Dipartimento di Informatica e
Scienze dell'Informazione

MetaFJig

A Meta-Circular Composition Language for Java-like Classes

by

Marco Servetto

Theses Series **DISI-TH-2011-03**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy `http://www.disi.unige.it/`

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

MetaFJig

A Meta-Circular Composition Language for Java-like Classes

by

Marco Servetto

February, 2011

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science (S.S.D. INF/01)

Submitted by Marco Servetto
DISI, Univ. di Genova
servetto@disi.unige.it

Date of submission: February 2011 Release date: May 2011

Title: **MetaFJig**
A Meta-Circular Composition Language for Java-like Classes

Advisors:
Elena Zucca
DISI, Univ. di Genova
zucca@disi.unige.it

Giovanni Lagorio
DISI, Univ. di Genova
lagorio@disi.unige.it

Ext. Reviewers:
Shigeru Chiba
Tokyo Institute of Technology
chiba@is.titech.ac.jp

Ferruccio Damiani
Università di Torino
damiani@di.unito.it

Atsushi Igarashi
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Dedicated to the memory of Claudio De Prà.

Acknowledgements

These years of Ph.D. studies have had a very important influence on my development, I now recognize in the world a depth that I ignored before.

I am very thankful to **Antonio** and **Daniele** for keeping my passion for programming always alive, eventually prompting me to take the Ph.D. course.

My deepest thanks to **Elena**, for teaching me what science really is. You have allowed me to see a world uncorrupted by any economical aspects, not tempted by sterile careerism: a world where love for truth and mathematical beauty overcome individual egoism to fulfil the life with meaning. Moreover, you have enabled me to communicate effectively by teaching me how to explain what I see.

My deepest thanks to **Giovanni**, you were my last programming guru. I have never known anyone able to understand complex interactions of so many elements and to bind them within a single tiny thought. Your strength and your guidance have allowed me to start again in my growing process, so that I could overcome my limit yet another time.

My sincere thanks to **Alessio** and **Salvatore** for being the best friends one can desire.

Abstract

We present a new language design coupling disciplined meta-programming features with a composition language, in the context of Java-like classes. That is, programmers can write meta-expressions that combine class definitions, on top of a small set of primitive composition operators.

This approach implies compile-time execution, as in template meta-programming, but exploits some different key features, which are beneficial in many respects. The meta-language coincides with the conventional language, hence programmers are not required to learn new syntax and idioms. Soundness is guaranteed by a lightweight technique, called *checked compile-time execution*, where conventional typing errors are detected by an *incremental typechecking* approach, and a meta-expression can be reduced only if it can be successfully typechecked with the current type information. Class composition errors, instead, are detected dynamically, allowing programmers to appropriately handle these compilation errors. The approach is modular, that is, can be defined and implemented on top of typechecking and execution of the conventional language. Last but not least, our technique ensures an additional important property, called *meta-level soundness*, stating that typing errors never originate from already compiled (meta-)code, that is, programmers can safely use libraries.

The module composition language is inspired by the seminal Bracha's Jigsaw framework. However, an important novelty is that composition operators are *deep*, that is, they allow to manipulate (e.g., rename or duplicate) fields, methods and nested classes at any depth level. The result has a great expressive power, allowing, e.g., to solve the expression problem, encode the main AOP mechanisms, bring some refactoring techniques at the language level, cooperate with external data sources and develop active libraries, while keeping a very simple semantics and type system that are a natural extension for, say, a Java programmer.

The outcome of this thesis is mainly to present new ideas in the design of programming languages and to show their effectiveness. Moreover, this language design work is supported by a rigorous formalization of both the language and the checked compile-time execution process, including proofs of important properties, and by a prototype that effectively shows the modularity of the approach.

Contents

List of Figures

Chapter 1

Introduction

Support for code reuse is a key feature which should be offered by programming languages, in order to automate and standardize a process that programmers should, otherwise, do by hand: duplicating code for adapting it to best solve a particular instance of some generic problem. Two different strategies, which can be adopted to achieve code reuse, are *(module) composition languages* and *meta-programming*.

In the former approach programmers can write fragments of code (*modules*) which are not self-contained, but depend on other fragments. Such dependencies can be later resolved by combining modules via composition operators, obtaining different customized behaviors. These operators form a *module composition language*.

Our work focuses on the case where modules are *classes*, in the sense of object-oriented class-based languages with nominal types, which we briefly call *Java-like languages*. In such languages, the original composition mechanism is *inheritance*. That is, they only support the asymmetric *extends* operator, allowing to obtain, roughly, the same effect programmers would get by copying the code of the parent class into the heir.

Inheritance has been extremely successful and has caused a profound change in the development of software systems. However, the need of going beyond has been recognized for a long time, leading to a variety of proposals for improving the flexibility and expressive power of object-oriented programming. We mention, among others, mixin classes [FKF98, ALZ03], virtual classes [Ern01, EOC06], generic classes as in Java 5, mixin modules [FF98, ALZ06] and other proposals for adding a module/component level [MFH01, ACN02], traits [SDNB03, FR04, LS08b], multimethods [BC97, CMLC06], aspect-oriented programming [KLM+97].

All these proposals share a common limitation: programmers are provided with a fixed set of composition mechanisms, and they cannot define their own operators, as it happens, e.g., with function/method definitions.

A natural way to overcome this limitation is to allow programmers to write (meta-)code that can be used to generate customized code for solving particular instances of a generic problem. In the context of Java-like languages, the most widely used meta-programming facility is *template meta-programming*, as, e.g., in C++ [Str00], where templates, that is, parametric functions or classes, can be defined and later instantiated to obtain highly-optimized specialized versions. The instantiation mechanism requires the compiler to generate a temporary (specialized) source code, which is compiled along with the rest of the program.

The use of templates can be thought of as *compile-time execution* [CE00]. This technique is very powerful, yet can be very difficult to understand, since its syntax and idioms are esoteric compared to conventional programming. For the same reasons, maintaining and evolving code which exploits template meta-programming is rather complex. Moreover, well-formedness of generated source code can only be checked "a posteriori", making the whole process hard to debug. Despite all these limitations, template meta-programming is widely used, proving that there is a strong need for its expressive power.

Proposed approach The key idea exploited in this thesis is to distil the best of the two approaches, that is, to couple disciplined meta-programming features with a composition language. That is, we are going to design a Java-like language allowing programmers to define their own customized operators for combining classes.

In such a language, a class declaration associates an expression of primitive type **`class`** with a class name. The simplest form of such an expression is a *basic class*, which is similar to a Java class body. For instance,

```
{ int answer() { return 42; } }
```

is a basic class declaring a single method named `answer`. We can give the name `A` to that class by writing:

```
A = {  int answer() { return 42; }  }
```

Compound expressions of type **`class`** can be constructed using *primitive composition operators*, for instance *sum*:

```
A = // as before
B = A [+] { int one() { return 1; } }
```

This program is equivalent to the following:

```
A = // as before
B = {
  int answer() { return 42; } }
  int one() { return 1; }
}
```

Moreover, since **class** is a primitive type of the language, a class can be the result of a method.

For instance, the following program

```
C = {
  class m(){
    return { int one(){return 1;} };
  }
}
D = { int answer() {return 42;} } [+] new C().m()
```

declares two classes, `C` and `D`. The former, `C`, declares a single method named `m`, which returns a value of type **class**. This value, in turn, is a basic class[1] declaring the method `one`. The latter, `D`, is defined as the sum of a basic class and a method invocation, which has to be evaluated in order to obtain the corresponding basic class. This program could be equivalently written as:

```
C = //as before
D = { int answer() {return 42;} } [+] { int one(){return 1;} }
```

Hence, compilation of a class table requires to perform (meta-)reduction steps, by a process that we call *compile-time execution*, as in template meta-programming.[2] However, our approach exploits some different key features, which are beneficial in many respects:

- The meta-language coincides with the conventional language programmers are used to, hence they are not required to learn new syntax and idioms. This approach is called *meta-circular*[3].
- Soundness is guaranteed by a lightweight technique, called *checked compile-time-execution*, where conventional typing errors are detected by an *incremental typechecking* approach, and a meta-expression can be reduced only if it can be successfully typechecked with the current type information. Class composition errors, instead, are detected dynamically, allowing programmers to appropriately handle these compilation errors.
- The approach is *modular*, that is, can be defined and implemented on top of typechecking and execution of the conventional language.
- Last but not least, our technique ensures an additional important property, called *meta-level soundness*, stating that typing errors never originate from already compiled (meta-)code, that is, programmers can safely use libraries.

[1] For helping readability, throughout the thesis we emphasize in grey basic classes occurring in method bodies as meta-expressions.

[2] We emphasize that this is a real execution, and not some sort of abstract interpretation.

[3] Meta-circular languages are also called *homogeneous*, see, e.g., [She00].

The composition language is inspired by the seminal Bracha's Jigsaw framework [Bra92]. However, an important novelty is that composition operators are *deep*, that is, they allow to manipulate (e.g., rename or duplicate) fields, methods and nested classes at any depth level. The result has a great expressive power, allowing, e.g., to solve the expression problem, encode the main AOP mechanisms, bring some refactoring techniques at the language level, cooperate with external data sources and develop active libraries, while keeping a very simple semantics and type system that are a natural extension for, say, a Java programmer.

The outcome of this thesis is mainly to present new ideas in the design of programming languages and to show their effectiveness. Moreover, this language design work is supported by:

- A rigorous formalization of both the language and the checked compile-time execution process, presented in the form of calculi in the spirit of Featherweight Java [IPW99, IPW01] (FJ for short), including proofs of important properties.
- A prototype that effectively shows the modularity of the approach.

The rest of this chapter is devoted to an overview of the thesis.

1.1 Summary

In this thesis we present MetaFeatherweight Jigsaw (METAFJIG for short), a meta-circular composition language for Java-like classes. In order to better explain our approach, we introduce the language in an incremental way, focusing in each chapter on a subset of features. The name Featherweight Jigsaw (FJIG for short) is used for the language subset(s) with no meta-level features.

Basic classes (Chapter 2)

In this chapter we present FJIG_0, a minimal core of FJIG, that is, a language with no composition mechanism. In FJIG_0, classes are just basic classes, with abstract or defined fields and methods, and no inheritance. Expressions in method bodies are field access, method invocation, and object creation. The (unique) constructor of a class initializes all non abstract fields by arbitrary expressions, that is, FJIG_0 constructors have no canonical form as in FJ. Typing follows the Java nominal approach, that is, class names are types. However, the subtyping relation is *not* implicitly determined by sub-classing, as in Java, but must be explicitly chosen by the programmer, by declaring a set of *supertypes*, introduced by the keyword `implements`. In other words, FJIG_0 supports multiple supertypes, similarly to Java implemented interfaces.

Composition operators (Chapter 3)

We introduce FJIG$_1$, which extends FJIG$_0$ by an expressive set of composition operators allowing to manipulate classes:

- The *sum* operator combines two classes with no conflicting definitions for the same member, sharing abstract members with the same name.
- The *restrict* operator removes a definition from a class, making the corresponding member abstract.
- The *alias* operator duplicates the declaration of an existing member for another member.
- The *redirect* operator replaces all references with receiver `this` to a member name by a different name, and removes its declaration.

The design of the composition operators of FJIG$_1$ comes out naturally, yet not trivially, by taking FJ [IPW99, IPW01] as starting point and replacing inheritance by the more general composition operators of Jigsaw [Bra92].

The work presented in the first two chapters is based on [LSZ09c, LSZ09b, LSZ10a]. However, these previous versions of FJIG also included *frozen* and *private* members, and, correspondingly, operators such as *freeze* and *hide* were expressible, as in the original Jigsaw framework [Bra92]. In this thesis, since the main aim is the design of a meta-circular composition language with deep operators, we do not consider these features, which have a non trivial interaction with nesting, see Section 8.2).

Moreover, in [LSZ09c, LSZ09b, LSZ10a] we have defined two different execution models for FJIG, that is, *flattening* and *direct* semantics, and proved their equivalence. Here, we focus on flattening semantics, which reduces an FJIG$_1$ program to an FJIG$_0$ program by performing composition operators.

Meta-language for composition (Chapter 4)

In this chapter, we extend FJIG$_1$ by a *meta-level.* That is, we define a language METAFJIG$_1$ where class definitions are first-class values which can be combined by using the four primitive operators. This language achieves a great level of code reuse, allowing one to define classes whose shape depends on external input.

In METAFJIG$_1$, a *meta-program* is a sequence of class declarations which are associations between class names and arbitrary expressions. A meta-program is reduced to a conventional program, where right-hand sides of class declarations are basic classes, by a process which we

call *compile-time execution*. In order to guarantee that compile-time execution, if terminating, produces a well-typed program, we use an incremental typechecking approach, where a meta-expression can be reduced only if it can be successfully typechecked with the current type information. We call the resulting process *checked compile-time execution.*

Moreover, w.r.t. other approaches, METAFJIG$_1$ enjoys an important property, called *meta-level soundness*, which guarantees that already compiled code (libraries) will not cause type errors, differently from what happens, e.g., for C++ templates. This property holds thanks to the fact that during compile-time execution we cannot generate arbitrary code, but only compose basic classes which were explicitly written in the library.

Preliminary works exploring the METAFJIG$_1$ concepts are [LSZ09a, LSZ10b], while [SZ10] introduces a language and a notion of checked compile-time execution very close to those presented in this chapter. The main difference is that in [SZ10] we use annotations to explicitly keep track of the precomputed type information, providing a guideline for an optimized implementation. Here we prefer to stick to a more abstract version which is more easy to generalize to the case with nesting as we do in Chapter 6.

Integrating composition and nesting (Chapter 5)

The primitive composition operators of FJIG$_1$ are very expressive, but a single FJIG$_1$ class is not an adequate unit of reuse. In this chapter, we extend FJIG$_1$ in order to smoothly integrate composition operators with *nesting*, allowing hierarchical composition. In the resulting language, FJIG$_\star$, a class can embody a whole hierarchy of classes, and composition operators allow one to manipulate a nested class at any depth level.

The generalization of the composition operators to the case with nesting is very natural and intuitive[4], and, more generally, the language keeps a very simple semantics and type system which represent a natural extension for, say, a Java programmer.

Here, nominal typing means that types are paths of the form $\texttt{outer}^n.c_1.\ \ldots\ .c_k$ which, depending on the class (node) where they occur, denote another node in the nesting tree. However, paths denoting the same class are *not* necessarily equivalent, since they can behave differently w.r.t. composition operators.

FJIG$_\star$ operators provide an expressive power usually addressed by family polymorphism or virtual classes, allowing one to express a wide range of common patterns for code composition/adaptation, e.g., to solve the expression problem, to encode the main AOP mechanisms, and to bring some refactoring techniques at the language level.

This chapter can be read independently of Chapter 4. The work presented in it is an improvement

[4]From the point of view of the programmer: as we will see, the formalization is sometimes tricky.

of [CSZ10, CSZ11].

We first describe nested classes and deep composition operators, and provide some more interesting examples showing the expressive power of the language. Then we provide the formal syntax, semantics, and type system and prove the soundness results.

Meta-language for composition and nesting (Chapter 6)

Finally, in MetaFJig$_\star$, the expressive power of the MetaFJig$_1$ meta-level, together with the FJig$_\star$ capability of representing a whole component as a single class, allow one to encapsulate a library within a single meta-expression, still still offering all the guarantees of MetaFJig$_1$, including meta-level soundness.

The possibility offered by the meta-level to write classes whose structure depends on an external source, like a database table, is generalized to a whole hierarchy, as one can extract from a whole database or XML schema. Moreover, in MetaFJig$_\star$ it is possible to encode some concepts that usually require ad-hoc language extensions, like enumerations.

In this chapter, we provide an extended collection of examples showing the expressive power of MetaFJig$_\star$.

Moreover, we illustrate the design of the prototype compiler, notably explaining how it is built on top of Java compiler and Java Virtual Machine. We are currently working on the details of its implementation, which should be available in a couple of weeks at `http://www.disi.unige.it/person/ServettoM/MetaFJig`. At the time of submitting this thesis, this url shows a prototype which follows exactly the same schema on the simpler language of [SZ10], a slight variant of MetaFJig$_1$.

The integration of the composition operators of FJig$_\star$ with the meta-level is a novel contribution of this thesis.

Chapter 2

Basic classes

> *The trick, according to Chiang, was for Jonathan to stop seeing himself as trapped inside a limited body with a forty-two inch wingspan... the trick was to know that his true nature lived, as perfect as an unwritten number, everywhere at once across space and time.*
>
> (Richard Bach - Jonathan Livingston Seagull)

In this chapter, we describe an FJIG fragment, called FJIG_0, where classes are just *basic classes*. FJIG_0 is a simple Java-like calculus, in the spirit of FJ, with abstract or defined fields and methods, no inheritance, and where expressions in method bodies are field access, method invocation, and object creation. Typing follows the Java nominal approach, that is, class names are types. However, the subtyping relation is *not* implicitly determined by sub-classing, as in Java, but must be explicitly declared by the programmer.

We first informally introduce the language by some examples in Section 2.1, then give the formal syntax and reduction semantics in Section 2.2, the type system in Section 2.3, and finally prove the soundness of the type system in Section 2.4.

2.1 Examples

In FJIG_0, a program is a sequence of class declarations. A very simple example is the following:

```
Foo = {
  Foo m(){return this.m();}
}
```

consisting only of the declaration of class `Foo`, with a method `m` that loops, that is, calls `m` on the same object, represented by the special variable **`this`**. Note that the syntax of a class

declaration is slightly different from Java. Indeed, the equal (=) symbol is used to stress that a class declaration binds a *class name* to a *basic class*, which is similar to a Java class body. Other differences, discussed below, concern constructors, abstract members, and subtyping.

Constructors In FJIG there is no overloading, hence a class defines exactly one constructor. However, differently from FJ, where this unique constructor has a canonical form, there is no a priori relation between the parameter list and the constructor body, which is a sequence of *field expressions* associating initialization expressions to field names. Such expressions cannot refer to **this**.[1]

In this example:

```
Bar = {
  Foo f1;
  Foo f2;
  constructor(Foo x){
    this.f1 = new Foo();
    this.f2 = x;
  }
  Foo m(){return this.f1;}
}
```

the constructor takes one argument and initializes the fields `f1` and `f2`. The first initialization expression is an object creation expression. Note that, differently from Java, constructor declaration is introduced by the keyword **constructor** instead of the name of the class. Indeed, the constructor belongs to an (anonymous) basic class, which could be used as definition of different class names.

Abstract members Classes *declare* members, that can be fields or methods. In FJIG members are treated uniformly, hence both fields and methods can be either declared **abstract** or be *defined*, whereas in Java this modifier can only be applied to methods. As in Java, a class having at least one **abstract** member must be declared **abstract**, and an **abstract** class cannot be instantiated.[2] As shown in the following example, an **abstract** field has no corresponding field expression in the constructor, and we assume a default constructor with no parameters for the classes with no defined fields.

```
A = abstract{
```

[1]Usually Java-like languages do not forbid **this** inside constructors, allowing to access uninitialized fields. Forbidding to use **this** is a coarse-grained restriction, to ensure that fields are accessed only after initialization. It is possible to develop more fine-grained techniques for the same purpose [GS09], but this is outside the scope of our work.

[2]However, note that being abstract is a property of a basic class, rather than of a class declaration.

```
  abstract Foo f;
  //constructor(){} //implicitly defined
  abstract Foo m1();
  Foo m2(){return this.f;}
}
```

Note that, as expected, the body of method `m` can access the abstract field `f`.

Some proposals in traits literature are similar to abstract field declarations, for instance *required fields* in [BDG08].

Subtyping FJIG keeps the Java nominal approach, that is, class names are types. However, the subtyping relation is *not* implicitly determined by subclassing, as in Java, but must be explicitly declared by the programmer using the **`implements`** keyword. For instance, in:

```
FooBar = implements Foo, Bar{
  Foo f1;
  Foo f2;
  constructor(Foo x){
    this.f1 = x;
    this.f2 = new Foo();
  }
  Foo m(){return new FooBar(new Foo());}
}
```

class `FooBar` is declared to be a subtype of classes `Foo` and `Bar`. The type system checks, for each declared supertype, that the subtyping relation can be safely assumed, that is, members of the supertype are members of the basic class as well.[3] This check is analogous to that on implemented interfaces in Java. In the example, the subtyping annotation is safe, since `FooBar` has all the members of `Foo` (method `m`) and all the members of `Bar` (method `m` and fields `f1` and `f2`). Since `FooBar` is a subtype of `Foo`, method `m` can return a `FooBar`.

Abstract classes can be implemented as well:

```
B = implements A{
  Foo f;
  constructor(){this.f = new Foo();}
  Foo m1(){return this.f;}
  Foo m2(){return this.m1();}
}
```

[3]Formally, that the *class type* of the basic class is a subtype of the class type of the implemented class, see rule (WF-CLASS-TYPE) in Figure 2.4.

Class `B` is declared to be a subtype of `A`, hence should declare at least all the members of `A`, that is, field `f` and methods `m1` and `m2`. On the other hand, class `B` is *not* a subclass of `A`, that is, it does *not* reuse the code of `A` in any way. In other words, **`implements`** is a relation among types, and members are *not* inherited. Hence, `B` *must* declare a method `m2` even though `m2` is declared in `A`. For the same reason, differently from Java, where they are implicitly inherited, abstract members of the supertype must be declared as well, so that class types can be computed in isolation, as illustrated by the following example:

```
C = abstract implements A{
  Foo f;
  constructor(){this.f = new Foo();}
  abstract Foo m1();
  Foo m2(){return this.m1();}
}
```

Removing method `m1` from `C` would make the example ill-typed.

Finally note that, being a relation among types, nothing prevents **`implements`** to be cyclic.

We assume as default supertype the predefined class `Object` with no members.

2.2 Syntax and reduction rules

Syntax In Figure 2.1 we give the syntax of FJIG$_0$ programs. We use the bar notation for sequences, e.g., $\overline{d}$ is a sequence of declarations d, and ϵ denotes the empty string. We assume infinite sets of *class names* c, *(member) names* n, and variables (parameter names) x. The syntax is designed to keep a Java-like flavour as much as possible.

A program is a sequence of class declarations, where a class name is bound to a class expression. In this first version of the language, the only kind of class expressions are basic classes.

A basic class consists of an optional **`abstract`** modifier, a sequence of supertypes, a constructor, and a sequence of (member) declarations, which can be either abstract or non abstract (member definitions).

Members can be fields or methods, and member declarations are in the style of Java, but fields can be declared abstract.

Sequences of class declarations, supertypes, declarations, and field expressions are considered as sets, that is, order and repetitions are immaterial.

In a well-formed program, a class name cannot be declared twice, that is, a program is a map from class names to class expressions. Analogously, no member can be declared twice in a basic class, that is, a sequence of declarations $\overline{d}$ is a map from member names to declarations. This

p	::=	$\overline{cd}$	program
cd	::=	$c = ce$	class declaration
ce	::=	b	class expression
b	::=	$ch\ \{\ k\ \overline{d}\ \}$	basic class
ch	::=	μ **implements** $\overline{c}$	class header
μ	::=	$\epsilon \mid$ **abstract**	abstract modifier
k	::=	$kh\{\overline{fe}\}$	constructor
kh	::=	**constructor** $(\overline{T\ x})$	constructor header
fe	::=	**this.**$f = e$**;**	field expression
d	::=	$fd \mid md$	declaration
f, m	::=	n	field name, method name
fd	::=	$\mu\ T\ f$**;**	field declaration
md	::=	$mh\{$ **return** e**;** $\} \mid$ **abstract** mh**;**	method declaration
mh	::=	$T\ m(\overline{T\ x})$	method header
e	::=		expression
		$x \mid e.f \mid e.m\ (\overline{e}) \mid$ **new** $c\ (\overline{e})$	conventional
		$\mid c\ (\overline{fe})$	pre-object
T	::=	$c \mid \ldots$	type
v	::=	$c\ (\overline{fv})$	value
fv	::=	**this.**$f = v$**;**	field value

Figure 2.1: FJIG$_0$ syntax

implies that, differently from Java, there is no method overloading, and there is no overloading between field and method names. However, for better readability, we will use the metavariable f when a name is used for a field, m for a method.

For sequences that are actually maps, for instance $\overline{d}$, we can use the standard notations $\mathsf{dom}(\overline{d})$, $\overline{d}(n)$, and $\overline{d}\backslash n$, for the domain, the application to an element, and the map obtained by removing an element from the domain, respectively.

Note that in all the thesis, in all sequence that are maps the order is immaterial.

A parameter name cannot be declared twice in a constructor or method header. Finally, a field name cannot appear twice in a sequence of field expressions, hence a sequence $\overline{fe}$ is a map from field names to field expressions, and there is exactly a field expression in the constructor for each non abstract field (and none for abstract fields). Expressions include conventional constructs and pre-objects. Conventional constructs are similar to those of FJ. As in FJ, variables include the special variable **`this`**, with the usual meaning. We omit casts for simplicity, since they are not relevant for our technical treatment.

Pre-objects are runtime expressions which cannot be written in programmer's code, that are obtained by reducing a constructor invocation. Indeed, since the constructor has no canonical form, we need two different syntactic forms, differently from FJ. For the sake of simplicity we do not introduce two different syntactic categories for source expressions and runtime expressions, which are a superset of the former.

Types are class names or (non specified) primitive types, like **`int`**. Values are objects, that is, pre-objects where all field expressions are (recursively) values.

Reduction rules An FJIG$_0$ application consists of a *main expression* e which is executed in the program environment p. Formally, the reduction arrow has form $e_1 \underset{p}{\rightarrow} e_2$.

Reduction rules are straightforward, and are formally defined, together with one-hole evaluation contexts $\mathcal{E}^{\mathrm{r}}$, in Figure 2.2.

To be as conventional as possible, the reduction strategy is call-by-value, and evaluation contexts $\mathcal{E}^{\mathrm{r}}$ impose an order from left to right. We use the standard notations $\mathcal{E}^{\mathrm{r}}[\![e]\!]$ for filling a context's hole, $e[\overline{v}/\overline{x}]$ for simultaneous substitution for variables. Moreover, using the wildcard _ when the name of a meta-variable is not relevant:

- if $p(c) = \mu$ **`implements _ {`** $k\ \overline{d}$ **`}`**, then
  - $\mathsf{mBody}(p, c, m)$ returns parameter names and body of method m of class c in p, formally:
    $\mathsf{mBody}(p, c, m) = \langle x_1 \ldots x_n, e\rangle$ if $\overline{d}(m) = T\ m($**`(`**$T_1\ x_1, \ldots, T_n\ x_n$**`)`**$\{$**`return`** e**`;`**$\}$

$$\mathcal{E}^{\mathrm{r}} \quad ::= \quad \Box \mid \mathcal{E}^{\mathrm{r}}.f \mid \mathcal{E}^{\mathrm{r}}.m(\overline{e}) \mid v.m(\overline{v}, \mathcal{E}^{\mathrm{r}}, \overline{e}) \mid \mathbf{new}\, c\,(\overline{v}, \mathcal{E}^{\mathrm{r}}, \overline{e}) \mid c\,(\overline{v}, \mathcal{E}^{\mathrm{r}}, \overline{e})$$

$$e_1 \underset{p}{\rightarrow} e_2$$

$$\text{(CTX)}\ \frac{e_1 \underset{p}{\rightarrow} e_2}{\mathcal{E}^{\mathrm{r}}[\![e_1]\!] \underset{p}{\rightarrow} \mathcal{E}^{\mathrm{r}}[\![e_2]\!]} \qquad \text{(FIELD-ACCESS)}\ \frac{}{c\,(\overline{fv})\,.f \underset{p}{\rightarrow} v}\ \overline{fv}(f) = v$$

$$\text{(INVK)}\ \frac{}{c\,(\overline{fv})\,.m(\overline{v}) \underset{p}{\rightarrow} e[\overline{v}/\overline{x}][c\,(\overline{fv})\,/\mathbf{this}]}\ \mathsf{mBody}(p, c, m) = \langle \overline{x}, e \rangle$$

$$\text{(OBJ-CREATION)}\ \frac{}{\mathbf{new}\, c\,(\overline{v}) \underset{p}{\rightarrow} c\,(\overline{fe}[\overline{v}/\overline{x}])}\ \begin{array}{l}\mathsf{kBody}(p, c) = \langle \overline{x}, \overline{fe} \rangle \\ \mathsf{nonAbs}(p, c)\end{array}$$

Figure 2.2: FJIG$_0$ reduction rules

- $\mathsf{kBody}(p, c)$ returns parameter names and body of constructor of class c in p, formally: $\mathsf{kBody}(p, c) = \langle x_1 \ldots x_n, \overline{fe} \rangle$ if $k = \mathbf{constructor}\,(\,T_1\,x_1, \ldots, T_n\,x_n\,)\,\{\,\overline{fe}\,\}$
- $\mathsf{nonAbs}(p, c)$ holds if class c in p is non abstract, formally if $\mu \neq \mathbf{abstract}$.

Note that only non abstract classes can be instantiated.

2.3 Type system

Types and type environments are defined in Figure 2.3. A *class type environment* is either a program type or a pair consisting of the type of the enclosing class and a program type. The first form is used to type the main expression. A program type is a map from class names to class types. A class type is a tuple consisting of the kind (abstract or non abstract), the supertypes, the constructor type, and a map from member names to their kinds (abstract or non abstract) and types. The special type Λ is used in the type system as type of **this**, but cannot be written in programmer's code[4]. Hence, **this** can be used as method/constructor argument or method result only if the corresponding argument/result type is declared to be a supertype.

Typing rules for programs and classes are given in Figure 2.4, for expressions in Figure 2.5, and subtyping rules in Figure 2.6.

[4]In this first version of the language: in Chapter 5, Λ will become a particular case of *class path*, to be used analogously to a class name. This also motivates the chosen notation.

Δ	$::=$	$pt \mid ct; pt$	class type environment
pt	$::=$	$\overline{c{:}ct}$	program type
ct	$::=$	$[\mu \mid \overline{c} \mid kt \mid \overline{dt}]$	class type
kt	$::=$	$\overline{T}$	constructor type
dt	$::=$	$n{:}\mu\ mt$	declaration type
mt	$::=$	$T \mid \overline{T} \to T$	member type
c^Λ	$::=$	$c \mid \Lambda$	extended class name
T^Λ	$::=$	$T \mid \Lambda$	extended type
Γ	$::=$	$\overline{x{:}T^\Lambda}$	parameter type environment

Figure 2.3: FJIG$_0$ types and type environments

We use the following notations:

- $\mathsf{cType}(\Delta, c^\Lambda)$ is the class type corresponding to c^Λ in Δ, formally: $\mathsf{cType}((ct; \_), \Lambda) = ct$ and $\mathsf{cType}((\_; \_\ c{:}ct), c) = \mathsf{cType}(\_\ c{:}ct, c) = ct$,
- $\mathsf{exists}(\Delta, c^\Lambda)$ holds if c^Λ denotes an existing class in Δ, formally: $\mathsf{exists}(\Delta, c^\Lambda)$ iff $\mathsf{cType}(\Delta, c^\Lambda)$ is defined,
- if $\mathsf{cType}(\Delta, c^\Lambda) = [\mu \mid \overline{c} \mid kt \mid \overline{dt}]$, then
  - $\mathsf{defFields}(\Delta, c^\Lambda)$ are the non abstract fields of class c^Λ in Δ, formally: $\mathsf{defFields}(\Delta, c^\Lambda) = \overline{f{:}\epsilon\ T}$ if $\overline{dt} = \overline{f{:}\epsilon\ T}\ \overline{f{:}\mathbf{abstract}\ T}\ \overline{m{:}\mu\ T \to \overline{T}}$,
  - $\mathsf{mType}(\Delta, c^\Lambda, n)$ is the type of member n of class c^Λ in Δ, formally: $\mathsf{mType}(\Delta, c^\Lambda, n) = mt$ if $\overline{dt}(n) = n{:}\mu\ mt$,
  - $\mathsf{nonAbs}(\Delta, c^\Lambda)$ is analogous to $\mathsf{nonAbs}(p, c)$, but works over class types, formally: $\mathsf{nonAbs}(\Delta, c^\Lambda)$ iff $\mu \neq \mathbf{abstract}$,
  - $\mathsf{kType}(\Delta, c^\Lambda)$ is the type of constructor of class c^Λ in Δ, formally: $\mathsf{kType}(\Delta, c^\Lambda) = kt$,
  - $\mathsf{impl}(\Delta, c^\Lambda)$ are the declared supertypes of class c^Λ in Δ, formally: $\mathsf{impl}(\Delta, c^\Lambda) = \overline{c}$.

Rules (PROGRAM-T) is trivial.

Rule (BASIC-T) typechecks a basic class w.r.t. a program type pt. It checks that the constructor and the members are well-typed, and the resulting class type ct is well-formed, w.r.t. the class type environment composed by ct itself and pt.

