathtt{N}_0 \Rightarrow -$ or $\mathtt{X} \neq \mathtt{X}_0$, then the proof is similar to case (1). **(ii)** If $\Gamma \vdash \mathbb{N}(\mathtt{N})$ in $\mathtt{N}_0 \Rightarrow +$ and $\mathtt{X} = \mathtt{X}_0$, then

(2.1) Case when $\mathtt{N} = \mathtt{C}$ for some class C. By the inversion of the second rule for typing pattern statements, we have $\Gamma, \mathtt{X}^o, \Gamma' \vdash \mathtt{R}$ ok; $(\Gamma, \mathtt{X}^o, \Gamma', \overline{\mathtt{X}^-}, \overline{\eta}) \vdash \mathtt{P}$ ok; $\Theta' = \Theta \uplus [\mathtt{X}_0 \mapsto \delta(\mathtt{P})]$; and

$(\Gamma, \mathtt{X}^o, \Gamma', \overline{\mathtt{X}^-}, \overline{\eta}); \Theta' \vdash \mathtt{ns}$ ok. By lemma 39, we have $\Gamma, [\tilde{\mathtt{C}}/\mathtt{X}]\Gamma' \vdash [\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{R}$ ok. By lemma 36, we have $(\Gamma, [\tilde{\mathtt{C}}/\mathtt{X}]\Gamma', \overline{\mathtt{X}^-}, \overline{\eta}) \vdash [\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{P}$ ok. By the induction hypothesis, we have $(\Gamma, [\tilde{\mathtt{C}}/\mathtt{X}]\Gamma', \overline{\mathtt{X}^-}, \overline{\eta}); [\tilde{\mathtt{C}}/\mathtt{X}]\Theta' \vdash [\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{ns}$ ok. We know $\mathbb{N}([\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{R}) = \tilde{\mathtt{C}}$. From $\Theta' = \Theta \uplus [\mathtt{C} \mapsto \delta(\mathtt{P})]$, we have $[\tilde{\mathtt{C}}/\mathtt{X}]\Theta' = ([\tilde{\mathtt{C}}/\mathtt{X}]\Theta) \uplus [\mathtt{C} \mapsto \delta([\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{P})]$. By the first rule for typing statements, we have $\Gamma, [\tilde{\mathtt{C}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{C}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{ps}$ ok.

(2.2) Case when $\mathtt{N} = \mathtt{thisType}$. The proof is similar to case (2.1) except that in the last step, we need to use the third rule for typing statement (because $\mathbb{N}([\widetilde{\mathtt{thisType}}/\mathtt{X}]\mathtt{R}) = \widetilde{\mathtt{thisType}}$) to conclude that $\Gamma, [\widetilde{\mathtt{thisType}}/\mathtt{X}]\Gamma'; [\widetilde{\mathtt{thisType}}/\mathtt{X}]\Theta \vdash [\widetilde{\mathtt{thisType}}/\mathtt{X}]\mathtt{ps}$ ok.

(3) Case when $\mathbb{N}(\mathtt{R}) = \mathtt{X}_0 \wedge \mathtt{X}_0{}^- \in \Gamma$ or $\mathbb{N}(\mathtt{R}) = \mathtt{thisType}$. For case when $\mathbb{N}(\mathtt{R}) = \mathtt{thisType}$, the proof is similar to case (1). For case $\mathbb{N}(\mathtt{R}) = \mathtt{X}_0 \wedge \mathtt{X}_0{}^- \in \Gamma$:

(3.1) Case when $\mathtt{X}_0 \neq \mathtt{X}$, the proof of this case is similar to case (1).