$\vdash p : pt,$

$$\text{(PROGRAM-T)}\ \frac{pt \vdash ce_i : ct_i \quad \forall i \in 1..n}{\vdash p : pt} \quad \begin{array}{l} p = c_1 = ce_1 \ldots c_n = ce_n \\ pt = c_1{:}ct_1 \ldots c_n{:}ct_n \end{array}$$

$pt \vdash b : ct$

$$\text{(BASIC-T)}\ \frac{\Delta \vdash k : kt \quad \Delta \vdash \overline{d} : \overline{dt} \quad \Delta \vdash ct}{pt \vdash \mu\ \textbf{implements}\ \overline{c}\ \{\, k\ \overline{d}\,\} : ct} \quad \begin{array}{l} ct = [\mu \mid \overline{c} \mid kt \mid \overline{dt}] \\ \Delta = ct; pt \end{array}$$

$\Delta \vdash k : kt$

$$\text{(CONS-T)}\ \frac{\begin{array}{c}\Delta; x_1{:}T'_1, \ldots, x_n{:}T'_n \vdash e_i : T''_i \quad \forall i \in 1..k \\ \Delta \vdash T''_i \leq T_i \quad \forall i \in 1..k\end{array}}{\Delta \vdash kh\,\{\overline{fe}\} : T'_1 \ldots T'_n} \quad \begin{array}{l} kh = \textbf{constructor}\,(T'_1\ x_1, \ldots, T'_n\ x_n) \\ \overline{fe} = \textbf{this}.f_1 = e_1\textbf{;} \ldots \textbf{this}.f_k = e_k\textbf{;} \\ \mathsf{exists}(\Delta, T'_i) \\ \mathsf{defFields}(\Delta, \Lambda) = f_1{:}\epsilon\ T_1, \ldots, f_k{:}\epsilon\ T_k \quad \forall i \in 1..n \end{array}$$

$\Delta \vdash d : dt$

$$\text{(FIELD-T)}\ \frac{}{\Delta \vdash (\mu\ T\ f\textbf{;}) : (f{:}\mu\ T)}\ \mathsf{exists}(\Delta, T)$$

$$\text{(ABS-METHOD-T)}\ \frac{}{\Delta \vdash (\textbf{abstract}\ mh\textbf{;}) \ :\ (m{:}\textbf{abstract}\ T_1 \ldots T_n \rightarrow T_0)}\ \begin{array}{l} mh = T_0\, m(T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n \end{array}$$

$$\text{(METHOD-T)}\ \frac{\Delta; \textbf{this}{:}\Lambda, x_1{:}T_1, \ldots, x_n{:}T_n \vdash e : T^\Lambda \quad \Delta \vdash T^\Lambda \leq T_0}{\Delta \vdash mh\,\{\textbf{return}\ e\textbf{;}\} \ :\ (m{:}\epsilon\ T_1 \ldots T_n \rightarrow T_0)} \quad \begin{array}{l} mh = T_0\, m(T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n \end{array}$$

$\Delta \vdash ct$

$$\text{(WF-CLASS-TYPE)}\ \frac{}{\Delta \vdash ct} \quad \begin{array}{l} ct = [\mu \mid \overline{c} \mid \_ \mid \overline{dt}] \\ ct \leq \mathsf{cType}(\Delta, c) \quad \forall c \in \overline{c} \\ \mu = \epsilon \text{ implies } \overline{dt} = \overline{n{:}\epsilon\ mt} \end{array}$$

Figure 2.4: FJIG$_0$ typing rules for programs and classes

$\Delta; \Gamma \vdash e : T^\Lambda$

$$\text{(VAR-T)}\ \frac{}{\_; \Gamma \vdash x : T^\Lambda}\ \Gamma(x) = T^\Lambda \qquad \text{(FIELD-ACCESS-T)}\ \frac{\Delta; \Gamma \vdash e : c^\Lambda}{\Delta; \Gamma \vdash e.f : T}\ \mathsf{mType}(\Delta, c^\Lambda, f) = T$$

$$\text{(INVK-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash e : c^\Lambda \\ \Delta; \Gamma \vdash \overline{e} : \overline{T^\Lambda} \\ \Delta \vdash \overline{T^\Lambda} \leq \overline{T}\end{array}}{\Delta; \Gamma \vdash e.m\,(\overline{e})\ : T}\ \mathsf{mType}(\Delta, c^\Lambda, m) = \overline{T} \to T$$

$$\text{(NEW-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash \overline{e} : \overline{T^\Lambda} \\ \Delta \vdash \overline{T^\Lambda} \leq \overline{T}\end{array}}{\Delta; \Gamma \vdash \mathbf{new}\ c\,(\overline{e})\ : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \mathsf{kType}(\Delta, c) = \overline{T}\end{array}$$

$$\text{(OBJ-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash e_i : T_i^\Lambda \quad \forall i \in 1..n \\ \Delta \vdash T_i^\Lambda \leq T_i \quad \forall i \in 1..n\end{array}}{\Delta; \Gamma \vdash c\,(\overline{fe})\ : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fe} = \mathbf{this}.f_1 = e_1\mathbf{;}\ \ldots\ \mathbf{this}.f_n = e_n\mathbf{;} \\ \mathsf{defFields}(\Delta, c) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}$$

Figure 2.5: FJIG$_0$ typing rules for expressions

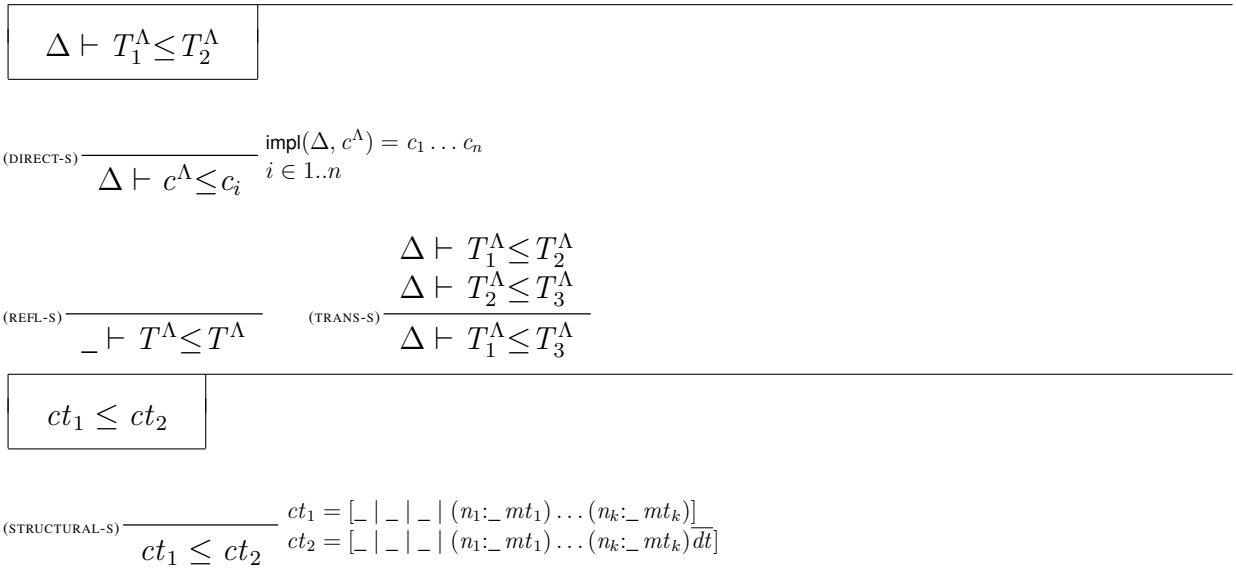

Figure 2.6: FJIG$_0$ subtyping rules

Rule (CONS-T) checks that all the initialization expressions are well-typed with a subtype of the corresponding field type, and that an initialization is provided for all fields.

Rules (FIELD-T), (ABS-METHOD-T) and (METHOD-T) are trivial.

Note that, while the parameter environment Γ used for the method body in (METHOD-T) contains **this**:Λ, the one used to typecheck initialization expressions in (CONS-T) does not contain **this**. However, the class type environment *does include* the type of the enclosing class. Indeed, even though not necessary in FJIG$_0$, this choice is more coherent with the language extension in Chapter 5, where Λ becomes a user-level type. Moreover, the type of the enclosing class would be necessary in an extension with, say, static members.

Rule (WF-CLASS-TYPE) checks that the declared supertypes can be safely assumed (see subtyping rule (STRUCTURAL-S)). This check is analogous to that on implemented interfaces in Java. It checks also that the class type be non abstract only if there are no abstract member types.

Rules (VAR-T), (FIELD-ACCESS-T), (INVK-T) and (NEW-T) are straightforward. Note that there is no need of a special rule for **this**, since it is typechecked with rule (VAR-T). Rule (OBJ-T) checks that each initialization expression has a subtype of the type of the corresponding field name, that the class is non abstract and that an initialization is provided for all (defined) fields.

Rules for subtyping are straightforward.

2.4 Results

This simple language enjoys the standard soundness property, proved as usual by progress and subject reduction theorems. Here relation $\xrightarrow[p]{}{}^{\star}$ denotes reflexive transitive closure of $\xrightarrow[p]{}$.

Theorem 1 (FJIG$_0$ soundness)**.** *If* $\vdash p : pt$*,* $pt; \emptyset \vdash e : T$ *and* $e_1 \xrightarrow[p]{}{}^{\star} e_2$ *then either* e_2 *is a value or* $e_2 \xrightarrow[p]{} \_$.

Theorem 2 (FJIG$_0$ progress)**.** *If* $\vdash p : pt$ *and* $pt; \emptyset \vdash e : T$ *then either* e *is a value or* $e \xrightarrow[p]{} \_$.

Proof. By induction on the typing rules.

Case $\dfrac{}{\_; \Gamma \vdash x : T^{\Lambda}}\,{\scriptstyle(\text{VAR-T})}\ \Gamma(x) = T^{\Lambda}$

This case is empty since $\Gamma = \emptyset$.

Case $\dfrac{\Delta; \Gamma \vdash e : c^{\Lambda}}{\Delta; \Gamma \vdash e.f : T}\,{\scriptstyle(\text{FIELD-ACCESS-T})}\ \mathsf{mType}(\Delta, c^{\Lambda}, f) = T$

From the premise by the inductive hypothesis we have two cases:

- $e \xrightarrow[p]{} \_$
  We can apply (CTX) to show that the whole term reduces.
- e is a well-typed value $c\,(\overline{fv})$, hence we have applied rule
  $$\dfrac{\begin{array}{c}\Delta; \Gamma \vdash v_i : T_i^{\Lambda} \quad \forall i \in 1..n \\ \Delta \vdash T_i^{\Lambda} \leq T_i \quad \forall i \in 1..n \end{array} \quad \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fv} = \mathtt{this}.f_1 = v_1; \ldots \mathtt{this}.f_n = v_n; \\ \mathsf{defFields}(\Delta, c) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}}{\Delta; \Gamma \vdash c\,(\overline{fv})\ : c}\,{\scriptstyle(\text{OBJ-T})}$$
  with $c = c^{\Lambda}$. We can apply rule $\dfrac{}{c\,(\overline{fv})\,.f \xrightarrow[p]{} v}\,{\scriptstyle(\text{FIELD-ACCESS})}\ \overline{fv}(f) = v$

  since $f \in \mathsf{dom}(fv)$. Indeed, from $\mathsf{mType}(\Delta, c^{\Lambda}, f) = T$ and $\mathsf{nonAbs}(\Delta, c)$ we get $f{:}T \in \mathsf{defFields}(\Delta, c)$, and by definition of $\mathsf{defFields}$ we get $\mathsf{dom}(\mathsf{defFields}(\Delta, c)) = \mathsf{dom}(\overline{fv})$.

Case $\dfrac{\begin{array}{c}\Delta; \Gamma \vdash e : c^{\Lambda} \\ \Delta; \Gamma \vdash \overline{e} : \overline{T^{\Lambda}} \\ \Delta \vdash \overline{T^{\Lambda}} \leq \overline{T}\end{array}}{\Delta; \Gamma \vdash e.m\,(\overline{e})\ : T}\,{\scriptstyle(\text{INVK-T})}\ \mathsf{mType}(\Delta, c^{\Lambda}, m) = \overline{T} \to T$

Assume $\overline{e} = e_1 \ldots e_n$. From the premise by the inductive hypothesis we have two cases:

- $e \xrightarrow[p]{} \_$, or $e_i \xrightarrow[p]{} \_$ for some $i \in 1..n$. We can apply (CTX) to show that the whole term reduces.

- e is a well-typed value of form $c\,(\overline{fv})$, and $e_1 \ldots e_n$ are values. Hence, we have applied rule

$$\text{(OBJ-T)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash v_i : T_i^{\Lambda} \quad \forall i \in 1..n \\ \Delta \vdash T_i^{\Lambda} \leq T_i \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash c\,(\overline{fv}) : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fv} = \texttt{this}.f_1 = v_1\texttt{;}\ \ldots\ \texttt{this}.f_n = v_n\texttt{;} \\ \mathsf{defFields}(\Delta, c) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}$$

with $c = c^{\Lambda}$. We can apply rule

$$\text{(INVK)}\ \frac{}{c\,(\overline{fv})\,.m(\overline{v}) \underset{p}{\rightarrow} e[^{\overline{v}}/_{\overline{x}}][^{c\,(\overline{fv})}/_{\texttt{this}}]}\ \mathsf{mBody}(p, c, m) = \langle \overline{x}, e \rangle$$

Indeed:

  - $\mathsf{mBody}(p, c, m) = \langle \overline{x}, e \rangle$ holds by $\mathsf{mType}(\Delta, c^{\Lambda}, m) = \overline{T} \rightarrow T$ and $\mathsf{nonAbs}(\Delta, c)$,
  - $\mid \overline{e} \mid\ =\ \mid \overline{x} \mid$ holds, since from the second premise we have $\mid \overline{e} \mid\ =\ \mid \overline{T} \mid$, by $\Delta \vdash \overline{T} \leq \overline{T'}$ we have $\mid \overline{T} \mid\ =\ \mid \overline{T'} \mid$, and by $\vdash p : pt$, (PROGRAM-T), (BASIC-T), (METHOD-T) and definition of mType we have $\mid \overline{x} \mid\ =\ \mid \overline{T'} \mid$.

Case $\text{(NEW-T)}\ \dfrac{\begin{array}{c}\Delta;\Gamma \vdash \overline{e} : \overline{T^{\Lambda}} \\ \Delta \vdash \overline{T^{\Lambda}} \leq \overline{T}\end{array}}{\Delta;\Gamma \vdash \texttt{new}\ c\,(\overline{e}) : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \mathsf{kType}(\Delta, c) = \overline{T}\end{array}$

Assume $\overline{e} = e_1 \ldots e_n$. From the premise by the inductive hypothesis we have two cases:

- $e_i \underset{p}{\rightarrow} \_$ for some $i \in 1..n$. We can apply (CTX) to show that the whole term reduces.
- $e_1 \ldots e_n$ are values. We can apply rule

$$\text{(OBJ-CREATION)}\ \frac{}{\texttt{new}\ c\,(\overline{v}) \underset{p}{\rightarrow} c\,(\overline{fe}[^{\overline{v}}/_{\overline{x}}])}\ \begin{array}{l}\mathsf{kBody}(p, c) = \langle \overline{x}, \overline{fe} \rangle \\ \mathsf{nonAbs}(p, c)\end{array}$$

Indeed:

  - $\mathsf{kBody}(p, c) = \langle \overline{x}, \overline{fe} \rangle$ holds by $\Delta;\Gamma \vdash \texttt{new}\ c\,(\overline{e}) : c$,
  - $\mathsf{nonAbs}(p, c)$ holds by (PROGRAM-T), (BASIC-T) and $\mathsf{nonAbs}(\Delta, c)$,
  - $\mid \overline{e} \mid\ =\ \mid \overline{x} \mid$ holds, since from the premise we have $\mid \overline{e} \mid\ =\ \mid \overline{T} \mid$, by $\Delta \vdash \overline{T} \leq \overline{T'}$ we have $\mid \overline{T} \mid\ =\ \mid \overline{T'} \mid$ and by $\vdash p : pt$, (PROGRAM-T), (BASIC-T), (CONS-T) and definition of kType we have $\mid \overline{x} \mid\ =\ \mid \overline{T'} \mid$.

Case $\text{(OBJ-T)}\ \dfrac{\begin{array}{c}\Delta;\Gamma \vdash e_i : T_i^{\Lambda} \quad \forall i \in 1..n \\ \Delta \vdash T_i^{\Lambda} \leq T_i \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash c\,(\overline{fe}) : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fe} = \texttt{this}.f_1 = e_1\texttt{;}\ \ldots\ \texttt{this}.f_n = e_n\texttt{;} \\ \mathsf{defFields}(\Delta, c) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}$

From the premise by the inductive hypothesis we have two cases:

- $e_i \underset{p}{\rightarrow} \_$ for some $i \in 1..n$. We can apply (CTX) to show that the whole term reduces.
- $e_1, \ldots, e_n$ are well-typed values. Hence, the whole term is a value. □

The following two lemmas are needed to prove subject reduction.

Lemma 3. *If* $\Delta \vdash c_2^\Lambda \leq c_1^\Lambda$ *and* $\mathsf{mType}(\Delta, c_1^\Lambda, n) = mt$,
then $\mathsf{mType}(\Delta, c_2^\Lambda, n) = mt$.

Proof. By straightforward induction over the definition of $\Delta \vdash c_2^\Lambda \leq c_1^\Lambda$. □

Lemma 4 (FJIG$_0$ substitution lemma)**.** *If* $\vdash p : pt$ *then:*

1. *If* $pt; \Gamma \vdash e_1 : T_1$, $pt; \Gamma \vdash e_2 : T_2$, $pt \vdash T_2 \leq T_1$, *and* $pt; \Gamma \vdash \mathcal{E}^{\mathrm{r}}[\![e_1]\!] : T_1'$, *then, for some* T_2', $pt; \Gamma \vdash \mathcal{E}^{\mathrm{r}}[\![e_2]\!] : T_2'$ *and* $pt \vdash T_2' \leq T_1'$.
2. *If* $pt; \Gamma, \overline{x}{:}\overline{T}_1 \vdash e : T_1$, $pt \vdash \overline{T}_2 \leq \overline{T}_1$ *and* $pt; \Gamma \vdash \overline{e} : \overline{T}_2$
then, for some T_2, $pt; \Gamma \vdash e[\overline{e}/\overline{x}] : T_2$ *and* $pt \vdash T_2 \leq T_1$.

Proof.

1. By straightforward structural induction on $\mathcal{E}^{\mathrm{r}}$. We show only two cases:

   Case □
   $\mathcal{E}^{\mathrm{r}}[\![e_i]\!] = e_i$ and $T_i = T_i'$ for $i \in 1..2$.

   Case $\mathcal{E}^{\mathrm{r}}.f$
   We have $pt; \Gamma \vdash \mathcal{E}^{\mathrm{r}}[\![e_1]\!] : T_1''$ by $pt; \Gamma \vdash (\mathcal{E}^{\mathrm{r}}.f)[\![e_1]\!] : T_1'$ and (FIELD-T), we have $pt; \Gamma \vdash \mathcal{E}^{\mathrm{r}}[\![e_2]\!] : T_2''$ and $pt \vdash T_2'' \leq T_1''$ by the inductive hypothesis, so we can apply Lemma 3. Finally, we get the thesis by (FIELD-T).

2. Analogously, by straightforward structural induction on the definition of e. □

Theorem 5 (FJIG$_0$ subject reduction)**.** *If* $\vdash p : pt$, $pt; \emptyset \vdash e : T$ *and* $e \underset{p}{\rightarrow} e'$ *then, for some* T', $pt; \emptyset \vdash e' : T'$ *and* $pt \vdash T' \leq T$.

Proof. By induction on the reduction rules.

Case $\dfrac{e_1 \underset{p}{\rightarrow} e_2}{\mathcal{E}^{\mathrm{r}}[\![e_1]\!] \underset{p}{\rightarrow} \mathcal{E}^{\mathrm{r}}[\![e_2]\!]}$ (CTX)

We get the thesis by Lemma 4-(1).

Case $\dfrac{}{c\,(\overline{fv})\,.f \underset{p}{\rightarrow} v}\ \overline{fv}(f) = v$ (FIELD-ACCESS)

typed by $\dfrac{\Delta;\Gamma \vdash c\,(\overline{fv})\ :\ c}{\Delta;\Gamma \vdash c\,(\overline{fv})\,.f\ :\ T}\ (\text{FIELD-ACCESS-T})\ \ \mathsf{mType}(\Delta, c, f) = T$

In this case, $c\,(\overline{fv})$ is typed by rule

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash v_i\ :\ T_i^{\Lambda} \quad \forall i \in 1..n \\ \Delta \vdash\ T_i^{\Lambda} \leq T_i \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash c\,(\overline{fv})\ :\ c}\ (\text{OBJ-T})\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fv} = \mathbf{this}.f_1 = v_1\mathbf{;}\ \ldots \mathbf{this}.f_n = v_n\mathbf{;} \\ \mathsf{defFields}(\Delta, c) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}$$

and, since $f \in \mathsf{dom}(\overline{fv})$, we have $f = f_i$ and $v = v_i$. By (PROGRAM-T), (BASIC-T) and definition of mType, we have $T = T_i$. Finally, we get the thesis by by $\Delta;\Pi \vdash e_i\ :\ T_i$ and $\Delta \vdash\ T_i^{\Lambda} \leq T_i$.

Case $\dfrac{}{c\,(\overline{fv})\,.m(\overline{v})\ \underset{p}{\rightarrow}\ e[\overline{v}/\overline{x}][c\,(\overline{fv})\,/\mathbf{this}]}\ (\text{INVK})\ \ \mathsf{mBody}(p, c, m) = \langle \overline{x}, e\rangle$

typed by $\dfrac{\begin{array}{c}\Delta;\Gamma \vdash c\,(\overline{fv})\ :\ c \\ \Delta;\Gamma \vdash \overline{e}\ :\ \overline{T^{\Lambda}} \\ \Delta \vdash\ \overline{T^{\Lambda}} \leq \overline{T}\end{array}}{\Delta;\Gamma \vdash c\,(\overline{fv})\,.m\,(\overline{e})\ :\ T}\ (\text{INVK-T})\ \ \mathsf{mType}(\Delta, c, m) = \overline{T} \rightarrow T$

In this case, $c\,(\overline{fv})$ is typed by rule

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash v_i\ :\ T_i^{\Lambda} \quad \forall i \in 1..n \\ \Delta \vdash\ T_i^{\Lambda} \leq T_i \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash c\,(\overline{fv})\ :\ c}\ (\text{OBJ-T})\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fv} = \mathbf{this}.f_1 = v_1\mathbf{;}\ \ldots \mathbf{this}.f_n = v_n\mathbf{;} \\ \mathsf{defFields}(\Delta, c) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}$$

and, by (PROGRAM-T) and (BASIC-T) we have that rule

$$\dfrac{\Delta;\mathbf{this}{:}\Lambda, x_1{:}T_1, \ldots, x_n{:}T_n \vdash\ e\ :\ T^{\Lambda} \quad \Delta \vdash\ T^{\Lambda} \leq T_0}{\Delta \vdash\ mh\,\{\,\mathbf{return}\ e\mathbf{;}\,\}\ :\ (m{:}\epsilon\ T_1 \ldots\ T_n \rightarrow T_0)}\ (\text{METHOD-T})\ \begin{array}{l}mh = T_0\, m(T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n\end{array}$$

can be applied. Finally, we get the thesis by Lemma 4-(2) and subtyping rule (TRANS).

Case $\dfrac{}{\mathbf{new}\ c\,(\overline{v})\ \underset{p}{\rightarrow}\ c\,(\overline{fe}[\overline{v}/\overline{x}])}\ (\text{OBJ-CREATION})\ \begin{array}{l}\mathsf{kBody}(p, c) = \langle \overline{x}, \overline{fe}\rangle \\ \mathsf{nonAbs}(p, c)\end{array}$

typed by $\dfrac{\begin{array}{c}\Delta;\Gamma \vdash \overline{v}\ :\ \overline{T^{\Lambda}} \\ \Delta \vdash\ \overline{T^{\Lambda}} \leq \overline{T}\end{array}}{\Delta;\Gamma \vdash \mathbf{new}\ c\,(\overline{v})\ :\ c}\ (\text{NEW-T})\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \mathsf{kType}(\Delta, c) = \overline{T}\end{array}$

In this case, by (PROGRAM-T) and (BASIC-T), we have that

$$\dfrac{\begin{array}{c}\Delta;x_1{:}T_1', \ldots, x_n{:}T_n' \vdash\ e_i\ :\ T_i'' \quad \forall i \in 1..k \\ \Delta \vdash\ T_i'' \leq T_i \quad \forall i \in 1..k\end{array}}{\Delta \vdash\ kh\,\{\,\overline{fe}\,\}\ :\ T_1' \ldots\ T_n'}\ (\text{CONS-T})\ \begin{array}{l}kh = \mathbf{constructor}\,(\,T_1'\ x_1, \ldots, T_n'\ x_n) \\ \overline{fe} = \mathbf{this}.f_1 = e_1\mathbf{;}\ \ldots \mathbf{this}.f_k = e_k\mathbf{;} \\ \mathsf{exists}(\Delta, T_i') \\ \mathsf{defFields}(\Delta, \Lambda) = f_1{:}\epsilon\ T_1, \ldots, f_k{:}\epsilon\ T_k \quad \forall i \in 1..n\end{array}$$

Hence, $c\,(\overline{fe})$ can be typed by

$$\text{(OBJ-T)}\ \frac{\begin{array}{c}\Delta;\Gamma \vdash e_i : T_i^{\Lambda} \quad \forall i \in 1..n \\ \Delta \vdash T_i^{\Lambda} \leq T_i \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash c\ (\overline{fe})\ :\ c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \overline{fe} = \texttt{this}.f_1 = e_1\texttt{;}\ \ldots\ \texttt{this}.f_n = e_n\texttt{;} \\ \mathsf{defFields}(\Delta, c) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}$$

since all the (OBJ-T) premises and the subtype requirements in side conditions can be verified from (NEW-T) premises by Lemma 4-(2) and subtyping rule (TRANS). □

Chapter 3

Composition operators

> *One Ring to rule them all, One Ring to find them, One Ring to bring them all and in the darkness bind them*
>
> (J. R. R. Tolkien - The Lord of the Rings)

In this chapter, we introduce *composition operators* and *flattening*. In FJIG_0 class declarations were associations between class names and basic classes. In the language FJIG_1 described here, class declarations are associations between class names and *class expressions*, which are constructed on top of basic classes and class names by four composition operators: *sum*, *restrict*, *alias* and *redirect*.

The semantics of an FJIG_1 program can be given in two ways: *flattening* semantics, that is, by translation into an FJIG_0 program, where all composition operators have been performed, and *direct* semantics, that is, by providing a direct execution model which formalizes dynamic look-up. In this thesis we focus only on the flattening semantics. For a comparison between flattening and direct semantics, and a proof of equivalence, refer to [LSZ09b]. We call *flattening* the process that, step by step, performs the composition operators and produces a *flat* program, that is, a FJIG_0 program. Flattening can be seen as a precompilation step, or a linking step if operators act on binaries rather than sources.

We first informally introduce class composition operators by means of some examples in Section 3.1, then in Section 3.2 we give the formal syntax and flattening semantics, and in Section 3.3 the type system. Finally, in Section 3.4 we prove the soundness of the type system w.r.t. flattening.

3.1 Examples

In the following, to write more readable examples, we assume a superset of the language used in the formal description, including primitive types **int**, **boolean** and their operations, and the conditional operator.

The simplest form of class expressions are basic classes and class names, as shown in the following example.

```
A = abstract{
  abstract int m1();
  int m2(){return this.m1() + 1;}
}
B = A
```

By a very intuitive copy semantics, this program is reduced by flattening to the following one:

```
A = // as before
B = abstract{
  abstract int m1();
  int m2(){return this.m1() + 1;}
}
```

Indeed, as already mentioned, in FJIG a class declaration just introduces a name for a class expression. Hence, the semantics of our language fully achieves the *substitutability principle*: a class expression can be replaced by an equivalent one in any context without affecting the overall semantics. This is different from what happens, e.g., in Java, where, when writing:

```
class B extends A { ... }
```

the use of name `A` is relevant, not only since inheritance implies nominal subtyping, but also, e.g., in case `A` has static members. An alternative syntax[1], stressing that class names have a copy semantics, could be

```
B = copy A
```

Compound class expressions can be constructed using *composition operators*. For instance, a concrete class can be obtained by applying the *sum* operator as follows:

```
Sum = A [+]
  abstract{
    abstract int m2();
```

[1] We will adopt this alternative syntax in Chapter 6 and later, to distinguish the constant meta-expression (literal) `A` of type **cpath** from the expression **copy** `A` of type **class**. Until then, we prefer to stick to the more natural and light syntax above.

```
    int m1() { return 1 + this.m2(); }
  }
```

This declaration is reduced to the following:

```
Sum = {
  int m1(){return 1 + this.m2();}
  int m2(){return this.m1() + 1;}
}
```

Conflicting definitions for the same member are not permitted, whereas **abstract** members with the same name are shared. Note that one can equivalently write the program in this way:

```
C = abstract{
  abstract int m2();
  int m1() { return 1 + this.m2(); }
}
Sum = A [+] C
```

Recall that a basic class defines one constructor that specifies a sequence of parameters and a sequence of initialization expressions, one for each non abstract field. In order to be composed by the sum operator, two classes should provide a constructor with the same parameter list. The effect is that the resulting class provides a constructor with the same parameter list, that executes both the original constructors, as shown in the following example:

```
A1 = abstract{
  abstract int f1;
  int f2;
  constructor(int x){
    this.f2 = x;
  }
  int m1(){return this.f1 + this.f2;}
  abstract int m2();
}
C1 =
  abstract{
    int f1;
    abstract int f2;
    constructor(int x){
      this.f1 = x + 1;
    }
  } [+] A1
```

which is equivalent to

```
A1 = // as before
```

```
C1 = abstract{
  int f1;
  int f2;
  constructor(int x){
    this.f1 = x + 1;
    this.f2 = x;
  }
  int m1(){return this.f1 + this.f2;}
  abstract int m2();
}
```

Classes composed by sum can share the same field, provided it is defined in (at most) one. Note that this corresponds to *sharing* fields as in, e.g., [BDNW08]; however, in our framework we do not need an ad-hoc notion.

Besides sum, FJIG$_1$ provides three other primitive composition operators: *restrict*, *alias*, and *redirect*, which take as arguments a class and one or two (member) names.

Restrict This operator removes a definition, making the corresponding member abstract. For example,

```
A1 = abstract{// the same class as before
  abstract int f1;
  int f2;
  constructor(int x){
    this.f2 = x;
  }
  int m1(){return this.f1 + this.f2;}
  abstract int m2();
}
RestrictM1 = A1 [restrict m1]
```

is reduced to

```
A1 = // as before
RestrictM1 = abstract{
  abstract int f1;
  int f2;
  constructor(int x){
    this.f2 = x;
  }
  abstract int m1();
  abstract int m2();
}
```

When a field is restricted, its corresponding initialization expression is removed as well, as shown in the following example:

```
A1 = // as before
RestrictF2= A1[restrict f2]
```

which is reduced to

```
A1 = // as before
RestrictF2 = abstract{
  abstract int f1;
  abstract int f2;
  constructor(int x){}
  int m1(){return this.f1 + this.f2;}
  abstract int m2();
}
```

On top of restrict and sum, we can define a useful derived operator: **override**. This operator is a variant of sum where conflicts are allowed and the left argument has the precedence. It can be defined in the following way:

```
C1[override]C2 ≡
  C1[+](C2[restrict n1] ... [restrict nk])
```

where **restrict** is applied to all fields or methods with the same name `ni` defined in both `C1` and `C2`.

The above encoding, originally defined in [Bra92], shows that standard single inheritance can be recovered as a special case of composition. In other words, quoting [Bra92], "the distinction between single and multiple inheritance disappears", provided "to treat the name collisions as errors", also in the diamond case, otherwise modularity is broken.

Alias This operator duplicates the declaration of an existing field or method for another field or method of the same class. In the following example, a definition is added for an abstract method:

```
A = abstract{
  abstract int m1();
  int m2(){return this.m1() + 1;}
}
Alias = A[alias m2 to m1]
```

is reduced to

```
A = // as before
Alias = {
```

```
  int m1(){return this.m1() + 1;}
  int m2(){return this.m1() + 1;}
}
```

The method body is duplicated, rather than just invoked, so that, in case the implementation of the original method is changed, the aliased one keeps the original semantics. Note also that, since the original method contains a reference to `m1`, the abstract method becomes a directly recursive method.

In the next example, instead, a new declaration is added:

```
Alias = // as before
Alias2 = Alias[alias m1 to m3]
```

is reduced to

```
Alias = // as before
Alias2 =  {
  int m1(){return this.m1() + 1;}
  int m2(){return this.m1() + 1;}
  int m3(){return this.m1() + 1;}
}
```

Note that the reference to `m1` is not affected, hence, whereas `m1` is a directly recursive method, `m3` is not.

It is possible to combine alias with override to emulate *super* calls, that is:

- super calls in the heir are encoded as calls to abstract methods with a special name, for example a super call to a method named `m` can be encoded by e`.super_m(...)`,
- the methods of the parent that occur in super calls in the heir are aliased to the corresponding special name.

Formally:

```
C1[overrideSuper m1 ... mn]C2 ≡
 C1[override](C2[alias m1 to super_m1] ... [alias mn to super_mn])
```

where `m1`, ..., `mn` are the method names occurring in super calls. The next example shows an application of such operator:

```
Alias = // as before
C = {
  int m2(){return this.super_m2()*2;}
```

```
  abstract int super_m2():
} [overrideSuper m2] Alias
```

This is equivalent to

```
Alias = // as before
C = {
  int m1(){return this.m1() + 1;}
  int m2(){return this.super_m2()*2;}
  int super_m2(){return this.m1() + 1;}
}
```

In summary, Java *extends* can be encoded using **override** to provide left preferential composition, **alias** to emulate *super* calls and **implements** declarations to provide subtyping.

Alias can also be uniformly used to duplicate abstract declarations and defined fields; in the latter case the corresponding field expression inside the constructor body is duplicated as well. For example,

```
A1 = abstract{
  abstract int f1;
  int f2;
  constructor(int x){
    this.f2 = x;
  }
  int m(){return this.f1 + this.f2;}
  abstract int m2();
}
Alias3 = A1 [alias f1 to f3][alias m2 to m3][alias f2 to f1]
```

is reduced to

```
A1 = // as before
Alias3 = abstract{
  int f1;
  int f2;
  abstract int f3;
  constructor(int x){
    this.f1 = x;
    this.f2 = x;
  }
  int m(){return this.f1 + this.f2;}
  abstract int m2();
  abstract int m3();
}
```

Redirect This operator replaces references with receiver **`this`** (**`this`** references for short) in method bodies or field expressions, to a member name by a different name, and removes its declaration. For example,

```
A = abstract{
  abstract int m1();
  int m2(){return this.m1() + 1;}
}
Redirect = A[redirect m1 to m2]
```

is reduced to

```
A = // as before
Redirect = {
  int m2() { return this.m2() + 1; }
}
```

Of course redirect can be also used over defined methods or fields, as shown in the following example:

```
A1 = abstract{
  abstract int f1;
  int f2;
  constructor(int x){
    this.f2 = x;
  }
  int m1(){return this.f1 + this.f2;}
  abstract int m2();
}
Redirect2 = A1 [redirect f2 to f1][redirect m1 to m2]
```

which is reduced to

```
A1 = // as before
Redirect2 = abstract{
  abstract int f1;
  constructor(int x){}
  abstract int m2();
}
```

Note that members `f2` and `m1` are no longer present in class `Redirect2`, and the initialization expression for `f2` inside the constructor body is removed as well.

In the languages presented in this thesis, the only way to remove a member from a class is to redirect it to another one of the same type. This restriction is needed since some method imple-

mentation could refer to it. In an extension with private members, like in [LSZ09b], operators can be added which support member hiding.

Since the operator **redirect** removes a member, its application could possibly violate supertype requirements. However, this is prevented by the type system. For instance, in the following example, BB is ill-typed, while CC is accepted.

```
A = {int m1(){return 1;}}
B = implements A{
  int m1(){return this.m2();}
  int m2(){return 2;}
}
C = {
  int m1(){return this.m2();}
  int m2(){return 2;}
}
BB = B[redirect m1 to m2]// ill-typed
CC = C[redirect m1 to m2]// ok
```

Note that the only difference between B and C is in the implementation of A. In other words, class BB is correct in itself, but cannot be a subtype of class A.

These four primitive operators are a minimal, yet very expressive set. Sum, restrict and alias are very similar to the trait operators defined in [DNS+06], with the difference that our version uniformly works on fields as well. Restrict is called *exclude* in trait literature; here we prefer to stick to the original name in Jigsaw [Bra92]. Other trait-based languages [BDG08, RT07] also include a *rename* operator. We prefer to have the *redirect* operator since it is conceptually more primitive, and it can be used to express many different forms of renaming.

For instance, the following simple encoding:

```
C[rename nOld to nNew]  ≡
  C[alias nOld to nNew][redirect nOld to nNew]
```

defines a rename operator which can be applied only if the old name nOld is present in C, and, if the new name nNew is present as well, then at most one can be non abstract and they must have the same type. Moreover, the two names must be different.

3.2 Syntax and flattening

Syntax In Figure 3.1 we show the syntax extension that turns FJIG$_0$ into FJIG$_1$. In this way, a program p becomes a set of associations from class names c to class expressions ce, constructed on top of basic classes and class names by operators sum, restrict, alias and redirect.

ce	$::=$		class expression
		b	basic class
		$\mid c$	class name
		$\mid ce_1\,[+]\,ce_2$	sum
		$\mid ce[\texttt{restrict}\ n]$	restrict
		$\mid ce[\texttt{alias}\ n^s\ \texttt{to}\ n^t]$	alias
		$\mid ce[\texttt{redirect}\ n^s\ \texttt{to}\ n^t]$	redirect

Figure 3.1: FJIG$_1$ syntax for class expressions

Flattening rules Figure 3.2 contains the rules defining the *flattening* relation $p_1 \rightarrow p_2$.

Rule (CTX) reduces the whole program by applying a reduction step to any of the class definitions. Here, $\mathcal{CE}$ denotes the standard one-hole context for class expressions. The flattening relation for class expression is of the form $ce_1 \underset{p}{\rightarrow} ce_2$.

Rule (CLASS-NAME) can be applied when the class expression occurring in position c in p is a basic class (FJIG$_1$ has a call-by-value semantics).

In rule (SUM), when the arguments of the sum operator are two basic classes, the operator can be applied, obtaining the sum of the basic classes, denoted by $b_1 \oplus b_2$. This sum is well-defined only if the arguments have the same constructor header and, for each member n declared in both arguments, the two declarations have the same kind (field or method) and type and at most one is non abstract. In this case, the resulting basic class has modifier non abstract only if the result has no abstract members, the union of the supertypes, the same constructor header, the (necessarily disjoint) union of the field expressions, and the union of the declarations, where two declarations for the same name are merged by keeping the non abstract, if any. Formally:

- If $b_i = \mu_i\ \texttt{implements}\ \overline{c}_i\ \{\, kh\,\{\,\overline{fe}_i\,\}\ \overline{d}_i\,\}$, then
  $b_1 \oplus b_2 = \mu\ \texttt{implements}\ \overline{c}_1\ \overline{c}_2\ \{\, kh\,\{\,\overline{fe}_1\ \overline{fe}_2\,\}\ \overline{d}_1 \oplus \overline{d}_2\,\}$
  where: $\mu = \epsilon$ iff $\overline{d}_1 \oplus \overline{d}_2 = \overline{\epsilon\ T\ f\texttt{;}}\ \overline{T\ m(\_)\,\{\,\texttt{return\_;}\,\}}$

- $\overline{d}_1 \oplus \overline{d}_2$ is defined by:
  - $\mathsf{dom}(\overline{d}_1 \oplus \overline{d}_2) = \mathsf{dom}(\overline{d}_1) \cup \mathsf{dom}(\overline{d}_2)$
  - $(\overline{d}_1 \oplus \overline{d}_2)(n) = \begin{cases} \overline{d}_1(n) & \text{if } n \in \mathsf{dom}(\overline{d}_1) \setminus \mathsf{dom}(\overline{d}_2) \\ \overline{d}_2(n) & \text{if } n \in \mathsf{dom}(\overline{d}_2) \setminus \mathsf{dom}(\overline{d}_1) \\ \overline{d}_1(n) \oplus \overline{d}_2(n) & \text{if } n \in \mathsf{dom}(\overline{d}_2) \cap \mathsf{dom}(\overline{d}_2) \end{cases}$
  - $\texttt{abstract}\ mh\texttt{;} \oplus \texttt{abstract}\ mh\texttt{;} = \texttt{abstract}\ mh\texttt{;}$

$$\mathcal{CE} \;::=\; \Box \mid \mathcal{CE}\,[+]\,ce \mid ce\,[+]\,\mathcal{CE} \mid \mathcal{CE}[\mathbf{restrict}\ n] \mid \mathcal{CE}[\mathbf{alias}\ n^s\ \mathbf{to}\ n^t] \mid \mathcal{CE}[\mathbf{redirect}\ n^s\ \mathbf{to}\ n^t]$$

$p_1 \rightarrow p_2$

$$\text{(CTX)}\ \frac{ce_1 \underset{p}{\rightarrow} ce_2}{p\,(c = \mathcal{CE}[\![ce_1]\!]) \rightarrow p\,(c = \mathcal{CE}[\![ce_2]\!])}$$

$ce_1 \underset{p}{\rightarrow} ce_2$

$$\text{(CLASS-NAME)}\ \frac{}{c \underset{p}{\rightarrow} b}\ b = p(c) \qquad \text{(SUM)}\ \frac{}{b_1\,[+]\,b_2 \underset{p}{\rightarrow} b_1 \oplus b_2}$$

$$\text{(RESTRICT)}\ \frac{}{b[\mathbf{restrict}\ n] \underset{p}{\rightarrow} b \ominus n \oplus \mathsf{abs}(d)}\ d = \mathsf{dec}(b, n)$$

$$\text{(ALIAS)}\ \frac{}{b[\mathbf{alias}\ n^s\ \mathbf{to}\ n^t] \underset{p}{\rightarrow} b' \oplus \mathsf{named}(n^t, d)}\ \begin{array}{l} d = \mathsf{dec}(b, n^s) \\ \mathsf{constr}(b) = kh\,\{\overline{fe}\} \\ b' = \begin{cases} b \oplus \mathbf{this}.n^t = e; & \text{if } \overline{fe}(n^s) = e \\ b & \text{if } n^s \notin \mathsf{dom}(\overline{fe}) \end{cases} \end{array}$$

$$\text{(REDIRECT)}\ \frac{}{b[\mathbf{redirect}\ n^s\ \mathbf{to}\ n^t] \underset{p}{\rightarrow} (b \ominus n^s)[n^s \rightsquigarrow n^t]}\ \begin{array}{l} n^s \in \mathsf{names}(b) \\ n^s \neq n^t \end{array}$$

Figure 3.2: FJIG$_1$ flattening rules

- $\texttt{abstract}\ mh\texttt{;} \oplus mh\,\{\,\texttt{return}\ e\texttt{;}\,\} =$
  $mh\,\{\,\texttt{return}\ e\texttt{;}\,\} \oplus \texttt{abstract}\ mh\texttt{;} = mh\,\{\,\texttt{return}\ e\texttt{;}\,\}$
- $\texttt{abstract}\ T\ f\texttt{;} \oplus \mu\ T\ f\texttt{;} = \mu\ T\ f\texttt{;} \oplus \texttt{abstract}\ T\ f\texttt{;} = \mu\ T\ f\texttt{;}$

In rule (RESTRICT), the operator replaces the definition of member n by the corresponding abstract declaration. We denote by $b \ominus n$ the basic class obtained from b by removing member n (if n is a field, then its initialization expression is removed as well), by $b \oplus d$ the basic class obtained from b by adding declaration d, by $\mathsf{abs}(d)$ the abstract version of the declaration d, by $\mathsf{dec}(b, n)$ the declaration for name n in basic class b, by $\mathsf{named}(n, d)$ the declaration equal to d except that the declared name is n. Formally:

- $\mu\ \texttt{implements}\ \overline{c}\ \{\,kh\,\{\,\overline{fe}\,\}\ \overline{d}\,\} \ominus n = \mu'\ \texttt{implements}\ \overline{c}\ \{\,kh\,\{\,\overline{fe}\backslash n\,\}\ \overline{d}\backslash n\,\}$
  $\mu = \epsilon$ iff $\overline{d}\backslash n = \overline{\epsilon\ T\ f\texttt{;}\ T\ m(\_)\,\{\,\texttt{return}\ \_\texttt{;}\,\}}$
- $\mu\ \texttt{implements}\ \overline{c}\ \{\,k\ \overline{d}\,\} \oplus d = \mu'\ \texttt{implements}\ \overline{c}\ \{\,k\ \overline{d} \oplus d\,\}$
  $\mu' = \epsilon$ iff $\overline{d} \oplus d = \overline{\epsilon\ T\ f\texttt{;}\ T\ m(\_)\,\{\,\texttt{return}\ \_\texttt{;}\,\}}$
- $\mathsf{abs}(\mu\ T\ f\texttt{;}) = \texttt{abstract}\ T\ f\texttt{;}$
  $\mathsf{abs}(\texttt{abstract}\ mh\texttt{;}) = \mathsf{abs}(mh\,\{\,\texttt{return}\ e\texttt{;}\,\}) = \texttt{abstract}\ mh\texttt{;}$
- $\mathsf{dec}(ch\ \{\,k\ \overline{d}\ d\,\}, n) = d$ with $d = \mathsf{named}(n, d')$
- $\mathsf{named}(n, \mu\ T\ f\texttt{;}) = \mu\ T\ n\texttt{;}$
  $\mathsf{named}(n, \texttt{abstract}\ T\ m(\overline{T\ x})\texttt{;}) = \texttt{abstract}\ T\ n(\overline{T\ x})\texttt{;}$
  $\mathsf{named}(n,\ T\ m(\overline{T\ x})\,\{\,\texttt{return}\ e\texttt{;}\,\}) = T\ n(\overline{T\ x})\,\{\,\texttt{return}\ e\texttt{;}\,\}$

In rule (ALIAS), the operator duplicates the declaration of an existing member n^s, for "source", for another member n^t, for "target". If n^s is a field, then the initialization expression is duplicated as well. We denote by $\mathsf{constr}(b)$ the constructor of basic class b and by $b \oplus fe$ the basic class obtained from b by adding field expression fe in the constructor. Formally:

- $\mathsf{constr}(b) = k$ if $b = \_\,\{\,k\ \_\,\}$
- $ch\ \{\,kh\,\{\,\overline{fe}\,\}\ \overline{d}\,\} \oplus fe = ch\ \{\,kh\,\{\,\overline{fe}\ fe\,\}\ \overline{d}\,\}$

In rule (REDIRECT), the operator replaces references with receiver **this** to existing member n^s by references to n^t. The declaration of n^s is removed, so n^s and n^t have to be different. We denote by $\mathsf{names}(b)$ the set of names declared in b, and by $b[n^s \rightsquigarrow n^t]$ the basic class obtained from b by replacing n^s with n^t in field accesses/method invocations whose receiver is **this**. Formally:

- $\mathsf{names}(b) = \mathsf{dom}(\overline{d})$ if $b = \_\,\{\,\_\ \overline{d}\,\}$

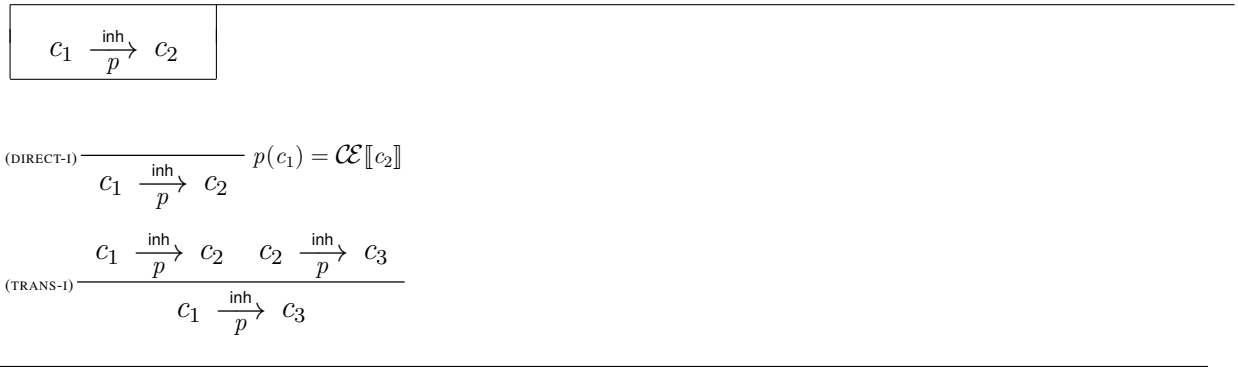

Figure 3.3: FJIG$_1$ generalized inheritance relation

- $b[n^s \rightsquigarrow n^t]$ propagates in all the subterms, and

$$\texttt{this}.f[n^s \rightsquigarrow n^t] = \begin{cases} \texttt{this}.n^t & \text{if } f = n^s \\ \texttt{this}.f & \text{otherwise} \end{cases}$$

$$\texttt{this}.m\,(\overline{e})\,[n^s \rightsquigarrow n^t] = \begin{cases} \texttt{this}.n^t\,(\overline{e}) & \text{if } m = n^s \\ \texttt{this}.m\,(\overline{e}) & \text{otherwise} \end{cases}$$

Generalized inheritance relation Analogously to what happens with standard inheritance, we require the generalized inheritance relation $\xrightarrow[p]{\text{inh}}$, formally defined in Figure 3.3, to be acyclic for well-formed programs. This is necessary to ensure that flattening reduces a program to a flat program, that is, a FJIG$_0$ program, where every class expression is a basic class.

Note that if the generalized inheritance is cyclic, what happens is that flattening goes stuck, rather than do not terminate. This is due to the fact that we have a call by value strategy: in rule (CLASS-NAME) in Figure 3.2 substitution can be performed only if the class name denotes a value.

3.3 Type system

The type system for FJIG$_1$ is obtained from that for FJIG$_0$, defined in the previous chapter in Figures 2.3-2.6, by adding typing rules for composition operators, given in Figure 3.4. These typing rules are analogous to the corresponding flattening rules. Moreover, rule (REDIRECT-T) checks that the member types for source name n^s and target name n^t are different and that the resulting class type is well-formed. This check fails if member n^s could not be removed, since present in one of the supertypes. We use the following notations:

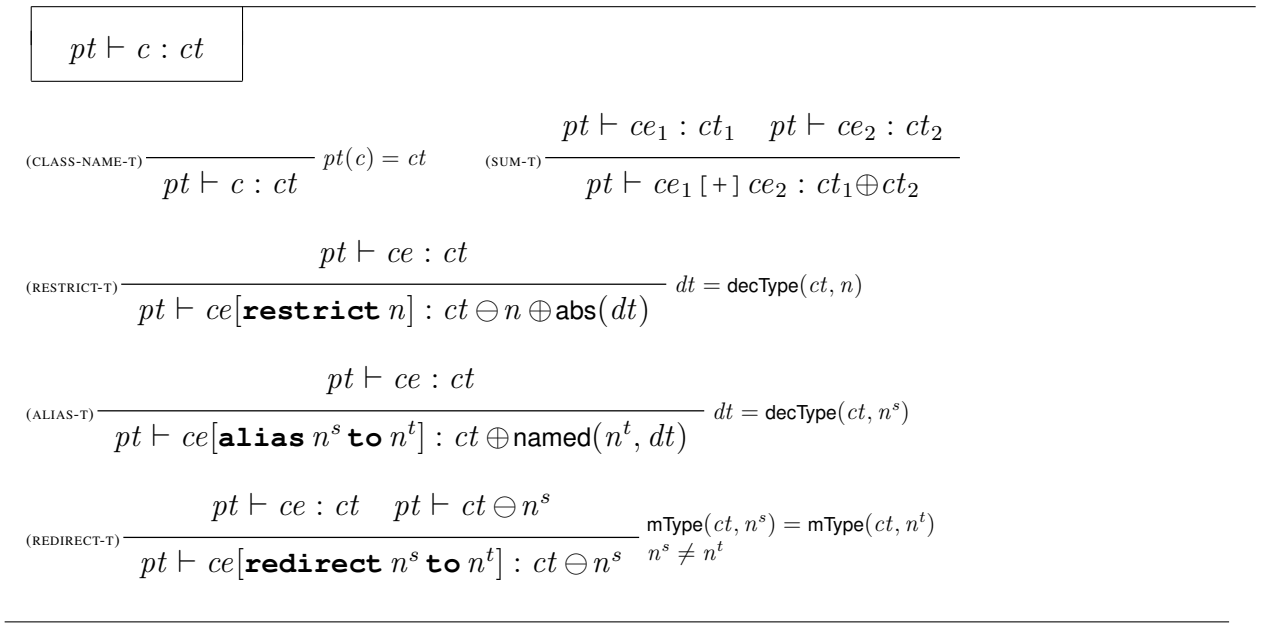

Figure 3.4: FJIG$_1$ typing rules

- $ct_1 \oplus ct_2$, $ct \ominus n$ and $ct \oplus dt$ are analogous to $b_1 \oplus b_2$, $b \ominus n$ and $b \oplus d$, respectively, but work over class types.
- $\mathsf{decType}(ct, n)$, $\mathsf{abs}(dt)$ and $\mathsf{named}(n, dt)$ are analogous to $\mathsf{dec}(b, n)$, $\mathsf{abs}(d)$ and $\mathsf{named}(n, d)$, respectively, but work over declaration types.
- $\mathsf{mType}(ct,n)$ is the member type of n in ct. Formally:
  $\mathsf{mType}(ct, n) = mt$ iff $\mathsf{decType}(ct, n) = n{:}\mu\, mt$

Note that we have chosen to express typing rules for FJIG$_1$ in a non algorithmic way for simplicity. However, they can be easily converted in a typechecking algorithm in two phases:

- first, for each declared class, we extract its class type ct. This is easy since for basic classes we have explicit type annotations, and for compound class expressions we can compute the resulting class type from the class types of the subexpressions, as shown in the typing rules. Of course this computation may fail, in case some side condition is not verified, for instance in `{abstract int f;} [+] {abstract Object f;}`
- second, using the Δ obtained by the first phase, we can check well-formedness of constructor and members, safety of the declared subtyping relations in implements clauses, and safety of the redirect operators (second premise of rule (REDIRECT-T) in Figure 3.4)

This approach smoothly generalizes to the language FJIG$_\star$ presented in Chapter 5.

3.4 Results

The type system is sound w.r.t. flattening, that is, a well-typed FJIG$_1$ program always reduces in some steps to a well-typed FJIG$_0$ program, as stated by the following theorem.

Theorem 6 (FJIG$_1$ soundness w.r.t. flattening)**.** *If* $\vdash p_1 : pt$*, then* $p_1 \xrightarrow{\star} p_2$ *for some* p_2 *flat program, and* $\vdash p_2 : pt$*.*

This soundness theorem follows from termination of flattening, progress, and subject reduction theorems below. Note that progress relies on the assumption that the generalized inheritance relation $\xrightarrow[p]{\mathsf{inh}}$ is acyclic.

In order to prove termination of flattening, we formally define the dimension of a class expression:

$\dim(b) = 0$
$\dim(c) = 1$
$\dim(ce_1 \,[+]\, ce_2) = \dim(ce_1) + \dim(ce_2) + 1$
$\dim(ce[\texttt{restrict\_}]) = \dim(ce) + 1$
$\dim(ce[\texttt{alias\_to\_}]) = \dim(ce) + 1$
$\dim(ce[\texttt{redirect\_to\_}]) = \dim(ce) + 1$

and we define $\dim(p)$ as:

$\dim(c_1 = ce_1 \ldots c_n = ce_n) = \dim(ce_1) + \ldots + \dim(ce_n)$

Lemma 7. *If* $ce_1 \xrightarrow[p]{} ce_2$ *then* $\dim(ce_1) > \dim(ce_2)$.

Proof. By cases on the flattening rules. We show only one case:

Case $\dfrac{}{b_1 \,[+]\, b_2 \xrightarrow[p]{} b_1 \oplus b_2}\,(\text{SUM})$

$\dim(b_1 \,[+]\, b_2) = 1$ and $\dim(b_1 \oplus b_2) = 0$.

The other cases are analogous. □

Theorem 8 (FJIG$_1$ termination of flattening)**.** *If* $p_1 \to p_2$ *then* $\dim(p_1) > \dim(p_2)$.

Proof. Only rule $\dfrac{ce_1 \xrightarrow[p]{} ce_2}{p\,(c = \mathcal{CE}[\![ce_1]\!]) \to p\,(c = \mathcal{CE}[\![ce_2]\!])}\,(\text{CTX})$

reduces a program. By Lemma 7, we have that $\dim(ce_1) > \dim(ce_2)$. We can conclude by definition of $\dim(p_1)$. □

In the next lemma we use the following definition of redex.

$r \quad ::= \quad b_1 \,[+]\, b_2 \mid b[\texttt{restrict}\ n] \mid b[\texttt{alias}\ n^s\ \texttt{to}\ n^t] \mid b[\texttt{redirect}\ n^s\ \texttt{to}\ n^t]$

Lemma 9 (FJIG$_1$ progress w.r.t. $ce_1 \underset{p}{\rightarrow} ce_2$)**.** *If* $\vdash p : pt$ *and* $pt \vdash r : ct$ *then* $r \underset{p}{\rightarrow} b$*, for some basic class* b*.*

Proof. By cases. We show only one case:

Case $r = b_1 \,[+]\, b_2$

typed by $\text{(SUM-T)}\dfrac{pt \vdash b_1 : ct_1 \quad pt \vdash b_2 : ct_2}{pt \vdash b_1 \,[+]\, b_2 : ct_1 \oplus ct_2}$

We get the thesis by application of rule $\text{(SUM)}\dfrac{}{b_1 \,[+]\, b_2 \underset{p}{\rightarrow} b_1 \oplus b_2}$, since $b_1 \oplus b_2$ is well-defined. Indeed, by (BASIC-T) and $ct_1 \oplus ct_2$ is defined, we know that the constructor of b_1 is equal to the constructor of b_2, and by definition of $\overline{dt}_1 \oplus \overline{dt}_2$ we know that $\overline{d}_1 \oplus \overline{d}_2$ is defined where $\overline{d}_i$ are the declarations of b_i and $\overline{dt}_i$ are the declaration types of ct_i. □

Theorem 10 (FJIG$_1$ progress)**.** *If* $\vdash p : pt$ *then* p *is flat or* $p \rightarrow \_$*.*

Proof. If p is not flat, then we have two cases:

Case $p = p'\,(c = \mathcal{CE}[\![r]\!])$

Since $\vdash p : pt$, by (PROGRAM-T) we know that $pt \vdash \mathcal{CE}[\![r]\!] : \_$, hence, since all typing rules require subcomponents to be well-typed, we also know by straightforward structural induction on $\mathcal{CE}$ that $pt \vdash r : \_$. We can conclude by Lemma 9 and applying rule $\text{(CTX)}\dfrac{ce_1 \underset{p}{\rightarrow} ce_2}{p\,(c = \mathcal{CE}[\![ce_1]\!]) \rightarrow p\,(c = \mathcal{CE}[\![ce_2]\!])}$.

Otherwise Since $\xrightarrow[p]{\text{inh}}$ is acyclic, $p = p'\,(c = \mathcal{CE}[\![c']\!])$ with $p'(c') = b$. We can conclude by applyng rules (CTX) and $\text{(CLASS-NAME)}\dfrac{}{c \underset{p}{\rightarrow} b}\ b = p(c)$. □

Lemma 11 (FJIG$_1$ substitution lemma)**.** *If* $\vdash p : pt$*,* $pt \vdash ce_1 : ct$*,* $pt \vdash ce_2 : ct$*, and* $pt \vdash \mathcal{CE}[\![ce_1]\!] : ct'$*, then* $pt \vdash \mathcal{CE}[\![ce_2]\!] : ct'$*.*

Proof. By straightforward structural induction on $\mathcal{CE}$. □

Lemma 12. *If* $\vdash p : pt$*,* $pt \vdash ce : ct$ *and* $ce \underset{p}{\rightarrow} ce'$ *then* $pt \vdash ce' : ct$*.*

Proof. By cases on the flattening rules. We show only one case:

Case $\text{(SUM)}\dfrac{}{b_1\,[+]\,b_2 \underset{p}{\rightarrow} b_1 \oplus b_2}$

typed by $\text{(SUM-T)}\dfrac{pt \vdash b_1 : ct_1 \quad pt \vdash b_2 : ct_2}{pt \vdash b_1\,[+]\,b_2 : ct_1 \oplus ct_2}$

We get the thesis by (BASIC-T), since $ct_1 \oplus ct_2$ is defined analogously to $b_1 \oplus b_2$. □

Theorem 13 (FJIG$_1$ subject reduction)**.** *If* $\vdash p_1 : pt$ *and* $p_1 \rightarrow p_2$ *then* $\vdash p_2 : pt$.

Proof. Only rule $\text{(CTX)}\dfrac{ce_1 \underset{p}{\rightarrow} ce_2}{p\,(c = \mathcal{CE}[\![ce_1]\!]) \rightarrow p\,(c = \mathcal{CE}[\![ce_2]\!])}$

reduces a program. We get the thesis by Lemma 12, Lemma 11 and (PROGRAM-T). □

Chapter 4

Meta-language for composition

> *Friends [...] I need your help again, let me use your powers so that I could be stronger. [...] Pursuit of the truth, faith in people, [...] I will show you what the human spirit is about!*
>
> (Ryo - Samurai Troopers)

In this chapter, we extend the class composition language described in previous chapters by a *meta-level.* That is, we define a language METAFJIG$_1$ where class definitions are first-class values which can be combined by using the four primitive operators of FJIG$_1$.

In METAFJIG$_1$, a *meta-program* is a sequence of class declarations which are associations between class names and arbitrary expressions. A meta-program is reduced to a conventional program, where right-hand sides of class declarations are basic classes, by a process which we call *compile-time execution*. In order to guarantee that compile-time execution, if terminating, produces a well-typed program, we use an incremental typechecking approach, where a meta-expression can be reduced only if it can be successfully typechecked with the current type information. We call the resulting process *checked compile-time execution.*

We first illustrate METAFJIG$_1$ and its meta-level features in Section 4.1, while in Section 4.2 we provide some more interesting examples showing the expressive power of the language. Section 4.3 and Section 4.4 contain the formal definition of the language. Section 4.5 formally defines checked compile-time execution. Formal statements and proofs of the good properties of our approach are presented in Section 4.6.

4.1 Examples

In the examples we will use a superset of the language used in the formal description, including primitive types **int** and **boolean**, predefined classes `Object`, `RuntimeException`[1] and `String`, arrays, **void** methods, statements **if**, **while**, **try**–**catch**, **for**–**each**, variable declaration and assignment.

Minimal example In METAFJIG$_1$, a class declaration associates a (meta-)expression of primitive type **class** with a class name. The simplest form of such an expression is a *basic class*, which is exactly like an FJIG$_0$ basic class. Since **class** is a primitive type of the language, a class definition can be the result of a method. For instance, in the following meta-program[2]

```
C = {
  class m(){
    return { int one(){return 1;} };
  }
}
D = new C().m()
```

method `m` returns a value of type **class**, that is, a basic class declaring the method `one`. Class `D` is defined by an expression that has to be evaluated in order to obtain the corresponding basic class.

For helping readability we emphasize in grey basic classes occurring in method bodies as meta-expressions.

More precisely, the definition of `D` is the value returned by the method `m` of `C`, so this program could be equivalently written as:

```
C = // as before
D = { int one(){return 1;} }
```

Note that a basic class must be closed w.r.t. variables, that is, cannot refer to the parameter names of the surrounding method. This avoids problems of scope-extrusion when (meta-)code is used. For instance, the following basic class is ill-formed:

```
{
  class m(class x){
    return { class n(){return x;} };
  }
```

[1] In Java exceptions are checked or unchecked. Here, for sake of simplicity, we only consider the unchecked ones.

[2] A METAFJIG$_1$ *meta-program* is an arbitrary sequence of class declarations, whereas a *program* is a sequence where all right-hand sides are basic classes.

```
}
```

Conditional compilation One very basic use of this mechanism is conditional compilation. For instance:

```
C = {
  class m(){
      if(DEBUG) return /*debug version*/;
      else return /*release version*/;
  }
}
```

Mixins Another simple example is the following method:

```
class mixinFoo(class parent){
  return { /* ... */ } [+] parent;
}
```

which behaves like a *mixin* class, extending in some way a parent class passed as argument.

Note that, instead of sum, we should more appropriately use the derived **`override`** operator introduced in Section 3.1 at page 35. In the sequel we will show how this and other derived operators, which in FJIG$_1$ are language extensions which can be encoded by a type-driven translation, can be directly defined by the programmer in METAFJIG$_1$.

The class to be used as parent could be constructed, having a class `ClassList` implementing lists of classes, by chaining an arbitrary number of classes:

```
class chain(ClassList parents){
  if(parents.isEmpty()) return {};
  else return parents.head() [+] this.chain(parents.tail());
}
```

This is indeed similar to mixin composition, with the advantage that the operands of this arbitrarily long chain do not have to be statically known.

Derived operators We show now how to derive other useful composition operators from the primitive ones. To this end, we use an additional primitive operator `[names]` which returns the array of member names of a class, omitted in the formalization in Section 4.3 since it poses no significant new technical problems. This operator allows to emulate a type-driven translation. Moreover, we use the following class

```
IsDefined = {
  boolean apply(class c, name n){
    try{
      c=c[restrict n];
      return true;
    }
    catch(CompositionException e){
      return false;
    }
  }
}
```

which defines an operator which checks whether a member is defined (that is, declared and non abstract) in a class. This operator can be derived from the restrict operator, which throws an exception when invoked on a non defined member. The predefined `CompositionException` should be a supertype of all the exceptions modelling composition errors.

In this and the following examples some choices are due to the fact that we want to stick to a simple model. For instance, we use arrays rather than lists to avoid assuming predefined classes except `Object`, `RunTimeException` and composition operator exceptions. Moreover, operators should better be implemented as static methods, but adding such a concept is future work, hence we write a class for each operator, with an `apply` method that represents the operator application.

Override The following class defines the override operator.

```
Override = {
  class apply(class h,class p){
    for(name n:h[names])
      if(new IsDefined().apply(h,n) && new IsDefined().apply(p,n))
        p=p[restrict n];
    try{
      return h [+] p;
    }
    catch(CompositionException se){
      throw new OverrideException("...");
    }
  }
}
OverrideException= implements CompositionException{...}
```

For all members which are defined both in heir `h` and in parent `p`, the definition in `p` has to be removed. An exception can still be raised if some member is declared in both with different

types, see rule (SUM-ERROR) in Figure 4.2.

Rename The following class defines a variant of the rename operator introduced in Section 3.1 on page 39.

```
Rename = {
  class apply(class c, name o, name n){
    if(!isIn(o,c[members]))
      throw new RenameException("...");
    if(o==n)return c;
    if(isIn(n,c[members]))
      throw new RenameException("...");
    return c[alias o to n][redirect o to n];
  }
  boolean isIn(name n,name[] ns){...}
}
```

To better illustrate the expressive power offered by the meta-level, this variant of the operator performs additional checks w.r.t. the version in Section 3.1: notably, the new name `n` must be not declared, and it is allowed to rename a name into itself.

Strong alias and unbind The following classes define two less standard operators.

```
StrongAlias = {
  class apply(class c, name o, name n){
    try{
     return c[alias o to n][redirect o to n][ alias n to o];
    }
    catch(CompositionException ce){
      throw new StrongAliasException("...");
     }
  }
}
Unbind = {
  class apply(class c, name o, name n){
    try{
      return new StrongAlias().apply(c,o,n)[restrict n];
    }
    catch(CompositionException ce){
      throw new UnbindException("...");
      }
    }
}
```

The former is a variant of alias which also replaces **this** references. A similar operator is defined in [LS08b], but only affects **this** references appearing in the body of the method itself. This operator works in three steps: duplicates the definition of `o` as `n`, redirects the name `o` to `n`, and finally restores the original definition of `o`.