(3.2) Case when $\mathtt{X}_0 = \mathtt{X}$. Because $\mathtt{X}_0{}^- \in \Gamma$, N cannot be thisType. Suppose $\mathtt{N} = \mathtt{C}$ for some class C. By the inversion of the third rule for typing pattern statements, we have $\Gamma, \mathtt{X}^o, \Gamma' \vdash \mathtt{R}$ ok; $(\Gamma, \mathtt{X}^o, \Gamma', \overline{\mathtt{X}^-}, \overline{\eta}) \vdash \mathtt{P}$ ok; $\Theta' = \Theta \uplus [\mathtt{C} \mapsto \delta(\mathtt{P})]$; and $(\Gamma, \mathtt{X}^o, \Gamma', \overline{\mathtt{X}^-}, \overline{\eta}); \Theta' \vdash \mathtt{ns}$ ok. By lemma 39, we have $\Gamma, [\tilde{\mathtt{C}}/\mathtt{X}]\Gamma' \vdash [\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{R}$ ok. By lemma 36, we have $(\Gamma, [\tilde{\mathtt{C}}/\mathtt{X}]\Gamma', \overline{\mathtt{X}^-}, \overline{\eta}) \vdash [\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{P}$ ok. By the induction hypothesis, we have $(\Gamma, [\tilde{\mathtt{C}}/\mathtt{X}]\Gamma', \overline{\mathtt{X}^-}, \overline{\eta}); [\tilde{\mathtt{C}}/\mathtt{X}]\Theta' \vdash [\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{ns}$ ok. From $\Theta' = \Theta \uplus [\mathtt{C} \mapsto \delta(\mathtt{P})]$, we have $[\tilde{\mathtt{C}}/\mathtt{X}]\Theta' = ([\tilde{\mathtt{C}}/\mathtt{X}]\Theta) \uplus [\mathtt{C} \mapsto \delta([\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{P})]$. By the first rule for typing statements, we have $\Gamma, [\tilde{\mathtt{C}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{C}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{C}}/\mathtt{X}]\mathtt{ps}$ ok.

Done.

After proving $\Gamma, [\tilde{\mathtt{N}}/\mathtt{X}]\Gamma'; [\tilde{\mathtt{N}}/\mathtt{X}]\Theta \vdash [\tilde{\mathtt{N}}/\mathtt{X}]\mathtt{ps}$ ok, to prove the rest of the non-return statements is straightforward: by induction on the typing rules for non-return statements □

Lemma 41 (Statement-Level Type Substitution Preserves Typing)**.** *If* $\Gamma, \eta, \Gamma'; \Theta \vdash \mathtt{ns}$ ok*, and* $\Gamma \vdash \mathtt{l}$ ok*, then* $\Gamma, \Gamma'; [\mathtt{l}/\eta]\Theta \vdash [\mathtt{l}/\eta]\mathtt{ns}$ ok.

Proof. Straightforward induction on the typing rules for non-return statements. □

Lemma 42 (Statement Concatenation Preserves Typing)**.** *Suppose function lvar collects all the local variables in a sequence of statements. If* $\Gamma; \Theta \vdash \mathtt{ns}_1$ ok*,* $\Gamma; \Theta \vdash \mathtt{ns}_2$ ok*, and lvars*$(\mathtt{ns}_1) \cap$

lvars($\texttt{ns}_2$) = $\emptyset$, *then* $\Gamma;\Theta \vdash \texttt{ns}_1\texttt{;ns}_2$ ok.

Proof. Suppose $\texttt{ns}_1 = \texttt{ss};\texttt{ns}_0$. By the typing rules for statements, from $\Gamma;\Theta \vdash \texttt{ns}_1$ ok, we have $\Gamma;\Theta \vdash \texttt{ss}$ ok and $\Gamma';\Theta \vdash \texttt{ns}_0$ ok where $\Gamma \subseteq \Gamma'$. From $\Gamma;\Theta \vdash \texttt{ns}_2$ ok, we have $\Gamma';\Theta \vdash \texttt{ns}_2$ ok (i.e. environment weakening). By the induction hypothesis, we have $\Gamma';\Theta \vdash \texttt{ns}_0\texttt{;ns}_2$ ok. Therefore, by the induction on the typing rules for statements, we can prove that $\Gamma;\Theta \vdash \texttt{ss};\texttt{ns}_0\texttt{;ns}_2$ ok. □

Theorem 15 (Type Preservation for Patterns)**.** *If* $\Gamma;\Theta \vdash \texttt{ps}$ ok *and* $\texttt{ps} \longrightarrow \texttt{ns}$*, then* $\Gamma;\Theta \vdash \texttt{ns}$ ok

Proof. Prove by induction on $\texttt{ps} \longrightarrow \texttt{ns}$ (see Figure 6.10). First suppose:

$\texttt{ps} = \texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> P in C} \diamond \texttt{\{ ... \} \{ ns; \}}$.