The latter replaces **this** references to a defined member `o` by a new name `n`. This effect can be obtained by first applying `StrongAlias`, then removing the definition of `n`. This operator has been introduced in [ALZ06] to deal with unanticipated code modification due to poor design. The converse operator, which binds **this** references to an existing defined member, is a special case of redirect.

4.2 Expressive power

The following example is a graphical library that adapts itself with respect to its execution environment, without requiring any extra-linguistic mechanisms. To keep the example compact, we do not detail all the used classes, and we simply assume that they are declared elsewhere.

```
GraphicalLibrary = {
  class produceLibrary(){
    class result = BaseGraphicalLibrary;
    String producer =
        new System().getProperty("sys.vcard.brand");
    if (producer.equals("NVIDIA"))
      result = NVIDIASupport [+] result;
    else if (producer.equals("ATI"))
      result = ATISupport [+] result;
    else throw
        new UnsupportedHardwareException(
          "No compatible hardware found");
    if (System.getProperty("os.name").contains("Windows"))
      result = new CygwinAdapter().adapt(result);
    return result;
  }
}
```

The method `produceLibrary` builds a platform-specific library by combining the base library `BaseGraphicalLibrary` with the brand-specific drivers (represented by the two classes `NVIDIASupport` and `ATISupport`) and wrapping the result, if required on the specific platform, with the class `CygwinAdapter`, which emulates a Linux-like environment on Windows operating systems.

In this way the compilation of the same source produces customized versions of the library depending on the execution platform. In other words, this approach can be used to write *active libraries* [CEG+00], that is, libraries that interact dynamically with the compiler, providing better services, as meaningful error messages, platform-specific optimizations and so on.

Another simple and interesting application is in managing at compile-time external applications like databases. For instance, the following class `DBRecord` provides the method `create` that, given a table name, produces a class which mimics the structure of the table.

```
DBRecord = {
  StringClassMap map=...
//{"int" ->  {int n;constructor(){this.n=0;}},
//"String"-> {String n;constructor(){this.n="";}},
//  ...};
  class create(String table){
    TableStructure structure=
        new DB().getTableStructure(table);
    class result={};
    for(Column c:structure.getColumns())
      result=result [+] new Rename().apply(
          map.get(c.myType),n,c.myName);
    return result;
    }
  }
```

The `create` method takes the name of a table, and for all the `Column c` of such table adds to the class stored in `result` a field with the type and name of `c`.

Many applications rely on the fact that the shape of some database table is known and immutable, but with conventional approaches the programmer needs to duplicate the structure of the external table in the code, and checks on the table shape are performed only at runtime.

With our approach, instead, a class defined by

```
Foo = new DBRecord().create("Foo")
```

will have the shape of the table `"Foo"` in the `DB`. That is, the following code:

```
int m(Foo x){return x.bar;}
```

will be successfully compiled only if the table `"Foo"` has a field `bar` of type **`int`**. That is, checks on the table shape are performed at compile-time.[3]

[3]However, since the table shape can change after compilation, the code that reads the table should perform integrity checks at runtime.

This approach relieves the programmer of a big burden: ensuring the consistency between the DB and the source code. This technique shifts this responsibility leveraging the type system of the conventional language.

The two examples above also show that our approach allows, in a statically typed setting, an expressive power which is typical of dynamically typed languages.

4.3 Syntax and reduction rules

In Figure 4.1 we give the syntax of METAFJIG$_1$.

A meta-program is a sequence of class declarations, where an expression is associated to a class name. Moreover, a meta program can reduce to a composition error.

A program is a meta-program where all these expressions are basic classes, that are as in FJIG$_0$.

In a well-formed (meta-)program, a class name cannot be declared twice, that is, a (meta-)program is a map from class names to expressions.

Expressions include those of FJIG$_0$ and are extended with *meta-expressions*. Meta-expressions denoting classes are constants (basic classes), class names, or are constructed by the four operators sum, restrict, alias, and redirect, with the same behaviour of FJIG$_1$. Meta-expressions denoting (member) names are just constants.

Types are class names and the primitive types **`class`** of classes and **`name`** of names.

In Figure 4.2 we give the (new) rules which define the reduction of an expression e in the context of a program p. The other rules, modelling conventional constructs, are exactly as in FJIG$_0$ and omitted here.

All the notations are the same as FJIG$_0$ and FJIG$_1$. Rule (CLASS-NAME) is formally the same of the corresponding flattening rule in FJIG$_1$, while for each composition operator there are two rules.

The former reduces the term analogously to the corresponding reduction rule in FJIG$_1$, and the side conditions are analogous to the ones of the corresponding typing rule in FJIG$_1$. The latter corresponds to the case when the operator cannot be performed, hence a composition error is raised. Indeed, in METAFJIG$_1$ all meta-expressions denoting classes have a unique primitive type **`class`**, rather than a class type as the corresponding class expressions in FJIG$_1$, and checks on the applicability of operators are runtime checks rather than static checks as in FJIG$_1$. However, note that the necessary type information can be extracted from the basic classes which are arguments of the operator, and, in the case of the redirect operator, from the program.

We denote by Δ^p the program type extracted from p, and by ct^b the class type extracted from b,

mp	::=	$\overline{c = e}$ | **errorC**	meta-program
p	::=	$\overline{c = b}$	program
b	::=	$ch\ \{\ k\ \overline{d}\ \}$	basic class
ch	::=	μ **implements** $\overline{c}$	class header
μ	::=	ϵ | **abstract**	abstract modifier
k	::=	$kh\{\overline{fe}\}$	constructor
kh	::=	**constructor** $(\ \overline{T\ x}\ )$	constructor header
fe	::=	**this.**f = e**;**	field expression
d	::=	fd | md	declaration
f, m	::=	n	
fd	::=	$\mu\ T\ f$**;**	field declaration
md	::=	**abstract** mh; | $mh\{$ **return** e**;** $\}$	method declaration
mh	::=	$T\ m(\overline{T\ x})$	method header
e	::=		expression
		x | $e.f$ | $e.m\ (\overline{e})$ | **new** $c\ (\overline{e})$	conventional
		| $c\ (\overline{fe})$	pre-object
		| b	basic class
		| c	class name
		| $e_1\ [+]\ e_2$	sum
		| $e[$**restrict** $e']$	restrict
		| $e[$**alias** e^s **to** $e^t]$	alias
		| $e[$**redirect** e^s **to** $e^t]$	redirect
		| n	(member) name
		| **errorC**	composition error
T	::=	c | **class** | **name**	type
v	::=	$c\ (\overline{fv})$ | b | n	value
fv	::=	**this.**f = v**;**	field value

Figure 4.1: METAFJIG$_1$ syntax

$$\mathcal{E}^{\mathrm{r}} \;::=\; \Box \mid \mathcal{E}^{\mathrm{r}}.f \mid \mathcal{E}^{\mathrm{r}}.m(\overline{e}) \mid v.m(\overline{v}, \mathcal{E}^{\mathrm{r}}, \overline{e}) \mid \mathtt{new}\ C\ (\overline{v}, \mathcal{E}^{\mathrm{r}}, \overline{e})$$
$$\mid \mathcal{E}^{\mathrm{r}}\ [+]\ e \mid v\ [+]\ \mathcal{E}^{\mathrm{r}} \mid \mathcal{E}^{\mathrm{r}}[\mathtt{restrict}\ e] \mid v[\mathtt{restrict}\ \mathcal{E}^{\mathrm{r}}] \mid \ldots$$

$e_1 \underset{p}{\rightarrow} e_2$

(CTX-ERROR) $\dfrac{}{\mathcal{E}^{\mathrm{r}}[\![\mathtt{errorC}]\!] \underset{p}{\rightarrow} \mathtt{errorC}}$ (CLASS-NAME) $\dfrac{}{c \underset{p}{\rightarrow} b}\ p(c) = b$

(SUM) $\dfrac{}{b_1\ [+]\ b_2 \underset{p}{\rightarrow} b_1 \oplus b_2}$ (SUM-ERROR) $\dfrac{}{b_1\ [+]\ b_2 \underset{p}{\rightarrow} \mathtt{errorC}}\ ct_1^b \oplus ct_2^b$ undefined

(RESTRICT) $\dfrac{}{b[\mathtt{restrict}\ n] \underset{p}{\rightarrow} b \ominus n \oplus \mathsf{abs}(d)}\ d = \mathsf{dec}(b, n)$

(RESTRICT-ERROR) $\dfrac{}{b[\mathtt{restrict}\ n] \underset{p}{\rightarrow} \mathtt{errorC}}\ \mathsf{mType}(ct^b, n)$ undefined

(ALIAS) $\dfrac{}{b[\mathtt{alias}\ n^s\ \mathtt{to}\ n^t] \underset{p}{\rightarrow} b' \oplus \mathsf{named}(n^t, d)}$ $d = \mathsf{dec}(b, n^s)$, $\mathsf{constr}(b) = kh\,\{\overline{fe}\}$, $b' = \begin{cases} b \oplus \mathtt{this}.n^t = e; & \text{if } \overline{fe}(n^s) = e \\ b & \text{if } n^s \notin \mathsf{dom}(\overline{fe}) \end{cases}$

(ALIAS-ERROR) $\dfrac{}{b[\mathtt{alias}\ n^s\ \mathtt{to}\ n^t] \underset{p}{\rightarrow} \mathtt{errorC}}$ $\mathsf{mType}(ct^b, n^s)$ undefined or $\mathsf{mType}(ct^b, n^s) \neq \mathsf{mType}(ct^b, n^t)$

(REDIRECT) $\dfrac{}{b[\mathtt{redirect}\ n^s\ \mathtt{to}\ n^t] \underset{p}{\rightarrow} (b \ominus n^s)[n^s \rightsquigarrow n^t]}$ $\mathsf{mType}(ct^b, n^s) = \mathsf{mType}(ct^b, n^t)$, $n^s \neq n^t$, $\Delta^p \vdash ct^b$

(REDIRECT-ERROR) $\dfrac{}{b[\mathtt{redirect}\ n^s\ \mathtt{to}\ n^t] \underset{p}{\rightarrow} \mathtt{errorC}}$ $\mathsf{mType}(ct^b, n^s) \neq \mathsf{mType}(ct^b, n^t)$ or $n^s = n^t$ or $\Delta^p \not\vdash ct^b$

Figure 4.2: METAFJIG$_1$ reduction rules

formally defined by:

- $\Delta^{(c_1 = b_1 \ldots c_n = b_n)} = c_1{:}ct^{b_1} \ldots c_n{:}ct^{b_n}$
- $ct^b = [\mu \mid \overline{C} \mid kt^k \mid dt_1^d \ldots dt_n^d]$ if $b = \mu\ \texttt{implements}\ \overline{C}\ \{k\ d_1 \ldots d_n\}$
  $dt^{\mu\ T\ f\texttt{;}} = f{:}\mu\ T$
  $dt^{\texttt{abstract}\ T\ m(T_1\ \_,\ldots,T_n\ \_)\texttt{;}} = m{:}\texttt{abstract}\ T_1 \ldots T_n \to T$
  $dt^{T\ m(T_1\ \_,\ldots,T_n\ \_)\{\texttt{return}\ \_\texttt{;}\}} = m{:}\epsilon\ T_1 \ldots T_n \to T$
  $kt^{\texttt{constructor}\ (T_1\ \_,\ldots,T_n\ \_)\ \{\_\}} = T_1 \ldots T_n$

4.4 Type system

A METAFJIG$_1$ meta-program is *not* statically typechecked, but rather *incrementally* typechecked, as we will detail in Section 4.5. That is, only the portions of the meta-program which are (closed) programs can be statically typechecked. Correspondingly, the METAFJIG$_1$ type system does not include a typing judgment for meta-programs, but only for programs.

This typing judgment is defined analogously to the one of FJIG$_0$. More precisely, type environments and subtyping rules are exactly as in FJIG$_0$ ad FJIG$_1$, hence not reported here. Typing rules for programs and classes are reported in Figure 4.3.

We report all the rules for clarity. However, they are exactly as in FJIG$_0$ and FJIG$_1$, apart that the typing judgement $\Delta; \Gamma \vdash e : T^\Lambda$ for expressions, used in premises of rules (CONS-T) and (METHOD-T), has been replaced by the judgement $\Delta; \Gamma \vdash^\star e : T^\Lambda$. Indeed, we distinguish two typing judgments for METAFJIG$_1$ (meta-)expressions: *strong well-typedness*, denoted by $\vdash^\star$, and *(weak) well-typedness*, denoted by $\vdash^o$.

The former judgement is used when typechecking (expressions occurring in) programs, and requires, in order for a constant meta-expression b to be well-typed (of type **class**), b to be well-typed as basic class. This implies, notably, that all expressions (recursively) occurring in b must be well-typed as well, as suggested by the notation $\vdash^\star$. The latter judgement, instead, is used when typechecking (expressions occurring in) meta-expressions defining class names, and requires, in order a constant meta-expression b to be well-typed (of type **class**), only a weak requirement, detailed below, ensuring that reduction does not get stuck. This implies, notably, that no check is performed on expressions occurring in b, as suggested by the notation $\vdash^o$.

The difference is illustrated by the following example.

```
A = { int foo(){return 0;} }
B = class{ int n(){return new A().bar();} } [+] {}
C = { class k(){return class{ int n(){return new A().bar();} }; }
```

$\vdash p : pt$

$$\text{(PROGRAM-T)}\ \frac{pt \vdash ce_i : ct_i \quad \forall i{\in}1..n}{\vdash p : pt}\ \begin{array}{l} p = c_1 = b_1 \ldots c_n = b_n \\ pt = c_1{:}ct_1 \ldots c_n{:}ct_n \end{array}$$

$pt \vdash b : ct$

$$\text{(BASIC-T)}\ \frac{ct; pt \vdash k : kt \quad ct; pt \vdash \overline{d} : \overline{dt} \quad ct; pt \vdash ct}{pt \vdash \mu\ \texttt{implements}\ \overline{c}\ \{\, k\ \overline{d}\,\} : ct}\ ct = [\mu \mid \overline{c} \mid kt \mid \overline{dt}]$$

$\Delta \vdash k : kt$

$$\text{(CONS-T)}\ \frac{\begin{array}{c} pt; x_1{:}T'_1, \ldots, x_n{:}T'_n \vdash^{\star} e_i : T''_i \quad \forall i \in 1..k \\ \Delta \vdash T''_i {\leq} T_i \quad \forall i \in 1..k \end{array}}{\Delta \vdash kh\,\{\overline{fe}\,\} : T'_1 \ldots T'_n}\ \begin{array}{l} kh = \texttt{constructor}\,(\,T'_1\ x_1, \ldots, T'_n\ x_n) \\ \overline{fe} = \texttt{this.}f_1 = e_1\texttt{;}\ \ldots \texttt{this.}f_k = e_k\texttt{;} \\ \Delta = ct; pt \\ \mathsf{exists}(\Delta, T'_i) \quad \forall i \in 1..n \\ \mathsf{defFields}(\Delta, \Lambda) = f_1{:}\epsilon\ T_1, \ldots, f_k{:}\epsilon\ T_k \end{array}$$

$\Delta \vdash d : dt$

$$\text{(FIELD-T)}\ \frac{}{\Delta \vdash (\mu\ T\ f\texttt{;}) : (f{:}\mu\ T)}\ \mathsf{exists}(\Delta, T)$$

$$\text{(ABS-METHOD-T)}\ \frac{}{\Delta \vdash (\texttt{abstract}\ mh\texttt{;}) : (m{:}\texttt{abstract}\ T_1 \ldots T_n {\rightarrow} T_0)}\ \begin{array}{l} mh = T_0\, m(\,T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n \end{array}$$

$$\text{(METHOD-T)}\ \frac{\Delta; \texttt{this}{:}\Lambda, x_1{:}T_1, \ldots, x_n{:}T_n \vdash^{\star} e : T^{\Lambda} \quad \Delta \vdash T^{\Lambda} {\leq} T_0}{\Delta \vdash mh\,\{\texttt{return}\ e\texttt{;}\} : (m{:}\epsilon\ T_1 \ldots T_n {\rightarrow} T_0)}\ \begin{array}{l} mh = T_0\, m(\,T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n \end{array}$$

Figure 4.3: METAFJIG$_1$ typing rules for programs and classes

The marked basic class, occurring two times in the above code, is clearly ill-typed as class, since there is no method `bar` in class `A`. Hence (as meta-expression) it is weakly, but not strongly, well-typed.

As a consequence, the meta-expression defining `B` is weakly well-typed as well, hence can be reduced, and, actually, its reduction can safely happen. This weaker requirement allows compilation of class declarations where type annotations are mutually recursive, see the end of Section 4.5.

On the other side, the program composed by classes `A` and `C` is ill-typed, since method `C.k` contains a basic class that is not strongly well-typed. Requiring strong well-typedness of expressions occurring in programs, that is, that all basic classes occurring in a program at any meta-level must be well-typed as classes, is necessary to ensure *meta-level soundness*, an additional important property discussed later on, see Theorem 18.

Rules in Figure 4.4 define typing of expressions. As informally explained above, the only difference in the definition of the two judgements is in the rule for typing basic classes.

Rule (BASIC-META-T$\star$) states that a basic class is a deeply well-typed meta-expression only if it is well-typed as class.

On the other hand, in order to be a weakly well-typed class meta-expression, the only condition a basic class must satisfy is that its class type must be well-formed, as shown in rule (BASIC-META-T0). Recall that this means that the basic class must declare all the members needed to be a (structural) subtype of the declared supertypes,[4]see rule (WF-CLASS-TYPE) in Figure 2.4.

In the other rules given in Figure 4.4, L ranges over 0 and $\star$. Such rules are analogous to the ones of FJIG$_0$ or straightforward.

4.5 Checked compile-time execution

We consider now *meta-programs*, that is, sequences of class declarations where arbitrary expressions, rather than basic classes, are associated to class names. A meta-program can be reduced to a program by a process that we call *compile-time execution*, formally modelled by the relation $\Rightarrow$ defined in Figure 4.5. As modelled by (UNCHECKED-META-RED), the right-hand side of a class declaration can be reduced in the context of the program part of the current meta-program.[5]

[4]Technically, the only condition which is strictly needed to ensure that reduction does not get stuck is that, when applying a redirect operator, we *can* actually check whether we would get a well-formed class type, see rule (REDIRECT) and (REDIRECT-ERROR). This check is possible only if the definitions of all the declared supertypes have already been reduced to basic classes. However, we preferred to stick to the more uniform requirement that class types of basic classes are *always* well-formed.

[5]In the premise we use the transitive closure of the reduction relation to include different strategies. A modular implementation constructed on top of the JVM, as our prototype, will not stop JVM execution once started. Formally,

$\Delta; \Gamma \vdash^{\iota} e : T^{\Lambda}$

$$\text{(VAR-T)}\ \frac{}{\Delta; \Gamma \vdash^{\iota} x : T^{\Lambda}}\ \Gamma(x) = T^{\Lambda} \qquad \text{(FIELD-ACCESS-T)}\ \frac{\Delta; \Gamma \vdash^{\iota} e : c^{\Lambda}}{\Delta; \Gamma \vdash^{\iota} e.f : T}\ \mathsf{mType}(\Delta, c^{\Lambda}, f) = T$$

$$\text{(INVK-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash^{\iota} e : c^{\Lambda} \\ \Delta; \Gamma \vdash^{\iota} \overline{e} : \overline{T^{\Lambda}} \\ \Delta \vdash \overline{T^{\Lambda}} \leq \overline{T}\end{array}}{\Delta; \Gamma \vdash^{\iota} e.m\,(\overline{e})\ : T}\ \mathsf{mType}(\Delta, c^{\Lambda}, m) = \overline{T} \to T$$

$$\text{(NEW-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash^{\iota} \overline{e} : \overline{T^{\Lambda}} \\ \Delta \vdash \overline{T^{\Lambda}} \leq \overline{T}\end{array}}{\Delta; \Gamma \vdash^{\iota} \mathbf{new}\ c\,(\overline{e})\ : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, c) \\ \mathsf{kType}(\Delta, c) = \overline{T}\end{array}$$

$$\text{(OBJ-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash^{\iota} e_i : T_i^{\Lambda} \quad \forall i \in 1..n \\ \Delta \vdash T_i^{\Lambda} \leq T_i \quad \forall i \in 1..n\end{array}}{\Delta; \Gamma \vdash^{\iota} c\,(\overline{fe})\ : c}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, C) \\ \overline{fe} = \mathbf{this}.f_1 = e_1; \ldots \mathbf{this}.f_n = e_n; \\ \mathsf{defFields}(\Delta, C) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}$$

$$\text{(BASIC-META-T}\star\text{)}\ \frac{\Delta \vdash b : ct}{\Delta; \Gamma \vdash^{\star} b : \mathbf{class}} \qquad \text{(BASIC-META-T0)}\ \frac{}{\Delta; \Gamma \vdash^{0} b : \mathbf{class}}\ \Delta \vdash ct^{b}$$

$$\text{(SUM-T)}\ \frac{\Delta; \Gamma \vdash^{\iota} e_1 : \mathbf{class} \quad \Delta; \Gamma \vdash^{\iota} e_2 : \mathbf{class}}{\Delta; \Gamma \vdash^{\iota} e_1\,[+]\,e_2 : \mathbf{class}}$$

$$\text{(RESTRICT-T)}\ \frac{\Delta; \Gamma \vdash^{\iota} e : \mathbf{class} \quad \Delta; \Gamma \vdash^{\iota} e' : \mathbf{name}}{\Delta; \Gamma \vdash^{\iota} e[\mathbf{restrict}\ e'] : \mathbf{class}}$$

$$\text{(ALIAS-T)}\ \frac{\Delta; \Gamma \vdash^{\iota} e : \mathbf{class} \quad \Delta; \Gamma \vdash^{\iota} e^{s} : \mathbf{name} \quad \Delta; \Gamma \vdash^{\iota} e^{t} : \mathbf{name}}{\Delta; \Gamma \vdash^{\iota} e[\mathbf{alias}\ e^{s}\ \mathbf{to}\ e^{t}] : \mathbf{class}}$$

$$\text{(REDIRECT-T)}\ \frac{\Delta; \Gamma \vdash^{\iota} e : \mathbf{class} \quad \Delta; \Gamma \vdash^{\iota} e^{s} : \mathbf{name} \quad \Delta; \Gamma \vdash^{\iota} e^{t} : \mathbf{name}}{\Delta; \Gamma \vdash^{\iota} e[\mathbf{redirect}\ e^{s}\ \mathbf{to}\ e^{t}] : \mathbf{class}}$$

$$\text{(NAME-T)}\ \frac{}{\Delta; \Gamma \vdash^{\iota} n : \mathbf{name}}$$

Figure 4.4: METAFJIG$_1$ typing rules for expressions

$$mp_1 \Rightarrow mp_2$$

$$\text{(UNCHECKED-META-RED)}\ \frac{e_1 \xrightarrow[p]{\dagger} e_2}{mp\ p\ (c = e_1) \Rightarrow mp\ p\ (c = e_2)}$$

Figure 4.5: METAFJIG$_1$ compile-time execution

However, soundness of compile-time execution is not guaranteed, that is, reduction could get stuck, or produce as right-hand side of a class declaration a value different from a basic class, or an ill-typed basic class.

To prevent these error situations we propose a simple solution, called *checked compile-time execution*, which integrates meta-reduction with incremental typechecking. If typechecking fails, then the program reduces to **`errorT`**. The advantage is that the technique can be modularly defined on top of an arbitrary Java-like language.

Rules defining checked compile-time execution are given in Figure 4.6. We assume that the whole (meta-)program is *closed*, that is, only class names which are declared can appear in expressions. However, meta-variables mp and p can denote open sections of the program, for example in rule (UNCHECKED-META-RED) mp can refer to classes declared in p.

The first rule, (META-RED), models a meta-reduction step. Note that now meta-reduction is *checked*, that is, the rule can be applied only if p is well-typed and e_1 is a weakly well-typed meta-expression of type **`class`** w.r.t. the type of p. This guarantees that (meta-)reduction does not get stuck, see Theorem 14.

The other rules correspond to different cases of abnormal termination.

In rule (META-RED-ERROR), meta-reduction raises a composition error, that is, the applicability conditions for some composition operator do not hold.

In rules (META-CHECK-ERROR), a typechecking error is raised, since a right-hand side e of a class declaration is not weakly well-typed of type **`class`**. Analogously, in rule (CHECK-ERROR), a typechecking error is raised, since a program portion p is ill-typed. In both cases, the error should be raised only if the type information needed to typecheck e and p, respectively, is available. This is captured by the side conditions $\mathsf{closed}(p, e)$ and $\mathsf{closed}(p)$, respectively, explained below. For example in the following program

```
A = { int one(){return 1;} } [+] {}
```

this corresponds to take the maximal length of $\xrightarrow[p]{\dagger}$.

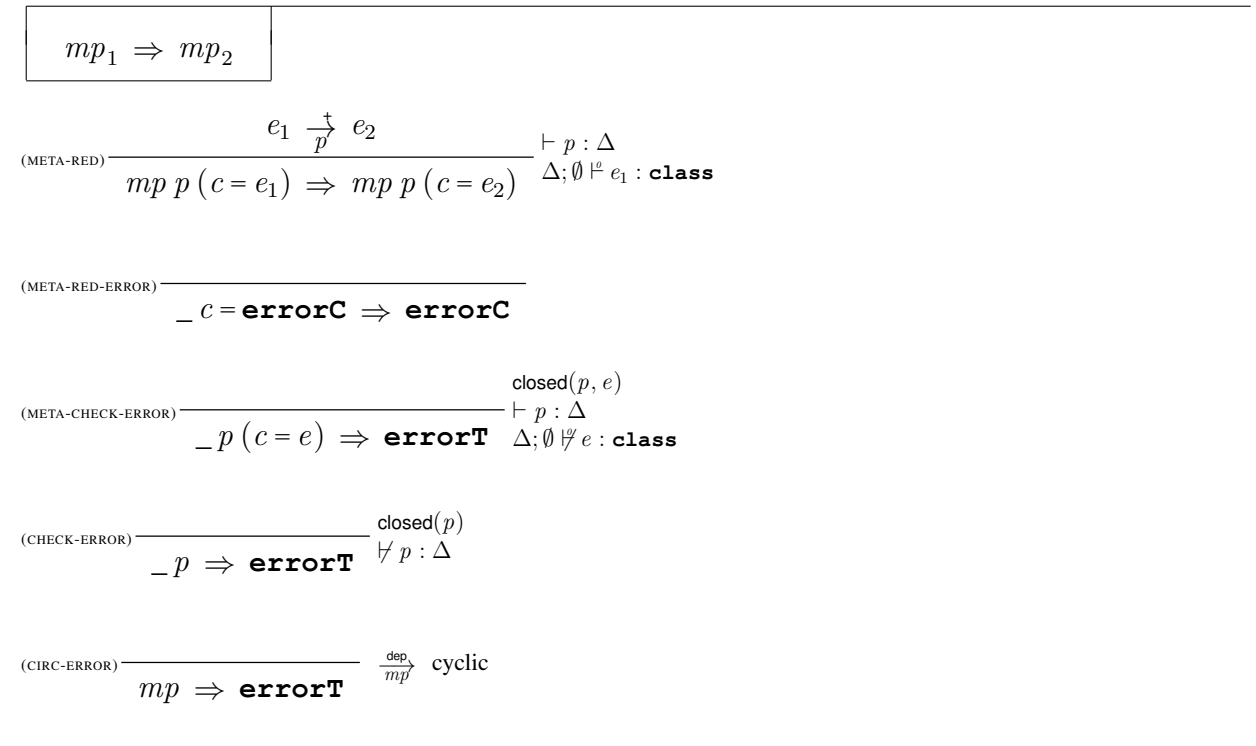

Figure 4.6: METAFJIG$_1$ checked compile-time execution

```
B = { int m(){return 1 + true;} }
C = { int m(A x){return 1 + true;} }
D = { int m(A x){return 1 + x.one();} }
```

class `B` and `C` are obviously ill-typed. However, there are not yet enough type information to typecheck classes `C` and `D`. Class `D` is intuitivelly well-typed, but without the type information of `A`, the type system would declare class `D` ill-typed.

Finally, in (CIRC-ERROR), a *circularity error* is raised. This happens when a cyclic dependency among classes is detected, where a class c *depends on* c' in mp if, in order to typecheck the meta-expression defining c, we need the type information about c' which, hence, should be reduced to a basic class *before* typechecking c. Indeed, in case of cyclic dependency there is no hope to be able to reduce the involved classes in the future. A very simple example of cyclic dependency is `C = new C().m()`.

In the rules we use the following notations:

- $\mathsf{closed}(p, e)$ holds if all (clients of) class names directly occurring in e are declared in p, formally:
  $\mathsf{closed}(p, e)$ iff, for all $c \in \mathsf{cnames}^0(e)$, $\{c' \mid c \xrightarrow[p]{\mathsf{client}} c'\} \subseteq \mathsf{dom}(p)$, where:
  - $\mathsf{cnames}^0(e)$ are the class names directly occurring in e,
  - $c \xrightarrow[p]{\mathsf{client}} c'$ holds if c is a *client* of c' in p, that is, either $c' = c$, or, transitively, (a client of) c' occurs in the basic class defining c in p,
  - $\mathsf{cnames}^\star(b)$ are the class names occurring in a basic class b.
- $\mathsf{closed}(p)$ holds if clients of class names declared in p are declared in p as well, formally:
  $\mathsf{closed}(p)$ iff, for all $c \in \mathsf{dom}(p)$, $\{c' \mid c \xrightarrow[p]{\mathsf{client}} c'\} \subseteq \mathsf{dom}(p)$.
- $c \xrightarrow[mp]{\mathsf{dep}} c'$ holds if (a client of) either c' or (transitively) a class which depends on c' directly occurs in the meta-expression e defining c in mp.

Figure 4.7 contains the formal definition of clientship and dependency relation and auxiliary functions $\mathsf{cnames}^0(e)$ and $\mathsf{cnames}^\star(b)$.

Clientship is analogous to the same notion in Java. In Java, this relation is indeed used when the compiler is invoked to determine the full set of classes to be compiled or present in bytecode form. We assume that any class is a client of itself for technical convenience. Note that, in the definition of $\mathsf{cnames}^0(e)$, occurrences of class names in basic classes in e are not taken into account. For instance, if e is

```
new C().m({ D n() { return new D(); } })
```

$c_1 \xrightarrow[mp]{\text{client}} c_2$

$$\text{(DIRECT-C)}\ \frac{p(c_1) = b}{c_1 \xrightarrow[p]{\text{client}} c_2}\ c_2 \in \mathsf{cnames}^\star(b) \qquad \text{(REFL-C)}\ \frac{}{c \xrightarrow[p]{\text{client}} c} \qquad \text{(TRANS-C)}\ \frac{c_1 \xrightarrow[p]{\text{client}} c_2 \quad c_2 \xrightarrow[p]{\text{client}} c_3}{c_1 \xrightarrow[p]{\text{client}} c_3}$$

$\mathsf{cnames}^\star(\mu\ \mathbf{implements}\ \overline{c}\ \{\, k\ \overline{d}\,\}) = \overline{c} \cup \mathsf{cnames}^\star(k) \cup \mathsf{cnames}^\star(\overline{d})$
$\mathsf{cnames}^\star(\mathbf{constructor}\,(\,T_1\ x_1 \ldots T_n\ x_n\,)\ \{\overline{fe}\,\}) = \mathsf{cnames}^\star(\,T_1 \ldots T_n) \cup \mathsf{cnames}^\star(\overline{fe})$
$\mathsf{cnames}^\star(\mathbf{this}.f = e\,\mathbf{;}) = \mathsf{cnames}^\star(e)$
$\mathsf{cnames}^\star(\mu\ c\,f\,\mathbf{;}) = c$
$\mathsf{cnames}^\star(\mathbf{abstract}\ T_0\ m(\,T_1\ x_1 \ldots T_n\ x_n)\mathbf{;}) = \mathsf{cnames}^\star(\,T_0 \ldots T_n)$
$\mathsf{cnames}^\star(\,T_0\ m(\,T_1\ x_1 \ldots T_n\ x_n)\{\,\mathbf{return}\ e\,\mathbf{;}\,\}) = \mathsf{cnames}^\star(\,T_0 \ldots T_n) \cup \mathsf{cnames}^\star(e)$
$\mathsf{cnames}^\star(x) = \emptyset$
$\mathsf{cnames}^\star(e.f) = \mathsf{cnames}^\star(e)$
$\mathsf{cnames}^\star(e.m\,(\,\overline{e}\,)\,) = \mathsf{cnames}^\star(e) \cup \mathsf{cnames}^\star(\overline{e})$
$\mathsf{cnames}^\star(\mathbf{new}\ c\,(\,\overline{e}\,)\,) = c \cup \mathsf{cnames}^\star(\overline{e})$
$\mathsf{cnames}^\star(\,c\,(\,\overline{e}\,)\,) = c \cup \mathsf{cnames}^\star(\overline{e})$
$\mathsf{cnames}^\star(c) = c$
$\mathsf{cnames}^\star(\,T) = \emptyset$ if $T \neq c$
$\mathsf{cnames}^\star(e_1[\mathbf{restrict}\ e_2]) = \mathsf{cnames}^\star(e_1) \cup \mathsf{cnames}^\star(e_2)$
...

$c_1 \xrightarrow[p]{\text{dep}} c_2$

$$\text{(DIRECT-D)}\ \frac{c_2 \xrightarrow[p]{\text{client}} c_3 \quad mp(c_1) = e}{c_1 \xrightarrow[p\ mp]{\text{dep}} c_3 \quad mp(c_3) \neq b}\ c_2 \in \mathsf{cnames}^0(e) \qquad \text{(TRANS-D)}\ \frac{c_1 \xrightarrow[mp]{\text{dep}} c_2 \quad c_2 \xrightarrow[mp]{\text{dep}} c_3}{c_1 \xrightarrow[mp]{\text{dep}} c_3}$$

$\mathsf{cnames}^0(x) = \emptyset$
$\mathsf{cnames}^0(e.f) = \mathsf{cnames}^0(e)$
$\mathsf{cnames}^0(e.m\,(\,\overline{e}\,)\,) = \mathsf{cnames}^0(e) \cup \mathsf{cnames}^0(\overline{e})$
$\mathsf{cnames}^0(\mathbf{new}\ c\,(\,\overline{e}\,)\,) = c \cup \mathsf{cnames}^0(\overline{e})$
$\mathsf{cnames}^0(\,c\,(\,\overline{e}\,)\,) = c \cup \mathsf{cnames}^0(\overline{e})$
$\mathsf{cnames}^0(\mathbf{\_\ implements}\ \overline{c}\ \{\mathbf{\_\_}\}) = \overline{c}$
$\mathsf{cnames}^0(c) = c$
$\mathsf{cnames}^0(e_1[\mathbf{restrict}\ e_2]) = \mathsf{cnames}^0(e_1) \cup \mathsf{cnames}^0(e_2)$
...

Figure 4.7: METAFJIG$_1$ clientship and dependency relations

then C $\in$ cnames$^0(e)$, but D $\notin$ cnames$^0(e)$.

We show now some examples illustrating how checked compile-time execution works.

Example 1 First we give an example of successful reduction. In the meta-program

```
C = { class m(){return { int k(){return 1;} };} }
D = { int m(){return new E().k();} }
E = new C().m()
```

class D cannot be typechecked yet, since it is a client of class E whose definition is not a basic class.

This meta-program reduces by rule (META-RED) to

```
C = { class m(){return { int k(){return 1;} };} }
D = { int m(){return new E().k();} }
E = { int k(){return 1;} }
```

Example 2 The second example shows a case when checked compile-time execution terminates with an **errorT**.

```
C = {}
D = new C().k()
```

Class C is well-typed, and **new** C().k() only depends on C, so **new** C().k() can be typechecked, and turns out to be ill-typed, since class C has no method named k. By (META-CHECK-ERROR) **errorT** is raised.

Example 3 The meta-program

```
C = { class m(class x){return x [+] { int k(){return 1;} };} }
D = new C().m(class{ int h(){return new D().k();} })
```

reduces by rule (META-RED) to

```
C = { class m(class x){return x [+] { int k(){return 1;} };} }
D = {
  int h(){return new D().k();}
  int k(){return 1;}
}
```

This example illustrates why only weak well-typedness is required in rule (META-CHECK). Indeed, the fact that the expression **new** D().k() is well-typed can only be detected when the

result of `{ int h(){return new D().k();} } [+] { int k(){return 1;} }` is associated to `D`. That is, requiring only weak well-typedness allows compilation of class declarations where type annotations are mutually recursive.

4.6 Results

Properties of reduction The type system is sound, as formally stated by the following standard theorem.

Theorem 14 (METAFJIG$_1$ soundness)**.** *If* $\vdash p : \Delta$, $\Delta; \emptyset \vdash^{o} e : T$ *and* $e_1 \xrightarrow[p]{}{}^{\star} e_2$*, then either* e_2 *is a value, or* e_2 *is* **errorC***, or* $e_2 \xrightarrow[p]{} \_$.

As usual, soundness can be derived from progress and subject reduction properties.

Theorem 15 (METAFJIG$_1$ progress)**.** *If* $\vdash p : \Delta$ *and* $\Delta; \emptyset \vdash^{o} e : T$*, then either* e *is a value, or* e *is* **errorC***, or* $e \xrightarrow[p]{} \_$.

Theorem 16 (METAFJIG$_1$ subject reduction)**.** *If* $\vdash p : \Delta$, $\Delta; \emptyset \vdash^{o} e_1 : T_1$ *and* $e_1 \xrightarrow[p]{} e_2$*, then, for some* T_2, $\Delta; \emptyset \vdash^{o} e_2 : T_2$ *and* $\Delta \vdash T_2 \leq T_1$.

The proofs are a straightforward extension of those for FJIG$_0$. Indeed, METAFJIG$_1$ programs can be roughly seen as FJIG$_0$ programs with two primitive types.

Properties of checked compile-time execution We can state two significant properties for METAFJIG$_1$ checked compile-time execution: *soundness* (Theorem 17) and *meta-level soundness* (Theorem 18). The former means that checked-compile time execution never gets stuck, while the latter allows the programmer to safely use compiled libraries. We will discuss its meaning in detail later on.

A *value* w.r.t. checked compile-time execution is a well-typed program p, and an *error* is either **errorC** or **errorT**.

Theorem 17 (Soundness)**.** *If* $mp_1 \stackrel{\star}{\Rightarrow} mp_2$*, then either* mp_2 *is a value or* mp_2 *is an error or* $mp_2 \Rightarrow \_$.

Proof. The proof reduces to verify that there is always a rule applicable to mp if mp is neither a value nor an error.

If mp contains a closed ill-typed program p, then rule (CHECK-ERROR) is applicable.

Otherwise, consider the graph corresponding to the relation $\xrightarrow[mp]{\mathsf{dep}}$. If the graph is cyclic, then rule (CIRC-ERROR) is applicable.

Otherwise, since mp is neither a value, nor an error, nor a closed ill-typed program, it is necessarily of shape $\_\,p\,(c = e)$ with $e \neq b$, p the maximal program portion of mp such that $\vdash p : pt$, and c sink node in the dependency graph.[6] This means that there is no c' such that $c \xrightarrow[mp]{\mathsf{dep}} c'$, hence $\mathsf{closed}(p, e)$ holds, and one of rules (META-RED), (META-RED-ERROR) or (META-CHECK-ERROR) is applicable. □

Besides soundness, our technique ensures an additional important property called *meta-level soundness*, which guarantees that type errors are always due to programmers' code, and not to already compiled code, that is, the code of a library, differently from what happens, e.g., for C++ templates. This property holds thanks to the fact that during compile-time execution we cannot generate arbitrary code, but only compose basic classes which were explicitly written in the library.

In the statement of the theorem, we compile a meta-program mp which extends a well-typed program p, which plays the role of a library, obtained by a previous compilation. Note that since meta-programs are maps we can use the standard notation $p \subseteq mp$.

Theorem 18 (METAFJIG$_1$ meta-level soundness)**.** *If* $\vdash p : \Delta$, $p \subseteq mp$ *and* $mp \stackrel{\star}{\Rightarrow}$ **`errorT`**, *then* $\exists c \in \mathsf{dom}(mp) \backslash \mathsf{dom}(p)$ *such that* $\Delta; \emptyset \not\vdash^{\star} mp(c) :$ **`class`**.

Proof. The thesis follows from the following property:

$$\text{if } \vdash p : \Delta,\ \Delta; \emptyset \vdash^{\star} e : \texttt{class} \text{ and } e \xrightarrow[p]{\star} b, \text{ then , for some } ct,\ \Delta \vdash b : ct,$$

which, by rule (BASIC-META-T$\star$), can be derived from progress (Theorem 20) and subject reduction (Theorem 21) below. □

Meta-level soundness means that, if p is well-typed, then typechecking errors are never fault of p, but always originate from some $\_ = e$ in the meta-program mp, as modelled by the fact that judgement $\Delta; \emptyset \vdash^{\star} e :$ **`class`** does not hold. This can happen for two possible reasons: either e is ill-typed as expression of type **`class`**, or it contains some basic class which is ill-typed w.r.t. the library p.

Consider again Example 3 of page 67. Here, class `C` is part of the library p, and is well-typed. Class `D` is part of mp, but its expression is not strongly well-typed. This means that typechecking errors could still be raised, since expression defining `D` refers to a class which is not in the library (`D` itself). However, the result of compilation of `D` *could* be a well-typed class, as it is in this case indeed: after the (META-RED) step, `D` is a well-typed class.

If `k()` were replaced with, say, `k'()`, then rule (CHECK-ERROR) would be applicable. From now on, compilation of another class declaration, say `E = e`, using `C` and `D` as library will only raise typechecking errors due to `E` itself.

[6] In a directed graph a sink node is a node with out-degree equal to zero.

With a non strongly well-typed[7] library p, e.g.

```
D = { int foo(){return 0;} }
C = {
  class m(){
    return { int n(){return new D().bar();} };
  }
}
```

compiling another class declaration mp, say

```
E = new C().m()
```

could raise typechecking errors which are not fault of `E`.

This property is not granted by other approaches like C++ templates. Indeed, in C++ a template instantiation can raise type errors caused by the code of the template itself.

Lemma 19 states that by (successfully) applying a composition operator to strongly well-typed classes we always get a strongly well-typed class.

Lemma 19. *If* $\vdash p : \Delta$, $\Delta; \emptyset \vdash^{\star} b_1 : \mathbf{class}, \Delta; \emptyset \vdash^{\star} b_2 : \mathbf{class}$ *and*

1. $b_1\,[+]\,b_2 \underset{p}{\rightarrow} b$ *or*
2. $b_1[\mathbf{restrict}\ n] \underset{p}{\rightarrow} b$ *or*
3. $b_1[\mathbf{redirect}\ n_1\ \mathbf{to}\ n_2] \underset{p}{\rightarrow} b$ *or*
4. $b_1[\mathbf{alias}\ n_1\ \mathbf{to}\ n_2] \underset{p}{\rightarrow} b$

then $\Delta; \emptyset \vdash^{\star} b : \mathbf{class}$.

Proof. Analogously to Theorem 13. □

Theorem 20 (MetaFJig$_1$ meta-level progress)**.** *If* $\vdash p : \Delta$ *and* $\Delta; \emptyset \vdash^{\star} e : T$ *then either* e *is a value or* e *is* `errorC` *or* $e \underset{p}{\rightarrow} \_$.

Proof. Since $\Delta; \emptyset \vdash^{\star} e : T$ implies $\Delta; \emptyset \vdash^{0} e : T$, the thesis follows by Theorem 15. □

Theorem 21 (MetaFJig$_1$ meta-level subject reduction)**.** *If* $\vdash p : \Delta$, $\Delta; \emptyset \vdash^{\star} e_1 : T_1$ *and* $e_1 \underset{p}{\rightarrow} e_2$ *then, for some* T_2, $\Delta; \emptyset \vdash^{\star} e_2 : T_2$ *and* $\Delta \vdash T_2 \leq T_1$.

Proof. Analogously to Theorem 16 with application of Lemma 19. □

[7] Which, however, *cannot* be obtained as the result of checked compile-time compilation.

Advantages of Meta-level soundness w.r.t. other approaches Depending on the programming language we use, there is a (smaller or larger) class of programming errors which are prevented statically ("typing errors"), and another class which is detected at runtime ("dynamic errors"). Suppose Adam writes a library `L`, and then Bob uses the precompiled library `L` to produce a program `P`. What Bob expects is that:

- since `L` has been compiled yet, typing errors found compiling `P` are always "fault of `P`": they can be either intrinsic errors in `P`, or ill-formed uses of `L` (for instance, if `L` is a stack implementation providing a method `push(int)`, an invocation with a non-integer argument) Hence, Bob always gets error messages about his own code
- analogously, dynamic errors raised by execution of `P` are always "fault of `P`": they can be either intrinsic errors in `P`, or uses of `L` which violate its "contract" (for instance, an invocation of method pop can raise an exception if the stack is empty) In the latter case, Adam has likely provided significant error messages in the library, (for instance, "Attempting pop on an empty stack"). Note that these messages can be more expressive than predefined typechecking errors, since they refer to the expected semantics of `L`. Moreover, Bob can write a more robust program which takes an alternative action (catching the "empty stack" exception).

By "meta-level soundness" we mean that the same schema, with the same advantages, can be applied to meta-programming, where "execution of program `P`" becomes "compile-time execution of meta-program `P`".

Existing approaches fail in two main categories:

A Errors consisting in generation of ill-typed code are typing errors, that is, are detected "before compile-time execution". Meta-level soundness clearly holds. This is the MetaML [TS00] approach, and is very convenient in a context where the main aim is code optimization. However, in a context where the main aim is expressive power, this approach is too limited (see below).

B Errors consisting in generation of ill-typed code are only detected "a posteriori", by typechecking the whole generated code. Meta-level soundness does not hold, that is, as often happens in C++ template instantiation, Bob can get typing errors which originate from *inside* the template code Adam wrote.

This kind of errors look like the following one:

```
myFile: In member function 'int A<T>::publicLibraryFun(T) [with T = B]':
myFile:21:   instantiated from here
  libraryFile:13: error: no matching function for call to
    'A<B>::PrivateLibraryClass::m(B&)'
  libraryFile:9: note: candidates are:
    int A<T>::PrivateLibraryClass::m(int) [with T = B]
```

Hence, Bob can get error messages which are not very informative. Moreover these errors break modularity, since they expose implementation details of the library, and are "fatal", in the sense that there is nothing Bob can do in case such errors occur.

We proposes an intermediate approach between A and B, which keeps expressiveness and safety and, in our opinion, provides better error messages than both. That is, in METAFJIG the design choice is to model composition errors as dynamic rather than typing errors. In other words, composition errors are part of the "contract" of a library.

The approach is much more expressive than A, which does not allow to generate code whose shape depends on the input, hence cuts most of the interesting examples shown in the thesis. Moreover, when writing his (meta-)library, Adam can write ad-hoc error messages for composition errors, and they can be handled by Bob's code.

On the other hand, with the strong type system, it is possible to guarantee meta-level soundness, hence to avoid all drawbacks of approach B. Notably, if `P` is ill-typed, then Bob gets a typing error which always refer to his code. On the other side, if `P` requires the execution of composition operators with invalid arguments, then a composition error is raised. This can be used to detect the shape of a class, see, e.g., first example of page 50. Note also that in the thesis we assumed composition exceptions to be unchecked for simplicity, but in a realistic language they could be checked exceptions.

A composition exception which can be thrown by a library and is not correctly described in the documentation is a programming or documentation bug inside the library, exactly as for any other exception.

Chapter 5

Integrating composition and nesting

If we want things to stay as they are, things will have to change.
(Giuseppe Tomasi di Lampedusa - The Leopard)

In this chapter, we describe an extension of FJIG$_1$, called FJIG$_\star$, where composition operators have been generalized to manipulate nested classes. For instance, sum of two classes is hierarchical, in the sense that nested classes with the same name are recursively summed, similarly to *deep mixin composition* [OZ05, Hut06] and *family polymorphism* [Ern01, ISV05, IV07, ISV08], which, however, take an asymmetric approach. Analogously, it is possible to rename or make an alias of a field, method, or a nested class itself, at any depth level.

As in previously presented languages, typing is nominal, as characteristic of Java-like languages. Types are class paths, which are sequences of the form $\texttt{outer}^n.c_1.\ \ldots\ .c_k$ which, depending on the class (node) where they occur, denote another node in the nesting tree. However, class paths denoting the same class are *not* necessarily equivalent, since they can behave differently w.r.t. composition operators.

The resulting language offers a great expressive power, allowing, e.g., to solve the expression problem and to encode generics [BOSW98, GJSB05] and `MyType` [BOW98]. Moreover, since a whole application can be "packed" into a single class, also the basic AOP mechanisms can be expressed. Finally, the kind of code manipulation achieved by the composition operators corresponds to bring some refactoring techniques at the linguistic level.

On the other hand, the generalization of the composition operators to the case with nesting is very natural and intuitive[1], and, more generally, the language keeps a very simple semantics and type system which represent a natural extension for, say, a Java programmer.

We first illustrate FJIG$_\star$, describing how to declare and refer to nested classes and the semantics of deep composition operators in Section 5.1. In Section 5.2 we provide some more interesting

[1] From the point of view of the programmer: as we will see, the formalization is sometimes tricky.

examples showing the expressive power of the language. In Section 5.3 we provide the formal syntax and semantics, in Section 5.4 the type system and in Section 5.5 the soundness results.

5.1 Examples

In FJIG$_\star$, a basic class can also contain declarations of *nested classes*, as shown in the example below.

```
{
  A =
    {
      B = {
        C = {
          <> m() { return new <>(); }
        }
        D = C [+] outer.outer.G
      }
      E = B.C [+] outer.G
      F = { ... }
    } [+]
    { ... }
  G = { ... }
  H = A.B.C [+] G
}
```

Hence, a basic class has a tree shape, where the basic class is the root, the children of a basic class are its nested classes, and the children of a nested class are the basic classes appearing in its defining expression. For instance, the basic class in the example has three children, `A`, `G`, and `H`, and nested class `A` has two children.

Sequences of the form **outer**^{n}.c_1. ... .c_k, with $n, k \geq 0$, called *class paths*, denote a class, and can be used the same way as class names, that is, as types, in **new** expressions and as subterms of class expressions, as shown in the example. Class paths can be classified along two orthogonal dimensions:

- class paths with $k = 0$, called *outer class paths*, denote enclosing basic classes, whereas class paths with $k > 0$ denote nested classes. In particular, the class path Λ (written <> in code), that is, the unique class path where $n, k = 0$, denotes the directly enclosing basic class. As the reader may have noted, this class path has many analogies with the **MyType** notion in literature. However, we prefer the new notation <> to stress that it is just a special case of class path and that no sophisticated notion is needed in the type system, see more

comments in Section 5.2. For instance, in the basic class above the occurrences of <> denote the basic class appearing as the definition of nested class A.B.C.

- class paths with $n = 0$ refer to a class in the current scope, whereas class paths with $n > 0$ refer to outer levels. For instance, in the basic class above the occurrence of **outer.outer.**G denotes the basic class appearing as definition of nested class G.

Of course, in a top-level class expression[2], all class paths are expected to denote existing classes, as in the example above.

The following example illustrates the difference between outer class paths and others.

```
C = { int m(){ return 1; } }
  [+] { int k(){ return new outer.C().m(); } }
```

Here, class C is defined as the sum of two basic classes. In the latter, the method invocation is well-typed, since **outer.**C denotes class C of the outer level, which has a method m provided by the former basic class. The version below, instead,

```
C = { int m(){ return 1; } }
  [+] { int k(){ return new <>().m(); } }
```

is ill-typed, since the basic class enclosing the invocation does *not* provide a method m.

When code is copied from one position to another, there is an important difference between *internal* and *external* class paths, illustrated by the following example. The program

```
A = {
  <> mInternal(){ return new <>(); }
  outer.A mExternal(){ return new outer.A(); }
}
B = A
```

is expected, with a very intuitive copy semantics, to be equivalent to the following:

```
A = // as before
B = {
  <> mInternal(){ return new <>(); }
  outer.A mExternal(){ return new outer.A(); }
}
```

Note that B.mInternal returns a B, while B.mExternal returns an A. In other words, when a basic class is duplicated in a new position, the class paths refering to the outside, such as the

[2]In FJIG$_\star$, differently from Java, Featherweight Java and FJIG$_1$, we have a notion of *top-level class expression* rather than *top-level program*.

return type **outer**.A, will still denote the "old" class, whereas other class paths, such as the return type <>, will denote a new class.

In more complex cases, class paths in the original code need to be modified to achieve this semantics. For example, consider the program

```
A = {
  B = {
    C = { ... }
    C m1() { ... }
    outer.B.C m2() { ... }
    outer.outer.A.B.C m3() { ... }
  }
}
D = A.B
```

where the three class paths which appear as return types denote the same class, which is denoted by A.B.C at top level.

At first sight, there is no difference among these three class paths, so one could think of normalizing code by always using the shortest class path denoting a given class in a given position, e.g., C in the example above. However, this is not the case, since this program is expected to be equivalent to the following:

```
A = // as before
D = {
  C = { ... }
  C m1() { ... }
  outer.A.B.C m2() { ... }
  outer.A.B.C m3() { ... }
}
```

In the definition of D the return type C denotes now a new class, denoted by D.C at top level, whereas the other two return types have been changed in order to still denote the "old" class. This will be expressed by the notation $C[\mathsf{from}\ C^s]$, formally defined in Figure 5.3, which returns the class path obtained by "moving" C from position C^s to the current position.

FJIG$_\star$ keeps the Java and FJIG nominal approach, that is, types are a generalization of class names, and two different types which are structurally equivalent are *not* considered equal. Note that this also holds for outer class paths, e.g., <> and **outer** are incompatible types, even in case the corresponding basic classes have exactly the same fields and methods.

Deep composition operators

In FJIG$_\star$, composition operators are *deep*, in the sense that they allow to manipulate nested classes at any depth level.

For instance, the sum of two classes "propagates" to their nested classes with the same name, similarly to *deep mixin composition* [OZ05], as shown by the following example:

```
C = {
  N = abstract{
    abstract int n();
    int m(){ return this.n(); }
  }
}
D =
  C [+]
  {
    N = {
      int n(){ return 1; }
      abstract int m();
    }
    int k(){ return new N().m(); }
  }
```

which is equivalent to the following:

```
C = // as before
D = {
  class N = {
    int n(){ return 1; }
    int m(){ return this.n(); }
  }
  int k(){ return new N().m(); }
}
```

The effect of the sum operator is that nested class `N` of `C` is summed with nested class `N` of the unnamed basic class. In this way, the resulting class `N` of `D` inherits the implementation for methods `n` and `m` from `N` of `C` and `N` of the unnamed basic class, respectively.

This example also illustrates the meaning of the modifier **`abstract`**. As in FJIG$_1$, the effect of the modifier is to forbid the creation of instances of a given class. However, in FJIG$_\star$, differently from FJIG$_1$, it is perfectly legal to declare abstract members inside a non abstract class, as shown by nested class `N` of the unnamed basic class above. The meaning is that the class is *incomplete*, that is, not executable. However, it can be safely used as a library, since it can be completed by composition with another class, as actually happens in the example, where method `k` of `D` can

correctly create an instance of nested class N of D.

Besides sum, FJIG$_\star$ provides other five composition operators: *restrict*, *alias*, *redirect*, *class alias*, and *class redirect*.

- Operators *restrict*, *alias* and *redirect* work as in FJIG$_1$, but can be applied to an arbitrary nested class.
- The *class alias* operator adds a definition for a nested class, duplicating that of an existing nested class. If a definition for the target is already present, then it is summed with the new definition.
- The *class redirect* operator replaces all the references to a class by a different class.

The next examples illustrate these operators in more detail.

```
C = {
  C = abstract{
    abstract int n1();
    int n2(){ return 2; }
    int n3(){ return 3; }
  }
  K = {
    int n1(){ return 10; }
    int n2(){ return 20; }
  }
  int m(){ return new K().n1(); }
}
Restrict = C[restrict n2 in .C]
Alias = C[alias n2 to n1 in .C]
AliasC = C[alias .K to .C.K]
AliasCSum = Restrict[alias .K to .C]
Redirect = C[redirect n1 of .K to n2]
RedirectC1 = AliasC[redirect .K to .C.K]
RedirectC2 = AliasC[redirect .K to C.K]
```

Before discussing the semantics of each operator, note that all of them have a class as first argument, and other arguments are either class paths or *paths*. Paths are sequences of the form $.c_1.\ldots.c_n$, which denote, as the notation suggests, a nested position inside the class occurring as first argument. Note the difference with class paths, which denote an existing class in the nesting hierarchy. For instance, in the class declaration

```
Restrict = C[restrict n2 in .C]
```

the class path occurring as first argument denotes the top-level class `C`, whereas the path (nested position) `.C` inside this class corresponds to the class `C.C`. Note also that, if we replace class path `C` with its defining class expression

```
Restrict = {
  C = abstract{
    abstract int n1();
    int n2(){ return 2; }
    int n3(){ return 3; }
  }
  K = {
    int n1(){ return 10; }
    int n2(){ return 20; }
  }
  int m(){ return new K().n1(); }
}[restrict n2 in .C]
```

the path `.C` denotes a nested position in an anonymous basic class, corresponding to a class which *cannot* be denoted by a class path.

The empty path $.\Lambda$ is written `.<>` in code.

The restrict operator removes a definition in a nested class, making the corresponding member abstract. For class `Restrict` we get the following definition:

```
Restrict = {//C[restrict n2 in .C]
  C = abstract{
    abstract int n1();
    abstract int n2();//now abstract
    int n3(){ return 3; }
  }
  K = // as before
  int m(){ return new K().n1(); }
}
```

Note that a **`restrict`** operator for classes makes no sense. On the other hand, a derived operator which recursively makes abstract all fields and methods of a class can be easily defined.

The alias operator duplicates the declaration of an existing field or method of a nested class (including `.<>`), for another field or method of the same class. Hence we get the following definition:

```
Alias = {//C[alias n2 to n1 in .C]
  C = abstract{
    int n1(){ return 2; }//now implemented
    int n2(){ return 2; }
```

```
    int n3(){ return 3; }
  }
  K = // as before
  int m(){ return new K().n1(); }
}
```

where the method `.C.n1` is now implemented using the implementation of `.C.n2`.

The class alias operator adds or modifies a nested class, by duplicating an existing basic class. More precisely, if there is no nested class in the target position (non empty path), then a new class declaration is inserted, as in the `AliasC` example. Hence we get the following definition:

```
AliasC = {//C[alias .K to .C.K]
  C = abstract{
    abstract int n1();
    int n2(){ return 2; }
    int n3(){ return 3; }
    K = {
      int n1(){ return 10; }
      int n2(){ return 20; }
    }
  }
  K = // as before
  int m(){ return new K().n1(); }
}
```

If, instead, there is already a nested class in the target position, as in the `AliasCSum` example, then the duplicated class is summed with the existing class. Hence we get the following definition:

```
AliasCSum = {//Restrict[alias .K to .C]
  C = {
    int n1(){ return 10; }
    int n2(){ return 20; }
    int n3(){ return 3; }
  }
  K = // as before
  int m(){ return new K().n1(); }
}
```

where nested class `.C` has been obtained by sum, hence has now implementations for `n1` and `n2` copied from `.K`.

The redirect operator replaces all the references to a field or method name whose receiver's static type is a nested class, by a different name, and removes its declaration. Hence we get the

following definition:

```
Redirect = {//C[redirect n1 of .K to n2]
  C = // as before
  K = {
    //int n1(){ return 10; }; //removed
    int n2(){ return 20; }
  }
  int m(){ return new K().n2(); }
}
```

where the the declaration of `.K.n1` has been removed, and the invocation of `.K.n1` is now an invocation of `.K.n2`.

The class redirect operator replaces all the references to a class by a different class, and removes its declaration. Hence we get the following definition:

```
RedirectC1 = {//AliasC[redirect .K to .C.K]
  C = // as before
  //K = // removed
  int m(){ return new C.K().n1(); }
}
```

where nested class `.K` has been removed, and the constructor invocation refers now to class `.C.K`.

Note the difference with the following case where the last argument is a class path:

```
RedirectC2 = {//AliasC[redirect .K to C.K]
  C = // as before
  //K = // removed
  int m(){ return new outer.C.K().n1(); }
}
```

On top of these four composition primitives, we can derive many other useful operators. For instance, the **`override`** operator already seen in FJIG$_1$, a variant of sum where conflicts are allowed and the left argument has the precedence, can be defined as follows:

```
C1[override]C2 ≡
  C1[+](C2[restrict n1 in .N1] ... [restrict nk in .Nk])
```

where **`restrict`** is applied to all fields or methods with the same name `ni` defined in nested class `Ni` in both `C1` and `C2`. Indeed, here override is deep, that is, it propagates to nested classes analogously to sum.

It is possible to define also a **`rename`** operator for nested classes as follows:

```
C[rename .COld to .CNew] ≡
  C[alias .COld to .CNew][redirect .COld to .CNew]
```

If `COld` has nested classes, then we need to recursively apply redirect to these classes.

Renaming of methods and fields can be encoded exactly as in FJIG$_1$:

```
C[rename nOld to nNew in .N]  ≡
  C[alias nOld to nNew in .N][redirect nOld of .N to nNew]
```

5.2 Expressive power

Expression problem First of all we show the expressive power of FJIG$_\star$ by considering as "benchmark" the classical *expression problem* (or *extensibility problem*) [Tor04, OZ05].

The expression problem can be formulated as follows: we have a datatype defined by a set of *variants*, and we have *processors* which operate on this datatype. The addition of new data variants and new processors are the two directions along which the system can be extended. The challenge is to do it in a modular and easy way.

For sake of concreteness, let us consider a `Base` class, modelling arithmetic expressions, defined as follows:

```
Base = {
  Expression = abstract{
    abstract String toString();
  }
  Num = implements outer.Expression{
    int e;
    constructor(int e){ this.e = e; }
    String toString(){ return ""+this.e; }
  }
  Sum = implements outer.Expression{
    outer.Expression l;
    outer.Expression r;
    constructor(outer.Expression l, outer.Expression r){
      this.l = l;
      this.r = r;
    }
    String toString(){
      return "("+this.l.toString()+"+"+this.r.toString()+")";
    }
  }
```

```
}
```

Assume that now Adam wants to add a `UMinus` class. This can be done in this way:

```
AddUMinus = {
  Expression = abstract{
    abstract String toString();
  }
  UMinus = implements outer.Expression{
    outer.Expression e;
    constructor(outer.Expression e){ this.e = e; }
    String toString(){return "-"+this.e.toString();}
  }
}
BaseWithUMinus = Base [+] AddUMinus
```

Bob wants to add an `eval` operator. This can be done in this way:

```
EvalBase = {
  Expression = abstract{ abstract int eval(); }
  Num = abstract implements outer.Expression{
    abstract int e;
    constructor(int e){}
    int eval(){ return this.e; }
  }
  Sum = abstract implements outer.Expression{
    abstract outer.Expression l;
    abstract outer.Expression r;
    constructor(outer.Expression l, outer.Expression r){}
    int eval(){ return this.l.eval()+this.r.eval(); }
  }
}
BaseWithEval = Base [+] EvalBase
```

Charles wants to use the work of Adam and Bob to obtain something with both the `UMinus` variant and the `eval` processor. The first step is to define the behaviour of `eval` on the `UMinus` variant.

```
EvalUMinus = {
  Expression = abstract{ abstract int eval(); }
  UMinus = abstract implements outer.Expression{
    abstract outer.Expression e;
    constructor(outer.Expression e){}
    int eval(){ return -this.e.eval(); }
  }
```

```
}
```

Now Charles has the following data variants and processors to deal with.

	constructor	`toString`	`eval`
`Num`	`Base`		`EvalBase`
`Sum`			
`UMinus`	`AddUMinus`		`EvalUMinus`

He has two legal ways to compose everything together:

- first `AddUMinus` with `EvalUMinus`, obtaining a fully fledged variant, and then the result with `BaseWithEval`;
- first `EvalBase` with `EvalUMinus`, obtaining a fully fledged `eval` processor, and then the result with `BaseWithUMinus`.

```
Solution1 = (AddUMinus [+] EvalUMinus) [+] BaseWithEval
Solution2 = (EvalBase [+] EvalUMinus) [+] BaseWithUMinus
```

This solution to the expression problem is very natural and fulfils all the requirements given in [OZ05], that is: extensibility in both dimensions, strong static type safety, no modification or duplication of source code[3], separate compilation, independent extensibility. However, what we have done is to "patch" some already existing code. It is possible to be even more modular if the software is written from the beginning in a fully modular way, that is: for each `Variant` of a `DataType`, we define a class `ConstrVariant`, containing field initializations.

```
ConstrVariant  = {
  DataType = abstract{}
  Variant = implements DataType{
   field declarations
   constructor(...) { ... }
  }
}
```

For each `Variant` and `processor` we define the corresponding processor implementation in a class `ProcessorVariant`.

[3]However, code is expanded by a precompilation step, formally modelled by the *flattening* relation in Figure 5.2. Code expansion could be delayed at invocation time by defining a *direct semantics*, as in [LSZ09c], generalizing dynamic method look-up.

```
ProcessorVariant = {
  DataType = abstract{ abstract processor(); }
  Variant =  abstract implements DataType{
    abstract field declarations required by processor
    processor(){...}
  }
}
```

Now we have a full grid of processors and data variants. For example, for improving modularity, we could have split the `Base` class defined before into four pieces: `ConstrNum`, `ConstrSum`, `ToStringNum`, `ToStringSum`. Analogously, `EvalBase` can be split in `EvalNum` and `EvalSum`, and `AddUMinus` in `ConstrUMinus` and `ToStringUMinus`.

	constructor	toString	eval
Num	ConstrNum	ToStringNum	EvalNum
Sum	ConstrSum	ToStringSum	EvalSum
UMinus	ConstrUMinus	ToStringUMinus	EvalUMinus

This allows the user to take any coherent (that is, where all existing processors are defined over all existing variants) subset of the cells of the grid, and extending the grid is also very natural. Analogously, the extension must be coherent, that is, the type system requires to add an entire row or column, to ensure that we are not leaving unmanaged cases.

This possibility of taking only a *subset* of the classes composing an application nicely implements the concept of *scalable-down* architecture [Par78], that is, a software architecture which can be not only easily extended, but also contracted when less functionalities are needed. Note that this means that code size is *truly* reduced, not just that some functionality is hidden as in other approaches.

Generics and `MyType` In FJIG$_\star$ we can encode generics. Indeed, at the foundational level it has been proved a long time ago [WV00, AZ02] that module calculi can encode lambda-calculus, and thanks to nesting FJIG$_\star$ classes play the role of modules with class components, similarly, e.g., to JAVAMOD [AZ01], a module layer for Java classes where an analogous encoding was possible. However, here the encoding is much more natural and does not require additional notions, as shown by the example below.

```
OList = {
  Elem = abstract{
    abstract boolean geq(<> other);
  }
  List = abstract{
    abstract <> insert(outer.Elem e);
```

```
  }
  EmptyList = implements outer.List{
    outer.List insert(outer.Elem e){
      return new outer.NonEmptyList(e,this);
    }
  }
  NonEmptyList = implements outer.List{
    outer.Elem e;
    outer.List tail;
    constructor(Elem e, outer.List tail){
      this.e=e;
      this.tail=tail;
    }
    outer.List insert(outer.Elem e){
      if(this.e.geq(e))return new <>(e,this);
      return new <>(this.e,this.tail.insert(e));
    }
  }
}
MyElem = {
  int e;
  constructor(int e){this.e=e;}
  boolean geq(<> other){
    return this.e>=other.e;
  }
}
MyElemOList = OList[redirect .Elem to MyElem]
```

The class `OList.List` models an ordered list of `Elem`, that offers a binary method `geq`. By the redirect operator it is possible to produce an instantiation of `OList` which represents a list of `MyElem`.