For the cases that the reduction of ps returns an empty statement sequence (we assume an empty statement sequence is also acceptable by our syntax) or fails to match a member in R, it is straightforward that the result is also well-typed.

For the case that the reduction of ps matches a member in R, the return is

$$[\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{ns; ps}'$$

Compared to ps, the only change of $\texttt{ps}'$ is in its range, where a member is removed from the structural part of the range. Because the structural part of a range has no effect on type-checking, so obviously $\Gamma;\Theta \vdash \texttt{ps}'$ ok.

From the premise that $\Gamma;\Theta \vdash \texttt{ps}$ ok, by the inversion of the first typing rule for pattern statements in Figure 6.8, we have $\Gamma \vdash \texttt{R ok}$; $\Theta' = \Theta \uplus [\texttt{C} \mapsto \delta(\texttt{P})]$; and $(\Gamma, \overline{\texttt{X}^-}, \overline{\eta});\Theta' \vdash \texttt{ns}$ ok. Because $\Gamma \vdash \texttt{R ok}$, we have $\Gamma \vdash \overline{\texttt{l}}$ ok and $\Gamma \vdash \overline{\texttt{T}}$ ok (i.e. $\overline{\texttt{l}}$ and $\overline{\texttt{T}}$ are obtained from R). By lemma 40 and lemma 41, we have $\Gamma;[\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\Theta' \vdash [\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{ns}$ ok, where $[\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\Theta' = [\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\Theta \uplus [\texttt{C} \mapsto \delta([\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{P})]$. Because variables $\overline{\eta}$ and $\overline{\texttt{X}}$ are bound to the pattern and do not appear in the outer environment of pattern, we have $[\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\Theta = \Theta$. Because pattern P matches a member in C, thus $[\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{P} \in \texttt{C}$. Therefore, $[\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\Theta' = \Theta$. So, $\Gamma;\Theta \vdash [\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{ns}$ ok. Because we rename the local variables in ns to avoid the duplicate of local variable names, thus by lemma 42, we finally have $\Gamma;\Theta \vdash [\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{ns; ps}'$ ok. Done. □

Theorem 16 (Progress for Patterns)**.** *If* $\emptyset; \Theta \vdash$ `ps` ok *and* `ps` *is in the form of* `pattern<`$\overline{\mathtt{X}}$`,`$\overline{\eta}$`>` `P in R {` $\mathtt{ns}_0$ `}`*, then there is some* `ns` *with* `ps` $\longrightarrow$ `ns` *(suppose* `ns` *includes empty statements).*

Proof. Because $\emptyset; \Theta \vdash$ `ps` ok , by the inversion of the typing rules for statement patterns, we have $\emptyset \vdash$ `R` ok. Because `R` is closed, then it could be `C`, `C`$|\{\overline{\mathtt{l}}\}$, `C`$\backslash\{\overline{\mathtt{l}}\}$, and `C` $\diamond$ $\{\overline{\mathtt{F}};\mathtt{Q};\overline{\mathtt{H}}\}$. Because `R` is well-typed, we know that `C` is well-defined. Although this paper does not present the evaluation rules for ranges, a well-formed range can be always reduced to some value `C` $\diamond$ $\{\overline{\mathtt{F}};\mathtt{Q};\overline{\mathtt{H}}\}$. So, if `R` is not a value, then the evaluation of `ps` is the reduction of `R`. Otherwise, we can always reduce the pattern statement using the rules that are partially listed in Figure 6.10.

Done. □