The example also shows the binary method `geq` where, as already mentioned, the class path `<>` plays the same role of **`MyType`** [BOW98], or **`ThisClass`** of LOOJ [BF04]. **`MyType`** can be used inside a method of a class to refer to the class itself, and, similarly to what happens with **`this`**, is redirected to the proper subclass when the method is inherited. Again, **`MyType`** was already expressible in a previous work on a module layer for Java classes [ALZ06], but here it is smoothly integrated with the overall language design, being just a special case of class path.

Note that in both cases there is no true polymorphism, since, as already mentioned, code for each different instantiation is obtained by expanding generic code by flattening. The advantage is that we can keep a standard Java-like type system.

Refactoring Refactoring tools allow to perform useful code transformations, notably renaming. For instance, using the rename operator

```
A = { ...
  C = { ... }
}[rename C to D]
B = { ... }
```

allows class `B` to use `A.D` instead of `A.C`.

The code transformation which *moves* a class up or down in the nesting hierarchy can also be encoded as a renaming. For instance:

```
A = {
  B = {
    C = { ... }
  }
}[rename B.C to C]
E = { ... }
```

allow class `E` to use `A.C` instead of `A.B.C`.

Note that a programming environment could allow the user to choose whether to perform flattening of a single class, getting the new source, or to leave the class expression as it stands, to be able to perform the roll back by simply removing the refactoring code. For instance, when working with a library, it can be useful to leave refactoring operations in the code, so that when the next version of the library is released, it can be seamless integrated with the application.

AOP Class composition languages and aspect-oriented programming take a different approach: the former construct new classes from existing ones, while the latter modifies the whole application at once. Since in FJIG$_\star$ "the whole application" is a class expression, we can use composition operators to modify the whole application as well. In this way, the effect is analogous to AOP in many respects: flattening is a code expansion as weaving, anonymous basic classes play the role of advices and composition operators individuate the pointcuts, even though the latter can be specified by a richer language.

Among the many kinds of code modifications allowed by aspects, the most relevant are *execution-around* and *call-around* [KHH+01]: the former replaces execution of a given method, determined by the receiver's dynamic type, the latter replaces invocation of a given method, determined by the receiver's static type.

Consider for instance the following basic class b:

```
{
  A = {
```

```
    int foo(){return 1; }
  }
  B = implements outer.A{
    int foo(){ return 2; }
  }
  int bar(A a){ return a.foo(); }
  String main(){ return new B().foo()+" "+
      this.bar(new B())+" "+this.bar(new A());
  }
}
```

Here a call of `main` produces `"2 2 1"`. Execution-around can be easily emulated by our operators; for example, this AspectJ-like code:

```
int around(): execution(int A.foo()){ return 10; }
```

can be encoded by

```
b[restrict foo in .A] [+]
  { A = { int foo(){ return 10; } } }
```

Now `main` produces `"2 2 10"` since we changed the result of the third call, which is the only one whose receiver has dynamic type `A`. Note that, instead of writing an `A` class with a `foo` method, we could have used an arbitrary named class with an arbitrary named method, and then the **`rename`** operator, to stress that basic classes and composition operators correspond to advices and pointcuts, respectively. However, we find this solution more readable.

Emulating call-around code like

```
int around(): call(int A.foo()) { return 10; }
```

requires a little more effort:

```
(b [+]
  { A = { int foo2(){return 10; } } }[alias .A to .B]
) [redirect foo of .A to foo2]
```

Now `main` produces `"2 10 10"` since we changed the behaviour of all invocations of `foo` whose receiver has static type `A`. The call of `B.foo` is not affected. Note that the **`alias`** is needed to keep `B` subtype of `A`. As in the AOP tradition, this approach does not require to change the source, that is, it is "not-invasive". The operation *proceed* can be encoded using the same pattern one can use to emulate *super* calls, that is, as calls to an alias of the original method. *After* and *before* can be encoded by *around* and a call to *proceed*. Other operation, like dynamic pointcuts, are much more complex to express (essentially, the same encoding used by the AspectJ implementation is needed).

5.3 Syntax and semantics

Syntax The syntax of the language is given in Figure 5.1. There are few differences w.r.t. FJIG$_1$; however, we repeat the whole syntax for sake of clarity.

Class expressions are basic classes, class paths, or are constructed by composition operators.

In a basic class, the key novelty w.r.t. FJIG$_0$ and FJIG$_1$ is that a declaration can now be also a (nested) class declaration. Correspondingly, we distinguish between *instance member names*, ranged over by i, which are names for fields and methods, and *member names*, which include both instance member names and class names. We use the term "instance member" to denote either a field or a method since nested classes are static members instead.

Supertype declarations, constructors, field and method declarations are exactly as in FJIG$_0$ and FJIG$_1$.

Class paths are sequences of the form **`outer.`**$\ldots$**`.outer.`**c_1**`.`**$\ldots$**`.`**c_n, whereas *paths* are sequences of the form **`.`**c_1**`.`**$\ldots$**`.`**c_n. Class paths and paths have a different syntactic and semantic role: a class path refers to an existing class in the nesting hierarchy, hence can be used the same way as a class name, that is, as type, in new expressions and as class expression. A path, instead, is used as argument of composition operators, referring to a nested position inside a class. For instance, the class alias operator requires a path π as last argument, since a new nested (or modified) class is inserted in position π.

We assume that, in a well-formed top-level class expression, all class paths refer to existing classes.

Expressions in method bodies are exactly the same of FJIG$_0$ and FJIG$_1$. However, field accesses and method invocations are annotated with the static type of the receiver. These annotations can be added during typechecking. However, for simplicity we do not model here two different languages and assume that they are already in source code. This is needed for the redirect operator, seen in the following.

Differently from FJIG$_1$, basic classes do no longer play the role of values w.r.t. the flattening relation. Indeed, since operators are deep, they can be applied only to basic classes where all nested class definitions are (recursively) basic classes. This is formally modelled by the definition of *class values* in Figure 5.1.

Flattening rules Figure 5.2 contains the rules defining the flattening relation. The relation is of the form $ce_1 \rightarrow ce_2$, where the unique rule (CTX) reduces the whole application (top-level class expression) by applying a reduction step to either the top-level class expression, or to a class expression appearing as right-hand side of a nested class declaration, at any level of depth. This is formally expressed by the contexts for flattening $\mathcal{CE}^{\mathrm{f}}$, defined in terms of the conventional

ce	$::=$		class expression
		b	basic class
		$\mid C$	class path
		$\mid ce_1\,[+]\,ce_2$	sum
		$\mid ce\,[\texttt{restrict}\,i\,\texttt{in}\,\pi]$	restrict
		$\mid ce\,[\texttt{alias}\,i^s\,\texttt{to}\,i^t\,\texttt{in}\,\pi]$	alias
		$\mid ce\,[\texttt{alias}\,\pi^s\,\texttt{to}\,\pi^t]$	class alias 1
		$\mid ce\,[\texttt{alias}\,C^s\,\texttt{to}\,\pi^t]$	class alias 2
		$\mid ce\,[\texttt{redirect}\,i^s\,\texttt{of}\,\pi\,\texttt{to}\,i^t]$	redirect
		$\mid ce\,[\texttt{redirect}\,\pi^s\,\texttt{to}\,\pi^t]$	class redirect 1
		$\mid ce\,[\texttt{redirect}\,\pi^s\,\texttt{to}\,C^t]$	class redirect 2
b	$::=$	$ch\,\{k\,\overline{d}\}$	basic class
ch	$::=$	$\mu\,\texttt{implements}\,\overline{C}$	class header
k	$::=$	$kh\{\overline{fe}\}$	constructor
kh	$::=$	$\texttt{constructor}\,(\overline{T\,x})$	constructor header
fe	$::=$	$\texttt{this}.f = e\,\texttt{;}$	field expression
d	$::=$	$fd \mid md \mid cd$	(member) declaration
fd	$::=$	$\mu\ T\,f\,\texttt{;}$	field declaration
md	$::=$	$\texttt{abstract}\ mh; \mid mh\{\texttt{return}\,e\,\texttt{;}\}$	method declaration
cd	$::=$	$c = ce$	class declaration
mh	$::=$	$T\,m(\overline{T\,x})$	method header
μ	$::=$	$\epsilon \mid \texttt{abstract}$	abstract modifier
n	$::=$	$i \mid c$	(member) name
f, m	$::=$	i	
T	$::=$	$C \mid \ldots$	type
C	$::=$	$\overline{\texttt{outer}}.\overline{c}$	class path
π	$::=$	$.\overline{c}$	path
e	$::=$	$x \mid e.[C]f \mid e.[C]m\,(\overline{e}) \mid \texttt{new}\,C\,(\overline{e})$	expression (conventional)
		$\mid C\,(\overline{fe})$	expression (pre-object)
v	$::=$	$C\,(\overline{fv})$	value
fv	$::=$	$\texttt{this}.f = v\,\texttt{;}$	field value
cv	$::=$	$ch\,\{k\,\overline{fd}\,\overline{md}\,\overline{c = cv}\}$	class value
σ	$::=$	$\overline{b}$	enclosing classes

Figure 5.1: FJIG$_\star$ syntax

context $\mathcal{CE}$.

The flattening relation for class expressions is of the form $ce_1 \underset{\sigma}{\rightarrow} ce_2$. Indeed, reduction of a class expression takes place in a *environment* σ which is the stack of all its enclosing basic classes, starting from the directly enclosing, needed to give semantics to outer class paths. We denote by $\mathsf{enclosing}(ce, \mathcal{CE}^{\mathrm{f}})$ the stack of basic classes enclosing the hole in $\mathcal{CE}^{\mathrm{f}}$, formally:

- $\mathsf{enclosing}(ce, \mathcal{CE}[\![\Box]\!]) = \Box$
  $\mathsf{enclosing}(ce, \mathcal{CE}[\![ch\ \{\, k\ \overline{d}\ c = \mathcal{CE}^{\mathrm{f}}\, \}]\!]) = ch\ \{\, k\ \overline{d}\ c = \mathcal{CE}^{\mathrm{f}}[\![ce]\!]\, \} \cdot \mathsf{enclosing}(ce, \mathcal{CE}^{\mathrm{f}})$

Rule (CLASS-PATH) can be applied when the class expression occurring in position C in σ is a class value (FJIG$_\star$ has a call-by-value semantics). The notation $\mathsf{cBody}(\sigma, C)$ is formally defined by:

- $\mathsf{cBody}(b_0 \ldots b_n, \mathbf{outer}^i.\overline{c}) = \mathsf{cBody}(b_i, .\overline{c})$ with $i \leq n$
  $\mathsf{cBody}(ch\ \{\, k\ \overline{d}\ (c = b)\, \}, .c.\pi) = \mathsf{cBody}(b, \pi)$
  $\mathsf{cBody}(b, .\Lambda) = b$

In this case, C is replaced by cv, a class value obtained from $\mathsf{cBody}(\sigma, C)$, denoting a class value in position C w.r.t. the current position, by "moving" all the occurrences of external class paths so that they still denote the same class.

The notations $cv[\mathsf{from}\ C^s]$, and $e[\mathsf{from}\ C^s]$ where s stands for "source", meaning "moving class value cv from C^s to the current position", and "moving expression e from C^s to the current position", respectively, are defined in Figure 5.3. Moving a class path to the current position is defined by an accumulation parameter j corresponding to the nesting level where the class path occurrence is found. For simplicity, we omit all trivial propagation clauses. A class path occurrence found at nesting level j needs to be moved only if it is external, that is, has form $\mathbf{outer}^n.\pi$ with n greater or equal to j. In this case, $\mathbf{outer}^n$ is replaced by the class path obtained from $\mathbf{outer}^j.C^s \equiv \mathbf{outer}^m.\pi'$ by removing from π', from right to left, n elements, adding **outer**s if there are not elements enough. For instance, in the example in Section 5.1,

```
A = {
  B = {
    C = { ... }
    C m1() { ... }
    outer.B.C m2() { ... }
    outer.outer.A.B.C m3() { ... }
  }
}
D = A.B
```

$$
\begin{array}{rcl}
\mathcal{CE} & ::= & \Box \mid \mathcal{CE}\,[+]\,ce \mid ce\,[+]\,\mathcal{CE} \mid \mathcal{CE}\,[\,\mathbf{restrict}\ i\ \mathbf{in}\ \pi\,] \\
& & \mid \mathcal{CE}\,[\,\mathbf{alias}\ i^s\ \mathbf{to}\ i^t\ \mathbf{in}\ \pi\,] \mid \mathcal{CE}\,[\,\mathbf{alias}\ C^s\ \mathbf{to}\ \pi^t\,] \mid \mathcal{CE}\,[\,\mathbf{alias}\ \pi^s\ \mathbf{to}\ \pi^t\,] \\
& & \mid \mathcal{CE}\,[\,\mathbf{redirect}\ i^s\ \mathbf{of}\ \pi\ \mathbf{to}\ i^t\,] \mid \mathcal{CE}\,[\,\mathbf{redirect}\ \pi^s\ \mathbf{to}\ C^t\,] \mid \mathcal{CE}\,[\,\mathbf{redirect}\ \pi^s\ \mathbf{to}\ \pi^t\,] \\
\mathcal{CE}^{\mathrm{f}} & ::= & \mathcal{CE}[\![\Box]\!] \mid \mathcal{CE}[\![\_\,\{\,\_\_\ c = \mathcal{CE}^{\mathrm{f}}\,\}]\!]
\end{array}
$$

$$ce_1 \rightarrow ce_2$$

$$\text{(CTX)}\ \frac{ce_1 \underset{\sigma}{\rightarrow} ce_2}{\mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] \rightarrow \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!]}\ \sigma = \mathsf{enclosing}(ce_1, \mathcal{CE}^{\mathrm{f}})$$

$$ce_1 \underset{\sigma}{\rightarrow} ce_2$$

$$\text{(CLASS-PATH)}\ \frac{}{C \underset{\sigma}{\rightarrow} cv}\ cv = \mathsf{cBody}(\sigma, C)[\mathsf{from}\ C] \qquad \text{(SUM)}\ \frac{}{cv_1\,[+]\,cv_2 \underset{\sigma}{\rightarrow} cv_1 \oplus cv_2}$$

$$\text{(RESTRICT)}\ \frac{}{cv\,[\,\mathbf{restrict}\ i\ \mathbf{in}\ \pi\,] \underset{\sigma}{\rightarrow} cv \ominus_\pi i \oplus_\pi \mathsf{abs}(d)}\ d = \mathsf{dec}(cv, \pi, i)$$

$$\text{(ALIAS)}\ \frac{}{cv\,[\,\mathbf{alias}\ i^s\ \mathbf{to}\ i^t\ \mathbf{in}\ \pi\,] \underset{\sigma}{\rightarrow} cv' \oplus_\pi \mathsf{named}(i^t, d)}\ \begin{array}{l}\mathsf{constr}(cv, \pi) = kh\,\{\overline{fe}\} \\ cv' = \begin{cases} cv \oplus_\pi \mathbf{this}.i^t = e; & \text{if } \overline{fe}(i^s) = e \\ cv & \text{if } i^s \notin \mathsf{dom}(\overline{fe}) \end{cases} \\ d = \mathsf{dec}(cv, \pi, i^s)\end{array}$$

$$\text{(CLASS-ALIAS1)}\ \frac{}{cv\,[\,\mathbf{alias}\ .\overline{c}\ \mathbf{to}\ \pi^t.c\,] \underset{\sigma}{\rightarrow} cv \oplus_{\pi^t} (c = cv')}\ cv' = \mathsf{cBody}(cv, .\overline{c})[\mathsf{from}\ \overline{c}[\mathsf{to}\ \pi^t]]$$

$$\text{(CLASS-ALIAS2)}\ \frac{}{cv\,[\,\mathbf{alias}\ C^s\ \mathbf{to}\ \pi^t.c\,] \underset{\sigma}{\rightarrow} cv \oplus_{\pi^t} (c = cv')}\ cv' = \mathsf{cBody}(\sigma, C^s)[\mathsf{from}\ \mathbf{outer}^{|\pi^t|}.C^s]$$

$$\text{(REDIRECT)}\ \frac{}{cv\,[\,\mathbf{redirect}\ i^s\ \mathbf{of}\ \pi\ \mathbf{to}\ i^t\,] \underset{\sigma}{\rightarrow} (cv \ominus_\pi i^s)\big[i^s \underset{[\pi]}{\rightsquigarrow} i^t\big]}\ \begin{array}{l} i^s \in \mathsf{names}(cv, \pi) \\ i^s \neq i^t \end{array}$$

$$\text{(CLASS-REDIRECT1)}\ \frac{}{cv\,[\,\mathbf{redirect}\ \pi^s.c\ \mathbf{to}\ .\overline{c}\,] \underset{\sigma}{\rightarrow} (cv \ominus_{\pi^s} c)\big[\pi^s.c \rightsquigarrow \overline{c}\big]}\ \begin{array}{l} \mathsf{noNested}(cv, \pi^s.c) \\ \pi^s.c \neq .\overline{c} \end{array}$$

$$\text{(CLASS-REDIRECT2)}\ \frac{}{cv\,[\,\mathbf{redirect}\ \pi^s.c\ \mathbf{to}\ C^t\,] \underset{\sigma}{\rightarrow} (cv \ominus_{\pi^s} c)\big[\pi^s.c \rightsquigarrow \mathbf{outer}.C^t\big]}\ \mathsf{noNested}(cv, \pi^s.c)$$

Figure 5.2: FJIG$_\star$ flattening rules

$cv[\text{from } C^s]$

$cv[\text{from } C^s] = cv[\text{from } C^s \setminus 1]_0$
$e[\text{from } C^s] = e[\text{from } C^s]_0$
$ch\ \{\, k\ \overline{d}\,\}[\text{from } C^s]_j = ch[\text{from } C^s]_{j+1}\ \{\, k[\text{from } C^s]_{j+1}\ \overline{d}[\text{from } C^s]_{j+1}\,\}$
$$C[\text{from } C^s]_j = \begin{cases} \texttt{outer}^j.(C'[\text{from } C^s]) & \text{if } C = \texttt{outer}^j.C' \\ C & \text{otherwise} \end{cases}$$

$C[\text{from } C^s]$

$\texttt{outer}^n.\overline{c}[\text{from } \texttt{outer}^m.\overline{c}'] = \texttt{outer}^m.(\overline{c}'\backslash n).\overline{c}$ where
$$c_1.\ \ldots\ .c_k\backslash n = \begin{cases} c_1 \ldots c_{k-n} & \text{if } n \leq k \\ \texttt{outer}^{n-k} & \text{if } n > k \end{cases}$$

Figure 5.3: Auxiliary definitions for moving class paths

class paths `outer.B.C` and `outer.outer.A.B.C` are "moved" from position `A.B`, where they are nested at level 0, hence they are external. By removing, respectively, one and two right elements from `outer.A.B` we get `outer.A` and `outer`, respectively, hence in the end we get two times `outer.A.B.C`.

The other rules model composition operators.

Rule (SUM) is analogous to the one of FJIG$_1$, however, differently from FJIG$_1$, the resulting class value has modifier abstract only if both were abstract. Moreover here two (basic) class declarations are merged in one where the basic class is the sum of the two class values.

For sake of clarity we repeat the whole definition of $cv_1 \oplus cv_2$.

- If $cv_i = \mu_i\ \texttt{implements}\ \overline{C}_i\ \{\, kh\,\{\overline{fe}_i\}\ \overline{d}_i\,\}$, then
  $cv_1 \oplus cv_2 = \mu\ \texttt{implements}\ \overline{C}_1\ \overline{C}_2\ \{\, kh\,\{\overline{fe}_1\ \overline{fe}_2\}\ \overline{d}_1 \oplus \overline{d}_2\,\}$
  where $\mu = \texttt{abstract}$ iff $\mu_1 = \mu_2 = \texttt{abstract}$

- $\overline{d}_1 \oplus \overline{d}_2$ is defined by:

  - $\text{dom}(\overline{d}_1 \oplus \overline{d}_2) = \text{dom}(\overline{d}_1) \cup \text{dom}(\overline{d}_2)$

- $(\overline{d}_1 \oplus \overline{d}_2)(n) = \begin{cases} \overline{d}_1(n) & \text{if } n \in \mathsf{dom}(\overline{d}_1) \setminus \mathsf{dom}(\overline{d}_2) \\ \overline{d}_2(n) & \text{if } n \in \mathsf{dom}(\overline{d}_2) \setminus \mathsf{dom}(\overline{d}_1) \\ \overline{d}_1(n) \oplus \overline{d}_2(n) & \text{if } n \in \mathsf{dom}(\overline{d}_2) \cap \mathsf{dom}(\overline{d}_2) \end{cases}$
- $\texttt{abstract}\ mh\texttt{;} \oplus \texttt{abstract}\ mh\texttt{;} = \texttt{abstract}\ mh\texttt{;}$
- $\texttt{abstract}\ mh\texttt{;} \oplus mh\,\{\,\texttt{return}\ e\texttt{;}\,\} =$
  $mh\,\{\,\texttt{return}\ e\texttt{;}\,\} \oplus \texttt{abstract}\ mh\texttt{;} = mh\,\{\,\texttt{return}\ e\texttt{;}\,\}$
- $\texttt{abstract}\ T\ f\texttt{;} \oplus \mu\ T\ f\texttt{;} = \mu\ T\ f\texttt{;} \oplus \texttt{abstract}\ T\ f\texttt{;} = \mu\ T\ f\texttt{;}$
- $(c = cv_1) \oplus (c = cv_2) = c = (cv_1 \oplus cv_2)$

In (RESTRICT), the operator replaces the definition of field or method i in nested class π by the corresponding abstract declaration. Notations $cv \ominus_\pi n$, $cv \oplus_\pi d$ and $\mathsf{named}(n, d)$ are the obvious extension of the corresponding ones in FJIG$_1$. Formally:

- $ch\ \{\,k\ \overline{d}\ (c = cv)\,\} \ominus_{.c.\pi} n = ch\ \{\,k\ \overline{d}\ (c = cv \ominus_\pi n)\,\}$
  $ch\ \{\,kh\,\{\,\overline{fe}\,\}\ \overline{d}\,\} \ominus_{.\Lambda} n = ch\ \{\,kh\,\{\,\overline{fe} \setminus n\,\}\ \overline{d} \setminus n\,\}$
- $ch\ \{\,k\ \overline{d}\ (c = cv)\,\} \oplus_{.c.\pi} d = ch\ \{\,k\ \overline{d}\ (c = cv \oplus_\pi d)\,\}$
  $ch\ \{\,k\ \overline{d}\,\} \oplus_{.\Lambda} d = ch\ \{\,k\ \overline{d} \oplus d\,\}$
- $\mathsf{abs}(\mu\ T\ f\texttt{;}) = \texttt{abstract}\ T\ f\texttt{;}$
  $\mathsf{abs}(\texttt{abstract}\ mh\texttt{;}) = \texttt{abstract}\ mh\texttt{;}$
  $\mathsf{abs}(mh\,\{\,\texttt{return}\ e\texttt{;}\,\}) = \texttt{abstract}\ mh\texttt{;}$
- $\mathsf{dec}(ch\ \{\,k\ \overline{d}\ (c = cv)\,\}, .c.\pi, n) = \mathsf{dec}(cv, \pi, n)$
  $\mathsf{dec}(ch\ \{\,k\ \overline{d}\ \mathsf{named}(n, d)\,\}, .\Lambda, n) = \mathsf{named}(n, d)$
- $\mathsf{named}(i, \mu\ T\ f\texttt{;}) = \mu\ T\ i\texttt{;}$
  $\mathsf{named}(i, \texttt{abstract}\ T\ m(\overline{T\ x})\texttt{;}) = \texttt{abstract}\ T\ i(\overline{T\ x})\texttt{;}$
  $\mathsf{named}(i,\ T\ m(\overline{T\ x})\,\{\,\texttt{return}\ e\texttt{;}\,\}) = T\ i(\overline{T\ x})\,\{\,\texttt{return}\ e\texttt{;}\,\}$
  $\mathsf{named}(c, c' = ce) = c = ce$

In (ALIAS), the operator adds a definition for field or method i^t, for "target", in class π, by duplicating that existing for i^s, for "source", in the same class. If i^s is a field, then the initialization expression is duplicated as well.

Notations $\mathsf{named}(n, d)$, $\mathsf{constr}(cv, \pi)$ and $cv \oplus_\pi fe$ are the obvious extension of the corresponding ones in FJIG$_1$. Formally:

- $\mathsf{constr}(cv, \pi) = k$ if $\mathsf{cBody}(cv, \pi) = \_\ \{\,k\ \_\,\}$

- $ch\ \{\, k\ \overline{d}\ (c = cv)\,\} \oplus_{.c.\pi} fe = ch\ \{\, k\ \overline{d}\ (c = cv \oplus_{\pi} fe)\,\}$
  $ch\ \{\, kh\,\{\overline{fe}\,\}\ \overline{d}\,\} \oplus_{.\Lambda} fe = ch\ \{\, kh\,\{\overline{fe}\ fe\,\}\ \overline{d}\,\}$

There are two variants of class alias operator, which take as second argument a class path and a path, respectively. Correspondingly there are two variants of redirect operator which take as last argument a class path and a path, respectively.

In (CLASS-ALIAS) and (CLASS-ALIAS-EXTERN), the operator adds a definition for nested class c in class π^t, by duplicating that existing for π^s and C^s, respectively, which must be a class value. If in position π^t there is already a nested class c, then the duplicated class value is summed with the existing class. Note that, analogously to rule (CLASS-PATH), the duplicated class value needs to be modified by "moving" all the occurrences of external class paths from the source *to* the target position. The notation $C^s[\mathsf{to}\ \pi^t]$, where s and t stand for "source" and "target", respectively, means "the shortest name for C^s in position π^t". Formally:

- $C^s[\mathsf{to}\ .\Lambda] = C^s$
  $c.\pi^s[\mathsf{to}\ .c.\pi^t] = \pi^s[\mathsf{to}\ \pi^t]$
  $C^s[\mathsf{to}\ .c.\pi^t] = (\mathtt{outer}.C^s)[\mathsf{to}\ \pi^t]$ if $C^s \neq c.\_$

To move to a descendant $c.\pi^t$ of the current position, if the source position is a descendant of child node c as well, that is, of form $c.\pi^s$, then we only have to move π^s to π^t (second clause). Otherwise, that is, if the source position C^s is either an ancestor of the current position or a descendant of a different child, this corresponds to move w.r.t. child node c, taken as current position, from $\mathtt{outer}.C^s$ to π^t (third clause).

Note that $C[\mathsf{to}\ \pi^t] = C'$ implies $C'[\mathsf{from}\ \pi^t] = C$; anyway, there can be many C'' such that $C''[\mathsf{from}\ \pi^t] = C$. For example $\mathtt{C.D.A}[\mathsf{to}\ \mathtt{C.D}] = \mathtt{A}$ and $\mathtt{A}[\mathsf{from}\ \mathtt{C.D}] = \mathtt{C.D.A}$ but also $\mathtt{outer.D.A}[\mathsf{from}\ \mathtt{C.D}] = \mathtt{C.D.A}$.

Also note that the source can be an outer class, that is, code to be duplicated can be taken from the outside, whereas of course the target, being code to be modified, cannot. The converse situation takes place for the class redirect operator, see below.

In (REDIRECT) the operator replaces references to an existing field or method i^s of class π by references to i^t. The declaration of i^s is removed, so i^s and i^t have to be different. We denote by $\mathsf{names}(cv, \pi)$ the set of names declared in nested class π of cv, and by $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$ the class value obtained from cv by replacing i^s with i^t in field accesses/method invocations whose receiver has static type π. Formally:

- $\mathsf{names}(cv, \pi) = \mathsf{dom}(\overline{d})$ if $\mathsf{cBody}(cv, \pi) = \_\ \{\_\ \overline{d}\,\}$

- $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t] = cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{.\Lambda}$
  $(c = cv)[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\pi'} = c = (cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\pi'.c})$
  $.[C]i[i^s \underset{[.\overline{c}']}{\rightsquigarrow} i^t]_{.\overline{c}} = \begin{cases} .[C]i^t & \text{if } C[\mathsf{from}\ \overline{c}] = \overline{c}' \text{ and } i = i^s \\ .[C]i & \text{otherwise} \end{cases}$

In (CLASS-REDIRECT) and (CLASS-REDIRECT-EXTERN) the operator replaces references to existing nested class c of class π^s by references to π^t and C^t, respectively. The declaration of c is removed, so $\pi^s.c$ and π^t have to be different. Redirect is only defined from a class that contains no nested classes, hence can be safely removed. It is trivial to define a derived operator which redirects a class with nested classes by performing a sequence of redirect applications. The predicate $\mathsf{noNested}(cv, \pi)$ holds iff class π has no nested classes, and $cv[\pi^s \rightsquigarrow C^t]$ is the class value obtained from cv by replacing π^s with C^t in type annotations and **`new`** expressions. Formally:

- $cv[\pi^s \rightsquigarrow C^t] = cv[\pi^s \rightsquigarrow C^t]_{.\Lambda}$
  $(c = cv)[\pi^s \rightsquigarrow C^t]_{\pi} = c = (ce[\pi^s \rightsquigarrow C^t]_{\pi.c})$
  $C[.\overline{c}' \rightsquigarrow C^t]_{.\overline{c}} = \begin{cases} C^t[\mathsf{to}\ .\overline{c}] & \text{if } C[\mathsf{from}\ \overline{c}] = \overline{c}' \\ C & \text{otherwise} \end{cases}$

- $\mathsf{noNested}(cv, \pi)$ if $\mathsf{cBody}(cv, \pi) = \_\ \{\ \_\ \overline{fd}\ \overline{md}\ \}$

Note that the notations $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$ and $cv[\pi^s \rightsquigarrow C^t]$ are defined by an accumulation parameter π corresponding to the nested class where the occurrence (of field or method name and path, respectively) is found. For simplicity, we omit all trivial propagation clauses.

Generalized inheritance relation Flattening reduces a class expression to a class value cv, that is, a basic class where every nested class is a class value as well. Termination is ensured by requiring the generalized inheritance relation $\xrightarrow[b]{\mathsf{inh}}$, defined in Figure 5.4, to be acyclic for well-formed basic classes.

Informally, $\pi \xrightarrow[b]{\mathsf{inh}} \pi'$ holds if the class expression defining (an enclosing class of) π has a subterm of shape either C, or $ce\,$**`[alias`**$\,C\,$**`to _]`**, with **`outer`**$.C$ denoting π' from position π. Indeed, in both cases, in order to reduce the class expression π in b, we need a class value in position π'.

Reduction An FJIG$_\star$ application consists of a *main expression* e which is executed in the environment σ. Formally, the reduction arrow has form $e_1 \underset{\sigma}{\rightarrow} e_2$. This models the fact that a *main* method could belong to an arbitrarily nested class. Reduction rules are straightforward, and formally defined in Figure 5.5.

$\pi_1 \xrightarrow[b]{\mathsf{inh}} \pi_2$

$$\text{(DIRECT-I)}\ \frac{}{.\overline{c}_1.\_ \xrightarrow[b]{\mathsf{inh}} .\overline{c}_2}\ \begin{array}{l} b(.\overline{c}_1) = \mathcal{CE}[\![C]\!] \text{ or } b(.\overline{c}_1) = \mathcal{CE}[\![ce\ [\,\texttt{alias}\ C\ \texttt{to}\ \_\,]]\!] \\ \texttt{outer}.C[\mathsf{from}\ \overline{c}_1] = \overline{c}_2 \end{array}$$

$$\text{(TRANS-I)}\ \frac{\pi_1 \xrightarrow[b]{\mathsf{inh}} \pi_2 \quad \pi_2 \xrightarrow[b]{\mathsf{inh}} \pi_3}{\pi_1 \xrightarrow[b]{\mathsf{inh}} \pi_3}$$

Figure 5.4: FJIG$_\star$ generalized inheritance relation

$e_1 \xrightarrow[\sigma]{} e_2$

$$\text{(FIELD-ACCESS)}\ \frac{}{C\ (\overline{fv})\ .f \xrightarrow[\sigma]{} v}\ \overline{fv}(f) = v$$

$$\text{(INVK)}\ \frac{}{C\ (\overline{fv})\ .m\ (\overline{v}) \xrightarrow[\sigma]{} e[\mathsf{from}\ C][\overline{v}/\overline{x}][^{C\ (\overline{fv})}/_{\texttt{this}}]}\ \mathsf{mBody}(\sigma, C, m) = \langle \overline{x}, e \rangle$$

$$\text{(OBJ-CREATION)}\ \frac{}{\texttt{new}\ C\ (\overline{v}) \xrightarrow[\sigma]{} C\ (\overline{fe}[\mathsf{from}\ \pi][\overline{v}/\overline{x}])}\ \begin{array}{l} \mathsf{kBody}(\sigma, C) = \langle \overline{x}, \overline{fe} \rangle \\ \mathsf{nonAbs}(\sigma, C) \end{array}$$

Figure 5.5: FJIG$_\star$ reduction rules

Δ	::=	$\overline{ct}$	class type environment
ct	::=	$[\mu \mid \overline{C} \mid kt \mid \overline{dt}]$	class type
kt	::=	$\overline{T}$	constructor type
dt	::=	$i{:}\mu\, mt \mid c{:}ct$	declaration type
mt	::=	$T \mid \overline{T} \to T$	field or method type
Γ	::=	$\overline{x{:}T}$	parameter type environment

Figure 5.6: FJIG$_\star$ types and type environments

The only significant difference w.r.t. FJIG$_0$ is that, in rule (INVK), expression e is found in position C, so class paths inside e must be moved to denote the same class value in the current position, and analogously in rule (OBJ-CREATION). We extend to environments σ and class paths C, and repeat here for clarity, the following notations already used in FJIG$_0$ on programs and class names:
if $\mathsf{cBody}(\sigma, C) = \mu$ **implements** _ { $k\ \overline{d}$ }

- $\mathsf{mBody}(\sigma, C, m)$ returns parameters and body of method m of class C in σ, formally: $\mathsf{mBody}(\sigma, C, m) = \langle x_1 \ldots x_n, e\rangle$ if $\overline{d}(m) = T\ m(T_1\ x_1 \ldots T_n\ x_n)${ **return** e**;** }
- $\mathsf{kBody}(\sigma, C)$ returns parameters and body of constructor of class C in σ, formally: $\mathsf{kBody}(\sigma, C) = \langle x_1 \ldots x_n, \overline{fe}\rangle$ if $k =$ **constructor** ($T_1\ x_1 \ldots T_n\ x_n$) { $\overline{fe}$ }
- $\mathsf{nonAbs}(\sigma, C)$ holds if class C in σ is non abstract, formally iff $\mu \neq$ **abstract**.

5.4 Type system

Types and type environments are defined in Figure 5.6.

Analogously to what happens for flattening, all typing judgements have on the left a class type environment, which is a stack of class types $ct_0 \ldots ct_n$, where ct_0 is the type of the directly enclosing class, ct_1 is the type of the **outer** class, and so on. A class type is a tuple consisting of the kind (abstract or non abstract), the supertypes, the constructor type, and a map from field/method names to their kinds (abstract or non abstract) and types, and from class names to class types.

Typing rules for environments, basic classes and well-formedness of class types are given in Figure 5.7. They are straightforward. The typing judgment for environments is used in Theorem 31 and Theorem 37.

$\vdash \sigma : \Delta$

$$\text{(ENV-T)}\ \frac{ct_{i+1} \ldots ct_n \vdash b_i : ct_i \quad \forall i \in 0..n}{\vdash \sigma : \Delta}\ \begin{array}{l}\sigma = b_0 \ldots b_n \\ \Delta = ct_0 \ldots ct_n\end{array}$$

$\Delta \vdash ce : ct$

$$\text{(BASIC-T)}\ \frac{ct \cdot \Delta \vdash k : kt \quad ct \cdot \Delta \vdash \overline{d} : \overline{dt} \quad ct \cdot \Delta \vdash ct}{\Delta \vdash \mu\ \texttt{implements}\ \overline{C}\ \{\, k\ \overline{d}\,\} : ct}\ \begin{array}{l}\mathsf{exists}(ct \cdot \Delta, \overline{C}) \\ ct = [\mu \mid \overline{C} \mid kt \mid \overline{dt}]\end{array}$$

$\Delta; \Gamma \vdash k : kt$

$$\text{(CONS-T)}\ \frac{\begin{array}{c}\Delta; x_1{:}T'_1, \ldots, x_n{:}T'_n \vdash e_i : T''_i \quad \forall i \in 1..k \\ \Delta \vdash T''_i \leq T_i \quad \forall i \in 1..k\end{array}}{\Delta \vdash kh\,\{\,\texttt{this.}f_1 = e_1\texttt{;} \ldots \texttt{this.}f_k = e_k\texttt{;}\,\} : T'_1 \ldots T'_n}\ \begin{array}{l}\mathsf{exists}(\Delta, T'_i) \quad \forall i \in 1..n \\ kh = \texttt{constructor}\,(T'_1\ x_1, \ldots, T'_n\ x_n) \\ \mathsf{defFields}(\Delta, \Lambda) = f_1{:}T_1, \ldots, f_k{:}T_k\end{array}$$

$\Delta \vdash d : dt$

$$\text{(FIELD-T)}\ \frac{}{\Delta \vdash (\mu\ T\ f\texttt{;}) : (f{:}\mu\ T)}\ \mathsf{exists}(\Delta, T)$$

$$\text{(ABS-METHOD-T)}\ \frac{}{\Delta \vdash (\texttt{abstract}\ mh\texttt{;}) : (m{:}\texttt{abstract}\ T_1 \ldots T_n \to T_0)}\ \begin{array}{l}mh = T_0\, m(T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n\end{array}$$

$$\text{(METHOD-T)}\ \frac{\Delta; \texttt{this}{:}\Lambda, x_1{:}T_1, \ldots, x_n{:}T_n \vdash e : T \quad \Delta \vdash T \leq T_0}{\Delta \vdash mh\,\{\,\texttt{return}\ e\texttt{;}\,\} : (m{:}T_1 \ldots T_n \to T_0)}\ \begin{array}{l}mh = T_0\, m(T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n\end{array}$$

$$\text{(CLASS-T)}\ \frac{\Delta \vdash ce : ct}{\Delta \vdash (c = ce) : (c{:}ct)}$$

$\Delta \vdash ct$

$$\text{(WF-CLASS-TYPE)}\ \frac{\begin{array}{c}ct \cdot \Delta \vdash ct_j \quad \forall j \in 1..k \\ ct \leq (\mathsf{cType}(ct \cdot \Delta, C_i)[\mathsf{from}\ C_i]) \quad \forall i \in 1..n\end{array}}{\Delta \vdash ct}\ \begin{array}{l}\overline{ct} = [\_ \mid C_1 \ldots C_n \mid \_ \mid \overline{dt}] \\ \overline{dt} = \overline{fd}\ \overline{md}\ c_1{:}ct_1 \ldots c_k{:}ct_k\end{array}$$

Figure 5.7: FJIG$_\star$ typing rules for environments, basic classes and well-formed class types

Note that in (BASIC-T) the constructor body, the method bodies and the nested classes are type-checked in the class type environment obtained by pushing the type of the basic class on the stack of class types. The judgement $\Delta \vdash ct$, see (WF-CLASS-TYPE), means that ct is well-formed w.r.t. Δ, that is, the subtyping relation induced by the supertypes in (all nested class types) in ct can be safely assumed. Note that we have to move $\mathsf{cType}(ct \cdot \Delta, C_i)$ from C_i to properly check structural subtyping, defined by rule (STRUCTURAL-S) in Figure 5.9.

We use the following notations:

- $ct \cdot \Delta$ is analogous to $b \cdot \sigma$
- $\mathsf{exists}(\Delta, C)$ holds if C denotes an existing class in Δ, formally if $\mathsf{cType}(\Delta, C)$ is defined.
- $\mathsf{defFields}(\Delta, C)$ are the declaration types corresponding to non abstract fields of class C in Δ, formally:
  $\mathsf{defFields}(\Delta, C) = \overline{f{:}T}$
  if $\mathsf{cType}(\Delta, C) = [\_ \mid \_ \mid \_ \mid \overline{f{:}T}\ \overline{f{:}\mathbf{abstract}\ T}\ \overline{m{:}\mu\ \overline{T} \to T}\ \overline{c{:}ct}]$
- $\mathsf{cType}(\Delta, C)$ and $ct[\mathsf{from}\ C]$ are analogous to $\mathsf{cBody}(\sigma, C)$ and $cv[\mathsf{from}\ C]$, respectively, but work over class types.

Typing rules for composition operators are given in Figure 5.8. They are similar to corresponding flattening rules.

In some cases, it is necessary to check for well-formedness of the resulting type, since the application of the operator can break the subtyping relation, notably: sum, alias and class alias can add a field or method to a nested class declared as supertype, redirect can remove a field or method needed to implement a supertype, and class redirect can replace a nested class declared as supertype with another one with more fields or methods.

Besides the checks performed in rules (CLASS-REDIRECT1) and (CLASS-REDIRECT2) , in (CLASS-REDIRECT1-T) and (CLASS-REDIRECT2-T) additional checks need to be performed to ensure that the class C^t can safely replace $\pi^s.c$. Here $C^{t'}$ is the type C^t as it is seen into $\pi^s.c$. That is: all the declared supertype of $\pi^s.c$ are supertype of $C^{t'}$, structural subtyping between the class type of $\pi^s.c$ and $C^{t'}$ holds, and if the source is instantiable, then the target have to be instantiable too, and the two constructor type have be the same.

We use the following notations:

- $ct_1 \oplus ct_2$, $ct \ominus_\pi n$ and $ct \oplus_\pi dt$ are analogous to $cv_1 \oplus cv_2$, $cv \ominus_\pi n$ and $cv \oplus_\pi d$, respectively, but work over class types.
- $\mathsf{decType}(\Delta, C, n)$, $\mathsf{abs}(dt)$ and $\mathsf{named}(n, dt)$ are analogous to $\mathsf{dec}(cv, \pi, n)$, $\mathsf{abs}(d)$ and $\mathsf{named}(n, d)$, respectively, but work over declaration types.

$\Delta \vdash ce : ct$

$$\text{(CLASS-PATH-T)}\ \frac{}{\Delta \vdash C : ct}\ ct = \mathsf{cType}(\Delta, C)[\mathsf{from}\ C]$$

$$\text{(SUM-T)}\ \frac{\Delta \vdash ce_1 : ct_1 \quad \Delta \vdash ce_2 : ct_2 \quad \Delta \vdash ct_1 \oplus ct_2}{\Delta \vdash ce_1\,[+]\,ce_2 : ct_1 \oplus ct_2}$$

$$\text{(RESTRICT-T)}\ \frac{\Delta \vdash ce : ct}{\Delta \vdash ce\,[\,\mathbf{restrict}\ i\ \mathbf{in}\ \pi\,] : ct \ominus_\pi i \oplus_\pi \mathsf{abs}(dt)}\ dt = \mathsf{decType}(ct, \pi, i)$$

$$\text{(ALIAS-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct \oplus_\pi \mathsf{named}(i^t, dt)}{\Delta \vdash ce\,[\,\mathbf{alias}\ i^s\ \mathbf{to}\ i^t\ \mathbf{in}\ \pi\,] : ct \oplus_\pi \mathsf{named}(i^t, dt)}\ dt = \mathsf{decType}(ct, \pi, i^s)$$

$$\text{(CLASS-ALIAS1-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct \oplus_{\pi^t} (c' {:}\mu\ ct')}{\Delta \vdash ce\,[\,\mathbf{alias}.\overline{c}\ \mathbf{to}\ \pi^t.c'\,] : ct \oplus_{\pi^t} (c' {:}\mu\ ct')}\ ct' = \mathsf{cType}(ct, \pi^s)[\mathsf{from}\ \overline{c}[\mathsf{to}\ \pi^t]]$$

$$\text{(CLASS-ALIAS2-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct \oplus_{\pi^t} (c' {:}\mu\ ct')}{\Delta \vdash ce\,[\,\mathbf{alias}\ C^s\ \mathbf{to}\ \pi^t.c'\,] : ct \oplus_{\pi^t} (c' {:}\mu\ ct')}\ ct' = \mathsf{cType}(\Delta, C^s)[\mathsf{from}\ \mathbf{outer}^{|\pi^t|}.C^s]$$

$$\text{(REDIRECT-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct \ominus_\pi i^s}{\Delta \vdash ce\,[\,\mathbf{redirect}\ i^s\ \mathbf{of}\ \pi\ \mathbf{to}\ i^t\,] : ct \ominus_\pi i^s}\ \begin{array}{l}\mathsf{mType}(ct, \pi, i^s) = \mathsf{mType}(ct, \pi, i^t)\\ i^s \neq i^t\end{array}$$

$$\text{(CLASS-REDIRECT1-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct'}{\Delta \vdash ce\,[\,\mathbf{redirect}\ \pi^s\ \mathbf{to}.\overline{c}\,] : ct'}\ \begin{array}{l}\pi^s = \pi'.c\\ ct' = (ct \ominus_{\pi'} c)[\pi^s \rightsquigarrow \overline{c}]\\ \pi^s \neq .\overline{c}\\ ct^s = \mathsf{cType}(ct, \pi^s)[\pi^s \rightsquigarrow \overline{c}]\\ \mathsf{noNested}(ct, \pi^s)\\ ct[\mathsf{in}\ \pi^s] \vdash \overline{c}[\mathsf{to}\ \pi^s] \lhd ct^s\end{array}$$

$$\text{(CLASS-REDIRECT2-T)}\ \frac{\Delta \vdash ce : ct \quad \Delta \vdash ct'}{\Delta \vdash ce\,[\,\mathbf{redirect}\ \pi^s\ \mathbf{to}\ C^t\,] : ct'}\ \begin{array}{l}\pi^s = \pi'.c\\ ct' = (ct \ominus_{\pi'} c)[\pi^s \rightsquigarrow \mathbf{outer}.C^t]\\ ct^s = \mathsf{cType}(ct, \pi^s)[\pi^s \rightsquigarrow \mathbf{outer}.C^t]\\ \mathsf{noNested}(ct, \pi^s)\\ \Delta[\mathsf{in}\ \pi^s] \vdash \mathbf{outer}^{|\pi^s|}.C^t \lhd ct^s\end{array}$$

Figure 5.8: FJIG$_\star$ typing rules for composition operators

- $\mathsf{mType}(\Delta, C, i)$ is the type of field or method i in class C in Δ.

- $\Delta \vdash T \leq T'$ holds if T is a subtype of T' in Δ, and $ct \leq ct'$ holds if ct is a (structural) subtype of ct'. Subtyping is straightforward and defined in Figure 5.9.

- $ct[\pi^s \rightsquigarrow C^t]$, $\mathsf{kType}(\Delta, C)$, $\mathsf{noNested}(ct, \pi)$ and and $\mathsf{nonAbs}(\Delta, C)$
  are analogous to $cv[\pi^s \rightsquigarrow C^t]$, $\mathsf{constr}(\sigma, C)$, $\mathsf{noNested}(cv, \pi)$ and $\mathsf{nonAbs}(\sigma, C)$.

- $\mathsf{impl}(\Delta, C)$ denotes the declared supertypes of class C in type environment Δ. Formally:
  $\mathsf{impl}(\Delta, C) = \overline{C}$ if $\mathsf{cType}(\Delta, C) = [\_ \mid \overline{C} \mid \_ \mid \_]$,

- $\Delta \vdash C^t \lhd ct^s$ holds if C^t in Δ denotes a type which can be used by clients in at least all the way ct^s can be used; That is a class whose type is ct^s can be redirected to class C^t. Formally:
  $\Delta \vdash C^t \lhd [\mu \mid \overline{c} \mid kt \mid \overline{dt}]$ holds iff
  $\Delta \vdash C^t \leq C \quad \forall C \in \overline{c}$
  $\mathsf{cType}(\Delta, C^t)[\mathsf{from}\ C^t] \leq [\mu \mid \overline{c} \mid kt \mid \overline{dt}]$
  $\mu \neq \mathbf{abstract}$ implies $\mathsf{nonAbs}(\Delta, C^t)$ and $\mathsf{kType}(\Delta, C^t)[\mathsf{from}\ C^t] = kt$

- $\Delta[\mathsf{in}\ C]$ is the class type environment which is seen into position C w.r.t. the current position, where the type environment is Δ. Formally:
  $(ct_1 \ldots ct_n \cdot \Delta)[\mathsf{in}\ \mathbf{outer}^n.\overline{c}] = \Delta[\mathsf{in}\ .\overline{c}]$
  $(ct \cdot \Delta)[\mathsf{in}\ .c.\pi] = (ct(c) \cdot ct \cdot \Delta)[\mathsf{in}\ \pi]$
  $\Delta[\mathsf{in}\ .\Lambda] = \Delta$

Typing rules for expressions are given in Figure 5.10, and are analogous to the ones of FJIG$_0$. Note that in order to use type T as declared in position C, we need to use $T[\mathsf{from}\ C]$.

5.5 Results

The type system is sound w.r.t. flattening, that is, a well-typed FJIG$_\star$ class expression ce always reduces in some steps to a well-typed class value cv, as stated by the following theorem.

Theorem 22 (FJIG$_\star$ soundness w.r.t. flattening)**.** *If* $\Lambda \vdash ce : ct$*, then* $ce \rightarrow^\star cv$*, and* $\Lambda \vdash cv : ct$.

This soundness theorem follows from termination of flattening, progress, and subject reduction theorems below. Note that progress relies on the assumption that the generalized inheritance relation $\xrightarrow[b]{\mathsf{inh}}$ is acyclic.

$$\text{(DIRECT-S)}\ \frac{}{\Delta \vdash C \leq (C_i[\mathsf{from}\ C])}\ \mathsf{impl}(\Delta, C) = C_1 \ldots C_n$$

$\boxed{\Delta \vdash T_1 \leq T_2}$

$$\text{(REFL-S)}\ \frac{}{\_ \vdash T \leq T} \qquad \text{(TRANS-S)}\ \frac{\Delta \vdash T_1 \leq T_2 \quad \Delta \vdash T_2 \leq T_3}{\Delta \vdash T_1 \leq T_3}$$

$\boxed{ct_1 \leq ct_2}$

$$\text{(STRUCTURAL-S)}\ \frac{}{ct_1 \leq ct_2}\ \begin{array}{l} ct_1 = [\_ \mid \_ \mid \_ \mid (i_1{:}\_\ mt_1) \ldots (i_n{:}\_\ mt_n)\overline{c{:}ct}] \\ ct_2 = [\_ \mid \_ \mid \_ \mid (i_1{:}\_\ mt_1) \ldots (i_n{:}\_\ mt_n)\overline{dt}] \end{array}$$

Figure 5.9: FJIG$_\star$ subtyping rules

In order to prove termination of flattening, we formally define the dimension of a class expression:

$\dim(\_\ \{\ \_\overline{fd}\ \overline{md}\ \overline{cd}\ \}) = \dim(\overline{cd})$
$\dim(\_ = ce) = \dim(ce)$
$\dim(ce_1\ [+]\ ce_2) = \dim(ce_1) + \dim(ce_2) + 1$
$\dim(ce\ [\texttt{restrict\_in\_}]) = \dim(ce) + 1$
$\ldots$
$\dim(C) = 1$

Lemma 23. *If* $ce_1 \underset{\sigma}{\rightarrow} ce_2$ *then* $\dim(ce_1) > \dim(ce_2)$.

Proof. By cases on the flattening rules. We show only one case:

Case $\text{(SUM)}\ \dfrac{}{cv_1\ [+]\ cv_2 \underset{\sigma}{\rightarrow} cv_1 \oplus cv_2}$

$\dim(b_1\ [+]\ b_2) = 1$ and $\dim(b_1 \oplus b_2) = 0$.

The other cases are analogous. □

Theorem 24 (FJIG$_\star$ termination of flattening)**.** *If* $ce_1 \rightarrow ce_2$ *then* $\dim(ce_1) > \dim(ce_2)$.

$\Delta; \Gamma \vdash e : T$

$$\text{(VAR-T)}\ \frac{}{\_; \Gamma \vdash x : T}\ \Gamma(x) = T \qquad \text{(FIELD-ACCESS-T)}\ \frac{\Delta; \Gamma \vdash e : C}{\Delta; \Gamma \vdash e.f : T[\mathsf{from}\ C]}\ \mathsf{mType}(\Delta, C, f) = T$$

$$\text{(INVK-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash e : C \\ \Delta; \Gamma \vdash \overline{e} : \overline{T} \\ \Delta \vdash \overline{T} \leq (\overline{T}'[\mathsf{from}\ C])\end{array}}{\Delta; \Gamma \vdash e.m\,(\overline{e}) : T[\mathsf{from}\ C]}\ \mathsf{mType}(\Delta, C, m) = \overline{T}' \to T$$

$$\text{(NEW-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash \overline{e} : \overline{T} \\ \Delta \vdash \overline{T} \leq (\overline{T}'[\mathsf{from}\ C])\end{array}}{\Delta; \Gamma \vdash \mathbf{new}\ C\,(\overline{e}) : C}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, C) \\ \mathsf{kType}(\Delta, C) = \overline{T}'\end{array}$$

$$\text{(OBJ-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash e_i : T_i \quad \forall i \in 1..n \\ \Delta \vdash T_i \leq (T_i'[\mathsf{from}\ C]) \quad \forall i \in 1..n\end{array}}{\Delta; \Gamma \vdash C\,(\overline{fe}) : C}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, C) \\ \overline{fe} = \mathbf{this}.f_1 = e_1\mathbf{;}\ \ldots\ \mathbf{this}.f_n = e_n\mathbf{;} \\ \mathsf{defFields}(\Delta, C) = f_1{:}\epsilon\ T_1', \ldots, f_n{:}\epsilon\ T_n'\end{array}$$

Figure 5.10: FJIG$_\star$ typing rules for expressions

Proof. Only rule $\text{(CTX)}\dfrac{ce_1 \xrightarrow[\sigma]{} ce_2}{\mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] \to \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!]}\ \sigma = \mathsf{enclosing}(ce_1, \mathcal{CE}^{\mathrm{f}})$

reduces a top level class expression. By Lemma 23, we have that $\dim(ce_1) > \dim(ce_2)$. We can conclude by definition of $\dim(ce_1)$. □

Soundness of the type system w.r.t. flattening is expressed by the following two theorems. Note that soundness relies on well-formedness of class expressions, that is, that the relation $\pi \xrightarrow[ce]{\mathsf{inh}} \pi'$ is acyclic.

In the next lemma we use the following definition of redex.

$$\begin{aligned} r \quad ::= \quad & cv_1\,\texttt{[+]}\,cv_2 \mid cv\,\texttt{[}\,\textbf{restrict}\,i\,\textbf{in}\,\pi\,\texttt{]} \mid cv\,\texttt{[}\,\textbf{alias}\,i^s\,\textbf{to}\,i^t\,\textbf{in}\,\pi\,\texttt{]} \\ & \mid cv\,\texttt{[}\,\textbf{redirect}\,i^s\,\textbf{of}\,\pi\,\textbf{to}\,i^t\,\texttt{]} \mid cv\,\texttt{[}\,\textbf{alias}\,\pi^s\,\textbf{to}\,\pi^t\,\texttt{]} \mid cv\,\texttt{[}\,\textbf{redirect}\,\pi^s\,\textbf{to}\,\pi^t\,\texttt{]} \\ & \mid cv\,\texttt{[}\,\textbf{alias}\,C^s\,\textbf{to}\,\pi^t\,\texttt{]} \mid cv\,\texttt{[}\,\textbf{redirect}\,\pi^s\,\textbf{to}\,C^t\,\texttt{]} \end{aligned}$$

Lemma 25. *If* $\vdash \mathcal{CE}^{\mathrm{f}}[\![ce]\!] : ct$ *then* $\sigma = \mathsf{enclosing}(ce, \mathcal{CE}^{\mathrm{f}})$, $\vdash \sigma : \Delta$ *and* $\Delta \vdash ce : ct$.

Lemma 26 (FJIG$_\star$ progress w.r.t. $ce_1 \xrightarrow[p]{} ce_2$)**.** *If* $\Delta \vdash r : ct$ *then* $r \xrightarrow[\_]{} cv$, *for some basic class* cv.

Proof. By cases. We show only one case:

Case $r = cv_1\,\texttt{[+]}\,cv_2$

typed by $\text{(SUM-T)}\dfrac{\Delta \vdash cv_1 : ct_1 \quad \Delta \vdash cv_2 : ct_2 \quad \Delta \vdash ct_1 \oplus ct_2}{\Delta \vdash cv_1\,\texttt{[+]}\,cv_2 : ct_1 \oplus ct_2}$

We get the thesis by application of rule $\text{(SUM)}\dfrac{}{cv_1\,\texttt{[+]}\,cv_2 \xrightarrow[\sigma]{} cv_1 \oplus cv_2}$, since $cv_1 \oplus cv_2$ is well-defined. Indeed, by (BASIC-T) and $ct_1 \oplus ct_2 = \_$, we know that the constructor of cv_1 is equal to the constructor of cv_2, and by definition of $\overline{dt}_1 \oplus \overline{dt}_2$ we know that $\overline{d}_1 \oplus \overline{d}_2 = \_$ where $\overline{d}_i$ are the declarations of cv_i and $\overline{dt}_i$ are the declaration types of ct_i. □

Theorem 27 (FJIG$_\star$ progress w.r.t. flattening)**.** *If* $\emptyset \vdash ce : ct$ *then* ce *is flat or* $ce \to \_$.

Proof. If ce is not flat, then we have two cases:

Case $ce = \mathcal{CE}^{\mathrm{f}}[\![r]\!]$

By Lemma 25 and applying rule $\text{(CTX)}\dfrac{ce_1 \xrightarrow[\sigma]{} ce_2}{\mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] \to \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!]}\ \sigma = \mathsf{enclosing}(ce_1, \mathcal{CE}^{\mathrm{f}})$.

Otherwise Since for all $b\ \pi \xrightarrow[b]{\mathsf{inh}} \pi'$ is acyclic then $ce = \mathcal{CE}^{\mathrm{f}}[\![C]\!]$ with $\mathsf{cBody}(\sigma, C) = cv$, $\sigma = \mathsf{enclosing}(ce, \mathcal{CE}^{\mathrm{f}})$. We can conclude by applying rules (CTX) and

$$\text{(CLASS-PATH)}\frac{}{C \underset{\sigma}{\rightarrow} cv}\ cv = \mathsf{cBody}(\sigma, C)[\mathsf{from}\ C]\ .$$ □

Lemma 28 (FJIG$_\star$ substitution lemma)**.** *If* $\Delta \vdash ce_1 : ct$, $\Delta \vdash ce_2 : ct$, *and* $\Delta \vdash \mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] : ct'$, *then* $\Delta \vdash \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!] : ct'$.

Proof. By straightforward structural induction on $\mathcal{CE}^{\mathrm{f}}$. □

Lemma 29. *If* $\vdash \sigma : \Delta$, $\Delta \vdash ce : ct$ *and* $ce_1 \underset{\sigma}{\rightarrow} ce_2$ *then* $\Delta \vdash ce_2 : ct$.

Proof. By cases on the flattening rules. We show only one case:

Case $\text{(SUM)}\dfrac{}{cv_1\,[+]\,cv_2 \underset{\sigma}{\rightarrow} cv_1 \oplus cv_2}$

typed by $\text{(SUM-T)}\dfrac{\Delta \vdash cv_1 : ct_1 \quad \Delta \vdash cv_2 : ct_2 \quad \Delta \vdash ct_1 \oplus ct_2}{\Delta \vdash cv_1\,[+]\,cv_2 : ct_1 \oplus ct_2}$

We get the thesis by (BASIC-T), since $ct_1 \oplus ct_2$ is analogous to $cv_1 \oplus cv_2$. □

Theorem 30 (FJIG$_\star$ subject reduction w.r.t. flattening)**.** *If* $\vdash ce_1 : ct$ *and* $ce_1 \rightarrow ce_2$ *then* $\vdash ce_2 : ct$.

Proof. Only rule $\text{(CTX)}\dfrac{ce_1 \underset{\sigma}{\rightarrow} ce_2}{\mathcal{CE}^{\mathrm{f}}[\![ce_1]\!] \rightarrow \mathcal{CE}^{\mathrm{f}}[\![ce_2]\!]}\ \sigma = \mathsf{enclosing}(ce_1, \mathcal{CE}^{\mathrm{f}})$

reduces a top level class expression. We get the thesis by Lemma 29, Lemma 28 and $\vdash ce_1 : ct$. □

In order to express soundness of the type system (w.r.t. expression reduction arrow), we introduce the notion of *complete* class type, that is, a class type which does not contain abstract fields or methods, and whose non abstract nested classes have, recursively, a complete class type. Formally:

$\mathsf{isComplete}([\mu \mid \_ \mid \_ \mid \overline{dt}])$ if $\overline{dt}(i) = i{:}\mathbf{abstract}\ \_$ implies $\mu = \mathbf{abstract}$, and $\overline{dt}(c) = c{:}ct$ implies $\mathsf{isComplete}(ct)$.

Once a class value cv has been obtained, an expression e, placed in a position π, can be reduced in the environment of $cv[\mathsf{in}\ \pi]$, where

- $\sigma[\mathsf{in}\ \pi]$ is the run time environment which is seen into position π w.r.t. a class value cv. is defined in this way:
  $(cv \cdot \sigma)[\mathsf{in}\ .c.\pi] = (cv(c) \cdot cv \cdot \sigma)[\mathsf{in}\ \pi]$
  $\sigma[\mathsf{in}\ .\Lambda] = \sigma$

A class expression with an incomplete class type can be safely used as a library, but not as an executable application.

Theorem 31 (FJIG$_\star$ progress)**.** *If* $\emptyset \vdash cv : ct$, $\mathsf{isComplete}(ct)$, $\Delta = ct[\mathsf{in}\ \pi]$ *and* $\Delta; \emptyset \vdash e : T$, *then either* e *is a value or* $e \underset{\sigma}{\rightarrow} \_$, *where* $\sigma = cv[\mathsf{in}\ \pi]$.

Proof. By induction over the typing rules, similar to the corresponding proof in FJIG$_0$. We show only one case:

Case $\text{(INVK-T)}\dfrac{\begin{array}{c}\Delta;\Gamma \vdash e : C \\ \Delta;\Gamma \vdash \overline{e} : \overline{T} \\ \Delta \vdash \overline{T} \leq (\overline{T}'[\mathsf{from}\ C])\end{array}}{\Delta;\Gamma \vdash e.m\,(\overline{e})\ : T[\mathsf{from}\ C]}\ \mathsf{mType}(\Delta, C, m) = \overline{T}' \rightarrow T$

Assume $\overline{e} = e_1 \ldots e_n$. From the premise by the inductive hypothesis we have two cases:

- $e \underset{\sigma}{\rightarrow} \_$, or $e_i \underset{\sigma}{\rightarrow} \_$ for some $i \in 1..n$. We can apply (CTX) to show that the whole term reduces.
- e is a value of form $C\,(\overline{fv})$, and $\overline{e} = \overline{v}$. Hence, we have applied rule

  $$\text{(OBJ-T)}\frac{\begin{array}{c}\Delta;\Gamma \vdash v_i : T_i \quad \forall i \in 1..n \\ \Delta \vdash T_i \leq (T_i'[\mathsf{from}\ C]) \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash C\,(\overline{fv})\ : C}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, C) \\ \overline{fv} = \mathtt{this}.f_1 = v_1\mathtt{;}\ \ldots\ \mathtt{this}.f_n = v_n\mathtt{;} \\ \mathsf{defFields}(\Delta, C) = f_1{:}\epsilon\ T_1', \ldots, f_n{:}\epsilon\ T_n'\end{array}$$

  We can apply rule

  $$\text{(INVK)}\frac{}{C\,(\overline{fv})\,.m\,(\overline{v})\ \underset{\sigma}{\rightarrow}\ e[\mathsf{from}\ C][{}^{\overline{v}}/_{\overline{x}}][{}^{C\,(\overline{fv})}/_{\mathtt{this}}]}\ \mathsf{mBody}(\sigma, C, m) = \langle \overline{x}, e \rangle$$

  Indeed:
  - $\mathsf{mBody}(p, C, m) = \langle \overline{x}, e \rangle$ holds by $\mathsf{mType}(\Delta, C, m) = \overline{T} \rightarrow T$ and $\mathsf{nonAbs}(\Delta, C)$,
  - $\mid \overline{e} \mid = \mid \overline{x} \mid$ holds, since by second premise we have $\mid \overline{e} \mid = \mid \overline{T} \mid$, by $\Delta \vdash \overline{T} \leq \overline{T}'$ we have $\mid \overline{T} \mid = \mid \overline{T}' \mid$, and by (BASIC-T), (METHOD-T) and definition of mType we have $\mid \overline{x} \mid = \mid \overline{T}' \mid$. □

In order to prove the subject reduction property w.r.t. expression reduction arrow we need some interesting lemmas.

In the following lemma, we use the notation π^n to denote a path of length n, and π^{n-j}, well-formed only if $0 \leq j \leq n$, to denote the path composed by the first $n - j$ elements of π^n.

Lemma 32. $T[\mathsf{from}\ C_1][\mathsf{from}\ C_2] = T[\mathsf{from}\ C_1[\mathsf{from}\ C_2]]$

Proof. By definition of _[from _] we have that
$\mathtt{outer}^n.\pi^k[\mathsf{from}\ \mathtt{outer}^{n_1}.\pi^{k_1}][\mathsf{from}\ \mathtt{outer}^{n_2}.\pi^{k_2}] =$
$(\mathtt{outer}^{n_1}.(\pi^{k_1} \setminus n).\pi^k)[\mathsf{from}\ \mathtt{outer}^{n_2}.\pi^{k_2}]$
and
$\mathtt{outer}^n.\pi^k[\mathsf{from}\ \mathtt{outer}^{n_1}.\pi^{k_1}[\mathsf{from}\ \mathtt{outer}^{n_2}.\pi^{k_2}]] =$
$\mathtt{outer}^n.\pi^k[\mathsf{from}\ \mathtt{outer}^{n_2}.(\pi^{k_2} \setminus n_1).\pi^{k_1}]$.
Again by definition of _[from _], we can show that both start with $\mathtt{outer}^{n_2}$ and end with π^k, so we have to show only that:
$(\mathtt{outer}^{n_1}.(\pi^{k_1} \setminus n))[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ (\pi^{k_2} \setminus n_1).\pi^{k_1}]$
By cases:

$n \geq k_1$ **and** $n_1 \geq k_2$
$\mathtt{outer}^{n_1+n-k_1}[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ \mathtt{outer}^{n_1-k_2}.\pi^{k_1}]$
$\pi^{k_2} \setminus n_1 + n - k_1 = \mathtt{outer}^{n_1-k_2}.(\pi^{k_1} \setminus n)$
$\mathtt{outer}^{n_1+n-k_1-k_2} = \mathtt{outer}^{n_1-k_2+n-k_1}$

$n < k_1$ **and** $n_1 \geq k_2$
$(\mathtt{outer}^{n_1}.\pi^{k_1-n})[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ \mathtt{outer}^{n_1-k_2}.\pi^{k_1}]$
$(\pi^{k_2} \setminus n_1).\pi^{k_1-n} = \mathtt{outer}^{n_1-k_2}.(\pi^{k_1} \setminus n)$
$\mathtt{outer}^{n_1-k_2}.\pi^{k_1-n} = \mathtt{outer}^{n_1-k_2}.\pi^{k_1-n}$

$n \geq k_1$ **and** $n_1 < k_2$
$\mathtt{outer}^{n_1+n-k_1}[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ \pi^{k_2-n_1}.\pi^{k_1}]$
$\pi^{k_2} \setminus n_1 + n - k_1 = (\pi^{k_2-n_1}.\pi^{k_1}) \setminus n$
$\pi^{k_2-n_1} \setminus n - k_1 = \pi^{k_2-n_1} \setminus n - k_1$

$n < k_1$ **and** $n_1 < k_2$
$\mathtt{outer}^{n_1}.\pi^{k_1-n}[\mathsf{from}\ \pi^{k_2}] = \mathtt{outer}^n[\mathsf{from}\ \pi^{k_2-n_1}.\pi^{k_1}]$
$(\pi^{k_2} \setminus n_1).\pi^{k_1-n} = (\pi^{k_2-n_1}.\pi^{k_1}) \setminus n$
$\pi^{k_2-n_1}.\pi^{k_1-n} = \pi^{k_2-n_1}.\pi^{k_1-n}$ □

The following lemma, stating that the type of an expression e is preserved (modulo moving paths) when the expression is copied into a position C.

Lemma 33 (Moving paths preserves typing)**.** *If* $\Delta[\mathsf{in}\ C]; \Gamma \vdash e : T$
then $\Delta; \Gamma[\mathsf{from}\ C] \vdash e[\mathsf{from}\ C] : T[\mathsf{from}\ C]$

Proof. By induction over the typing rules. We show only one case, since the others are either similar or trivial.

(FIELD-ACCESS-T) We have

(a) $\Delta[\mathsf{in}\ C];\Gamma \vdash e : C'$ from the premise,

(b) $\mathsf{mType}(\Delta[\mathsf{in}\ C], C', f) = T$ from the side condition, and

(c) $\Delta;\Gamma[\mathsf{from}\ C] \vdash e[\mathsf{from}\ C] : C'[\mathsf{from}\ C]$ from the inductive hypothesis and (a).

We have to prove that
$\Delta;\Gamma[\mathsf{from}\ C] \vdash e.f[\mathsf{from}\ C] : T[\mathsf{from}\ C'][\mathsf{from}\ C]$.
By (b), (BASIC-T) and definition of _[in _] we know that
$\mathsf{mType}(\Delta, C'[\mathsf{from}\ C], f) = T$.
By definition $e.f[\mathsf{from}\ C] = e[\mathsf{from}\ C].f$, which is typed by
rule (FIELD-ACCESS-T), with premise is (c), in this way:
$\Delta;\Gamma[\mathsf{from}\ C] \vdash e[\mathsf{from}\ C].f : T[\mathsf{from}\ C'[\mathsf{from}\ C]]$.
Finally, by Lemma 32 we show that $T[\mathsf{from}\ C'[\mathsf{from}\ C]] = T[\mathsf{from}\ C'][\mathsf{from}\ C]$. □

Lemma 34 (Moving paths preserves subtyping)**.** *If* $\Delta[\mathsf{in}\ C] \vdash T_1 \leq T_2$
then $\Delta \vdash T_1[\mathsf{from}\ C] \leq T_2[\mathsf{from}\ C]$

Proof. Analogous to the previous lemma. □

Lemma 35 (Substitution)**.** *If* $\Delta;\Gamma, x{:}T_1 \vdash e : T_1'$ *and* $\Delta;\emptyset \vdash v : T_2$ *and* $\Delta \vdash T_2 \leq T_1$,
then $\Delta;\Gamma \vdash e[{}^{v}/_{x}] : T_2'$ *and* $\Delta \vdash T_2' \leq T_1'$.

Lemma 36 (FJIG$_\star$ substitution lemma)**.** *If* $\Delta \vdash T_2 \leq T_1$ *then:*

1. *If* $\Delta;\emptyset \vdash e_1 : T_1$, $\Delta;\emptyset \vdash e_2 : T_2$, *and* $\Delta;\emptyset \vdash \mathcal{E}^{\mathrm{r}}[\![e_1]\!] : T_1'$
   then $\Delta;\emptyset \vdash \mathcal{E}^{\mathrm{r}}[\![e_2]\!] : T_2'$ *and* $\Delta \vdash T_2' \leq T_1'$.

2. *If* $\Delta;\Gamma, \overline{x}{:}\overline{T}_1 \vdash e : T_1$, $\Delta \vdash \overline{T}_1 \leq \overline{T}_2$. $\Delta;\emptyset \vdash \overline{e} : \overline{T}_2$
   then $\Delta;\Gamma \vdash e[{}^{\overline{e}}/_{\overline{x}}] : T_2$ *and* $\Delta \vdash T_1 \leq T_2$.

Proof. Analogous to Lemma 4. □

Theorem 37 (FJIG$_\star$ Subject-reduction)**.** *If* $\emptyset \vdash cv : ct$, $\mathsf{isComplete}(ct)$, $\Delta = ct[\mathsf{in}\ \pi]$, $\Delta;\Gamma \vdash e : T$,
$\sigma = cv[\mathsf{in}\ \pi]$ *and* $e_1 \underset{\sigma}{\rightarrow} e_2$, *then* $\Delta;\Gamma \vdash e_2 : T_2$ *and* $\Delta \vdash T_2 \leq T_1$.

Proof. By induction over the reduction rules. We show only one case, since the others are either similar or trivial.

Case $\dfrac{}{C\,(\overline{fv})\,.m\,(\overline{v}) \underset{\sigma}{\rightarrow} e[\mathsf{from}\ C][{}^{\overline{v}}/_{\overline{x}}][{}^{C\,(\overline{fv})}/_{\mathtt{this}}]}\ (\text{INVK})\ \mathsf{mBody}(\sigma, C, m) = \langle \overline{x}, e \rangle$

typed by $\text{(INVK-T)}\dfrac{\begin{array}{c}\Delta;\emptyset \vdash C\,(\overline{fv})\,:\,C \\ \Delta;\emptyset \vdash \overline{v}\,:\,\overline{T} \\ \Delta \vdash \overline{T} \leq (\overline{T}'[\mathsf{from}\ C])\end{array}}{\Delta;\emptyset \vdash C\,(\overline{fv})\,.m\,(\overline{v})\,:\,T_0[\mathsf{from}\ C]}\ \begin{array}{l}\mathsf{mType}(\Delta, C, m) = \overline{T}' \to T_0 \\ \overline{T}' = T_1 \ldots T_n\end{array}$

In this case, $C\,(\overline{fv})$ is typed by rule

$$\text{(OBJ-T)}\frac{\begin{array}{c}\Delta;\Gamma \vdash v_i\,:\,T_i \quad \forall i \in 1..n \\ \Delta \vdash T_i \leq (T_i'[\mathsf{from}\ C]) \quad \forall i \in 1..n\end{array}}{\Delta;\Gamma \vdash C\,(\overline{fv})\,:\,C}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, C) \\ \overline{fv} = \mathbf{this}.f_1 = v_1\mathbf{;}\ \ldots\ \mathbf{this}.f_n = v_n\mathbf{;} \\ \mathsf{defFields}(\Delta, C) = f_1{:}\epsilon\ T_1', \ldots, f_n{:}\epsilon\ T_n'\end{array}$$

By definition of $\mathsf{nonAbs}(\Delta, C)$ and $\mathsf{isComplete}(ct)$, we know that all methods of C are not **`abstract`**. By definition of $\mathsf{mType}(ct, \pi, m)$ we know that $\mathsf{cType}(\Delta, C)$ contains a (non abstract) method m with body e.
By (BASIC-T) we know that

$$\text{(METHOD-T)}\frac{\Delta[\mathsf{in}\ C];\Gamma \vdash e\,:\,T \quad \Delta[\mathsf{in}\ C] \vdash T \leq T_0}{\Delta[\mathsf{in}\ C] \vdash mh\,\{\,\mathbf{return}\ e\mathbf{;}\,\}\,:\,(m{:}T_1 \ldots T_n \to T_0)}\ \begin{array}{l}\Gamma = \mathbf{this}{:}\Lambda, x_1{:}T_1, \ldots, x_n{:}T_n \\ mh = T_0\,m(T_1\,x_1, \ldots, T_n\,x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n\end{array}$$

holds. By Lemma 33 $\Delta;\Gamma[\mathsf{from}\ C] \vdash e[\mathsf{from}\ C]\,:\,T[\mathsf{from}\ C]$ holds.
By Lemma 36-(2) we know that $\Delta;\Gamma[\mathsf{from}\ C] \vdash e[\mathsf{from}\ C][\overline{v}/_{\overline{x}}][^{C\,(\overline{fv})}/_{\mathbf{this}}]\,:\,T'$ and $\Delta \vdash T' \leq T[\mathsf{from}\ C]$. From $\Delta[\mathsf{in}\ C] \vdash T \leq T_0$, by Lemma 34 we know that $\Delta \vdash T[\mathsf{from}\ C] \leq T_0[\mathsf{from}\ C]$, the proof is concluded by (TRANS). □

Chapter 6

Meta-language for composition and nesting

> *An answer for you? [...] Yes. I have. [...] To the great Question of Life, the Universe and Everything [...] You're really not going to like it [...] The Answer to the Great Question Of Life, the Universe and Everything Is Forty-two.*
>
> (Douglas Adams - The Hitchhiker's Guide to the Galaxy)

In this chapter we integrate METAFJIG$_1$ and FJIG$_\star$, that is, we extend FJIG$_\star$ with a meta-level, analogously to what we have done with FJIG$_1$, obtaining METAFJIG$_\star$.

The expressive power of the meta-level, together with the capability of representing a whole component as a single class, will allow to encapsulate a library within a single meta-expression. Moreover, the possibility offered by the meta-level to write classes whose structure depends on an external source, like a database table, will be generalized to a whole hierarchy, as one can extract from a whole database or XML schema.

The conventional and meta-level features of METAFJIG$_\star$ are informally presented in Section 6.1, whereas Section 6.2 shows some real-world examples. Section 6.3 and Section 6.4 contain the formal definition of the language. Section 6.5 and Section 6.6 formally define checked compile-time execution and state and prove its properties. Finally, Section 6.7 describes how the prototype compiler works, notably explaining how it is built on top of Java compiler and Java Virtual Machine.

6.1 Examples

In METAFJIG$_\star$, an application is an expression of primitive type `class`. For instance, the following is a valid METAFJIG$_\star$ application, consisting in two basic classes composed by sum:

```
abstract{
  abstract int result()
  int main(){return this.result();}
}
[+]
{
  int result(){return 2;}
}
```

This application is reduced to the following:

```
{
 int main(){return this.result();}
 int result(){return 2;}
}
```

A basic class can have nested classes, that is, class declarations where class names are associated with arbitrary expressions of primitive type **class**.

Moreover, as in MetaFJig$_1$, a class definition can be the result of a method. For instance, in the following basic class:

```
{
  C = {
    class m(){
      return { int one(){return 1;} };
    }
  }
  D = new C().m() [+] copy C
}
```

method `m` returns a value of type **class**, that is, a class declaring the method `one`. Class `D` is defined by an expression that needs to be evaluated in order to obtain a class. Note that in MetaFJig$_\star$ we write **copy** _ to explicitly denote the injection from class paths to class expressions, which is silent in FJig$_\star$. This is necessary to distinguish, e.g., the constant meta-expression (literal) `C` of type **cpath**, which could be used as second argument of an alias or third of a redirect operator, from the expression **copy** `C` of type **class** used in the example.

This application is reduced, in some steps, to:

```
{
  C = //as before
  D =
    { int one(){return 1;} } [+]
    {
      class m(){
```

```
        return { int one(){return 1;} };
      }
    }
}
```

and finally to:

```
{
  C = //as before
  D = {
    int one(){return 1;}
    class m(){
      return { int one(){return 1;} };
    }
  }
}
```

Recall that in METAFJIG$_1$, in order to ensure meta-level soundness, we require expressions to be typechecked w.r.t. programs which are *strongly well-typed*, that is, such that meta-expressions of shape basic classes occurring at any inner meta-level are well-typed as classes. In order to ensure meta-level soundness for METAFJIG$_\star$ an analogous requirement is needed. However, in order to check whether a meta-expression of shape basic class denotes a well-typed class, all its nested classes must be (recursively) basic classes as well. Hence, during checked compile-time execution a basic class is considered a value only if *all* right-hand sides of class declarations, at any inner meta-level, are basic classes as well, otherwise they need to be reduced. This is illustrated by the example below.

```
{
  A = { class m1(){ return {}; } }
  B = { class m2(){
    return
      {
        C = new outer.outer.A().m1()
        D = { int m3(C x){return x.k();}}
      }
     [+] new outer.A().m1()
    }
  }
  E = new B().m2()
}
```

Here, `B` is not a value, since the basic class inside method `m2` contains the declaration of `C`, that still needs to be reduced. Hence, we cannot check whether `B` is strongly well-typed, hence it is not possible to typecheck and reduce `new B().m2()`; the application is first reduced to

```
{
  A = { class m1(){ return {}; } }
  B = { class m2(){
    return
      {
        C = {}
        D = { int m3(C x){return x.k();}}
      }
     [+] new outer.A().m1();
    }
  }
  E = new B().m()
}
```

Now `B` is a value, and could be instantiated, but method `m3` of class `D`, declared inside method `B.m2`, uses method `k` over an instance of (empty) class `C`. Hence, we get a compilation error.

Some readers could find counter-intuitive that reduction takes place inside the body of a method. However, it is not unusual in meta-programming approaches to perform reductions inside lambda-abstractions: for example, Meta-ML escape operator `~(exp)` allows the meta-programmer to execute the expression `exp` even if it is written inside a function declaration. In other words, all expressions occurring as right-hand sides of class declarations can be seen as "escaped up to the top level".

All the examples in METAFJIG$_1$ shown in Section 4.1 can be rephrased, with minimal syntactic adjustments, in METAFJIG$_\star$. The whole next section is devoted to interesting examples of METAFJIG$_\star$ usage.

6.2 Expressive power

Extensions In this section we will use many extensions that are implemented in the prototype but not formally modelled, since they pose no significant new technical problems.

First of all, we assume to have the primitive type **`type`**, that is, a supertype for both **`cpath`** and **`path`**. Moreover, all primitive types can now be used as expressions of type **`type`**. We extend the redirect operator in the obvious way, in order to be able to redirect to any type. It is possible to redirect to a primitive type only if the source is an empty abstract class with an empty **`implements`** declaration.

For emulating type-driven translations, we introduce the following *introspection* primitive operators: cv`[`**`inames`** π`]` and cv`[`**`cnames`** π`]`. They extract from a class value cv information about the class contained in the specified path; that is, an array containing the names of fields

and methods and the names of directly nested classes, respectively.

The prototype also supports operators for the type of a field or method i in π, the parameter types of the constructor and the implemented types. However, they do not appear in the following example code snippets.

We assume the class `String` to have, in addition to all the methods that it declares in Java, the methods `toIName` and `toPath`, which produce the **`iname`** and **`path`** corresponding to the `String` instance, if any, throwing an exception if this is not the case.

In the language presented in this thesis there are no static methods and fields, nor private members. Many of the examples in this chapter could be better expressed if such features where available. We plan to add such features in future work.

Composition exceptions Composition errors arising in languages allowing programmers to freely compose (language) terms can be very difficult to understand. In MetaFJig$_\star$, instead, the programmer can only compose classes using a predefined set of primitive composition operators. Each operator application may fail, producing an exception. Differently from the case of arbitrary composition, composition exceptions depend only on a small set of conditions, that should be examined by the programmer, providing useful and contextualized error messages. Moreover, such conditions depend only on the (public) interface of composed classes, that is, the implementation is immaterial.

Making expressive meta-programming safe and easy to debug is one of the main objectives in the design of MetaFJig$_\star$. Clearly, providing an helpful and contextualized error reporting is not easy. However, MetaFJig$_\star$ makes it possible to report error messages as good as if the checks were performed by the compiler, provided that enough effort is invested in error reporting code.

Notably, any composition operator may produce composition errors, and the exception object explains the reason for the failure. For example, if an operator application breaks an implements relation, then an exception of type `ImplementsRelationBrokenException` is raised. Such an exception contains two fields, **`path`** `subtype;` and **`type`** `supertype;`, such that the class denoted by the former implements the latter.

Utility classes We assume that all the examples are written at the first level of some application.

We first define some (boring) utility classes:

- `MessageException` and `MessageInfoException` are simple exception classes with a message, and a string message field and an additional information field, respectively,
- `PathUtil` concatenates paths and verifies whether a path is *simple*, that is, its size is exactly one,

- `Introspection` provides methods to verify whether a class contains a name or a path,
- `Fresh` generates names that are not contained inside a specific path in a class.

```
MessageException = implements RunTimeException{
  String message;
  constructor(String s){this.message=s;}
  String toString(){return this.message;}
}
MessageInfoException = implements RunTimeException{
  String message;
  Info = abstract{}
  Info info;
  constructor(String s,Info i){this.message=s;this.info=i;}
  String toString(){return this.message;}
}

PathUtil = {
  path append(path p1,path p2){
    if(p1==.<>)return p2;
    if(p2==.<>)return p1;
    return (""+p1+p2).toPath();
  }
  boolean isSimple(path p){
    return p!=.<> && (""+p).lastIndexOf(".")==0;
  }
}

Introspect = {
  class c;
  constructor(class c){this.c=c;}
  boolean hasName(path p,iname n){
    for(iname nn:this.c[inames p])
      if(nn==n)return true;
    return false;
    }
  boolean hasPath(path p){
    try{
      iname[] a=this.c[inames p];
      return true;
    }catch(INamesException e){
      return false;
    }
  }
```

```
}

Fresh = {
  outer.Introspect i;
  int n;
  constructor(class c){
    this.i=new outer.Introspect(c);
    this.n=0;
  }
  iname newName(path p){
    iname result=("_"+this.n).toiName();
    this.n=this.n+1;
    if(!this.i.hasName(p,result)) return result;
    return this.newName(p);
  }
  path newPath(path p){
    path result=new PathUtil().append(p,("._"+this.n).toPath());
    this.n=this.n+1;
    if(!this.i.hasPath(result))return result;
    return this.newPath(p);
    }
  }
```

Rename

Now we can define a class supplying many useful types of renaming.

First of all, since applying a chain of renaming is very common, we will use the *fluent interface* pattern: the constructor takes a class, all the methods return the current instance of the `Rename` class and the method `end` produces the final result.

For example It will be possible to use class `Rename` in this way:

```
Foo = new Rename(e)
  .ofName(.<>,foo,bar)
  .ofName(.a.b,foo,bar).end();
```

Exception classes related to renaming are defined on top of `Message` and `MessageInfo`. Specifically, there are four types of error that a renaming may produce: the old name is not denoting a member, the new name is already denoting a member, removing the old name breaks a subtype relation with some (even external) type or adding the new name we break a supertype relation with some (necessarily) internal class implementing path `p`.

The method `ofName` first checks for the first two error cases, then tries to apply the operator performing the rename, and, finally, checks for other possible errors. The third error case arises when the `ImplementsRelationBroken` exception reports the path to be the supertype, otherwise we are in the last error case.

For conciseness from now on we replace trivial and not meaningful code parts with dots.

```
Rename = {
  class c;
    constructor(class c){
      this.c=c;
    }
  class end(){return this.c;}
  NotExistingSourceException = copy outer.Message
  AlreadyPresentTargetException = copy outer.Message
  SubtypeRelationBrokenException =
    copy outer.MessageInfo[redirect .Info to type]
  SupertypeRelationBrokenException =
    copy outer.MessageInfo[redirect .Info to path]

  <> ofName(path p,iname o,iname n){
    outer.Introspection i=new outer.Introspection(this.c);
    if(!i.hasName(p,o))throw new NotExistingSourceException(...)
    if(o==n)return this;
    if(i.hasName(p,n))throw new AlreadyPresentTargetException(...);
    try{
      this.c=this.c[alias o to n in p][redirect o of p to n];
      return this;
    }catch(ImplementsRelationBrokenException e){
      if(p==e.supertype)
        throw new SupertypeRelationBrokenException(..., e.subtype);
      if(p==e.subtype)
        throw new SubtypeRelationBrokenException(...,e.supertype);
         //code not reachable, reports bug if executed
        throw new outer.MessageException(...);
    } catch(CompositionException e){
      //code not reachable, reports bug if executed
      throw new outer.MessageException(...);
    }
  }
 // continues below
```

The programmer of class `Rename` feels quite comfortable that no other possible error situations are possible, however, having no formal proof of this statement, it is not guaranteed. Therefore a

`Message` is thrown for bug report.

As you can see in the above code, there is actually *one* line that effectively performs the composition operator, while the remaining code is devoted to error and bug reporting. All the error reporting is either imposing semantic restrictions (first two cases), or translating errors that could be simply propagated. However, we believe that when errors are propagated far from their original context, some of their meaning is lost.

Now we continue our description of the `Rename` class, introducing the capability of renaming paths; for example It will be possible to use class `Rename` in this way:

```
Foo = new Rename(e)
  .ofPath(.A.B,.A.C)
  .ofPath(.D,.E).end();
```

Since it is not possible to directly redirect a class with nested classes, we provide the method `_recursiveRedirect`. Since private members are not supported by our language, by convention names starting with underscore are not to be used by clients.

```
  <> ofPath(path o,path n){
    outer.Introspection i=new outer.Introspection(this.c);
    if(!i.hasPath(o))throw new NotExistingSourceException(...)
    if(o==n)return this;
    if(i.hasPath(n))throw new AlreadyPresentTargetException(...);
    try{
      this.c=this.c[alias o to n];
      this._recursiveRedirect(o,n);
      return this;
    }catch(CompositionException e){
      //code not reachable, reports bug if executed
      throw new outer.Message(...);
    }
  }
  void _recursiveRedirect(path o,path n){
    for(path p:this.c[cnames o])
      this._recursiveRedirect(
          new PathUtil().append(o,p),
          new PathUtil().append(n,p)
      );
    this.c=this.c[redirect o to n];
    }
  <> toFreshName(path p,iname o){
    return this.ofName(p,o,new Fresh(this.c).newName());
  }
  <> toFreshPath(path o){
```

```
    return this.ofName(o,new Fresh(this.c).newPath());
  }
 // continues below
```

We also define (short-cut) methods to rename an instance name or a path into a fresh one.

The impossibility of renaming an instance member involved in an **implements** relation looks very restrictive. Intuitively, having a class named `Heir` implementing another one named `Base`, it should be possible to rename at the same time such member in both classes. This requires the following three steps:

1. aliasing (to the new position) the member in `Heir`,
2. renaming the member in `Base`,
3. redirecting the member in `Heir`.

However, solving the general case is not so trivial. Moreover, it looks infeasible if the subtype relation for that member is circular or involves external class paths. In the next example we complete class `Rename` with a method `onMultipleName`, that performs the simultaneous rename of a name in an array of paths `ps`, obtained by method `_calculateDependency`, which calculates `ps` the correct order, potentially throwing `CircularImplementRelation` or `ExternalImplementRelation`.

For example It will be possible to use class `Rename` in this way:

```
Foo = new Rename(e)
  .ofMultipleName(.A.B,foo,bar)
  .end();

  CircularImplementRelationException = copy outer.MessageException
  ExternalImplementRelationException = copy outer.MessageException

  <> onMultipleName(path startP,iname o,iname n){
    outer.Introspection i=new outer.Introspection(this.c);
    if(i.hasPath(startP,o))
      throw new NotExistingSourceException(...);
    if(o==n)return this;
    path[]ps=this._calculateDependency(startP,o);
    for(path p:ps)
      if(!i.hasPath(p,n))
        throw new AlreadyPresentTargetException(...);
    try{
      for(path p:ps)
```

```
        this.c=this.c[alias o to n in p];
      for(int i=ps.length-1;i>=0;i--)
        this.c=this.c[redirect o of ps[i] to n];
      return this;
    }catch(CompositionException e){
      //code not reachable, reports bug if executed
      throw new outer.MessageException(...);
    }
  }
  path[] _calculateDependency(path p, iname n){...}
}
```

Methods returning constants

It is sometimes very useful to generate methods returning a constant. One could be tempted to write code like:

```
ConstMeth = {
  class apply(Object val){
    return { Object n(){return val;} };
  }
}
MyConst = new ConstMeth().apply(new Foo(...))
```

but such code is ill-formed, since `val` is out of scope inside the body of method `n`.

That is, as in METAFJIG$_1$, a basic class must be closed w.r.t. variables, it cannot refer to the parameters of an enclosing method.

The reason for this limitation is that flattening would produce a method containing an object, rather than an instantiation expression. Moreover, in a stateful language, the object identifier would be a memory address. This would have two implications:

- the whole model would become much more complicated, since source code would have to include some memory representation for preinitialized objects, and
- such objects would behave like global variables, similarly to static variables inside C functions. In the example, each invocation of **new** `MyConst().n()` would return the *same* instance of `Foo`.

However, an obvious workaround is present into the language addressing this limitation, allowing to preserve the good property of having no already instantiated objects in the source code. Now we present an example of this workaround for **int**. Extending this example for `String` is trivial.

```
ConstMeth = {
  class intMeth(path cName, iname mName,int val){
    class result= { C = { int pred(){return 0;} } };
    class succ=
      {
        C = {
          int n(){return this.pred()+1;}
          abstract int pred();
        }
      };
    for(int i=0;i<val;i=i+1){
      result=new Rename(result[+]succ)
        .toFresh(.C,pred)
        .ofName(.C,n,pred).end();
      }
    return new Rename(result)
        .ofName(.C,pred,mName)
        .ofPath(.C,cName).end();
    }
  class stringMeth(path cName, iname mName, String val){...}
}
```

As you can see, we generate the desired value through a sequence of method calls. In an extension with private members all the fresh names would be private, in this way, a code optimizer could inline all the method calls, producing exactly the code intuitively generated by the first example.

There is also an easy solution for generating constant methods returning arbitrary objects: make such objects serializable to `String`, and use such strings to recreate the objects in the initialization. Following this idea, one would write something like:

```
class m(Foo x){
  class result=
    {
      C = { abstract String initString(); }
      outer.Foo x;
      constructor(){
        this.x=new outer.Foo(new C().initString());
      }
    };
  result=result [+] new ConstMeth.stringMeth(.C,initString,x.serialize());
  return new Rename(result).toFreshPath(.C).end();
}
```

Enumerations

In the first versions of Java there was no primitive language support for enumerations, so programmers used integer numbers for such a purpose. In a language like METAFJIG$_\star$ it is possible to implement a support for enumeration *in* the language, without requiring a language extension. Informally we want to be able to generate a good enumeration class from a sequence of enumeration names. To embed this in the prototype, we write a method taking an array of *simple* paths[1], that generates an abstract class representing the enumeration type, containing a nested class for each element of the array. Each element of the enumeration has a method `id`, returning its position in the original input array. We would also like to have a static method `values`, returning the array of all the possible values for such enumeration. However, having no static members in the prototype, we will generate also another nested class called `Values` with such a method.

Of course, enumerations can be enriched by the automatic generation of other useful methods like `toString`, `hashCode`, `equals` and so on, but we generate only `id` and `values` in order to keep the example compact.

With such a class, writing

```
AB = new Enum().with(new path[]{.A,.B})
```

allows to obtain

```
AB = abstract{
  abstract int id();
  A = implements outer{int id(){return 0;}}
  B = implements outer{int id(){return 1;}}
  Values = {
    outer[]values(){
      return new outer[]{new outer.A(),new outer.B()}
    int size(){return 2;}
    }
  }
}
```

Now we present the code performing this composition: at the beginning we declare an `EnumException` and we use it in order to perform a simple check for input validation:

```
Enum = {
  EnumException = copy outer.MessageException
  void _check(path[]ps){
    if(ps.length==0) throw new EnumException(...);
    for(path p:ps){
```

[1] A simple path is a path of size one

```
      if(p==.Values) throw new EnumException(...);
      if(!new PathUtils().isSimple(p)) throw new EnumException(...);
    }
  }
 // continues below
```

Then we declare method `_produceEnumElements` using local variables `result` and `succ` to generate nested classes. It relies on the nested class (originally) called `EnumElem`, and for each class we generate a method, (originally) called `n`, that creates an instance of such a nested class and put it into the array in the right position. Such array will be the result for the `values` method. Abstract method `pred` is called to propagate the initialization. More formally, in the for-loop

- `.EnumElem` becomes the correct enumeration name,
- `pred` becomes a fresh name and
- `n` becomes `pred`, so that at the next iteration it could satisfy the `pred` method inside the `succ`.

This technique is analogous to the one used in the implementation of `ConstMeth`.

Note how, at the and of the method, we add the `id` method to the nested classes.

```
class _produceEnumElements(path[]ps){
  class result=
    abstract{
      EnumElem = implements outer{}
      Values = {
        outer [] pred(outer [] v,int pos){return v;}
      }
    };
  class succ=
    abstract {
      EnumElem = implements outer{}
      Values = {
        outer [] n(outer [] v,int pos){
        v[pos]=new outer.EnumElem();
        return this.pred(v,pos+1);
      }
      abstract outer [] pred(outer [] v,int pos);
      }
    };
  for(path p:ps){
    result=new Rename(result[+]succ)
```

```
          .toFreshName(pred)
          .ofName(n,pred)
          .ofPath(.EnumElem,p).end();
      }
    for(int i=0;i<ps.length;i=i+1)
      result=result[+] new ConstMeth.intMeth(ps[i],id,i);
    return result;
  }
 // continues below
```

Now we are ready to write the method `with`, where local variable `result` declares method `values` that initializes the array with all the values equals to the first enumeration element,[2] and then call `pred` to properly initialize the array when composed with `this._produceNestedClasses()`. After such composition, name `pred` is no more relevant and can be "hidden". Note that method `size` is obtained by generating a method returning the length of the input.

```
  class with(path[]ps){
    this._check(ps);
    class result=
      abstract{
        abstract int id();
        EnumElem = implements outer{ abstract int id(); }
        Values = {
          abstract outer [] pred(outer [] v,int pos);
          abstract int size();
          outer [] values(){
            outer[] data=new outer[this.size()]{new outer.EnumElem()};
            return this.pred(data,0);
          }
        }
      };
    result=result[redirect .EnumElem to ps[0]];
    result=new Rename(result [+] this._produceNestedClasses(ps))
        .toFreshName(pred).end();
    return result [+] new ConstMeth.intMeth(.Values,size,ps.length);
  }
 // continues below
```

Now one can use `Enum` in this fashion:

```
Direction = new Enum().with(new path[]{.Left,.Right,.Top,.Down})
```

[2] In the prototype syntax `new t[n]{v}` initializes an array of type `t` and length `n`, containing `n` times the value `v`.

```
.... new Direction.Left()...
```

Note that in a more complete prototype, including the Java *varargs* feature and static methods, one could write

```
Direction = Enum.with(.Left,.Right,.Top,.Down)
```

It is possible to declare one or more supertypes for enumerations in two ways: the user can simply sum the enumeration with an empty abstract class having the desired supertypes, or one can extend the enumeration example in order to support a method `implementing` with obvious meaning, as in the following example.

```
A = Enum.implementing(outer.B).with(.A1,.A2,.A3)
```

A simple way of extending the class `Enum` with such feature is the following:

```
  class implementing(cpath t){return new EnumProxy(t);}
  EnumProxy={
    cpath t;
    constructor(cpath t){this.t=t;}
    class with(path[]ps){
      class c=new outer.Enum().with(ps);
      path p=new Fresh(c).newPath(.<>);
      class d=implements T{ T = abstract{} };
      d=new Rename(d).ofPath(.T,p).end();
      return c[redirect p to t];
    }
  }
}
```

In Java enumerations may have fields and methods; such a feature is very useful, and can be implemented here too. The idea is that one already has a class where nested classes represent the enumeration elements, and want to automatically instrument such classes to behave like enumerations.

```
RichEnum = {
  class produce(class c){
    this._check(c,p);
    try{
      return c [+] new Enum().produce(c[cnames .<>]);
    }catch(CompositionException e){
      //code not reachable, reports bug if executed
      throw new outer.MessageException(...);
    }
  }
```

```
  void _check(class c){...}
}
```

We take in input a class `c`, that have to follow very strict rules:

- the top level has to be abstract,
- any first level nested class has to implement the top level one,
- no first level nested class is called `Values`,
- all first level classes provide a zero argument constructor and
- no non first level nested class is allowed to implement the top level one.

The method `_check` verifies all those rules. The implementation of such method could be rather long, since it is basically type checking.

As future work, we think of developing a library that helps checking rules of that kind.

After the check we can simply compose by sum class `c` with the enumeration obtained using the first level nested classes.

This sum will be safe in a language with private members. Here, instead, we are using fresh names, so an error can potentially arise if the user provided classes whose names clash with the fresh names generated during the `Enum` generation. To avoid this problem a customized sum can perform suitable renaming on such fresh names, however language support (private members) would indeed be the best solution.

Containers

The limitation that a basic class must be closed does not prevent from encoding a generic class by a method taking a **`cpath`** parameter and returning a **`class`**, a very natural way of encoding generics by meta-programming capabilities. Indeed, this can be achieved by writing:

```
{
    class m(cpath x){
      return {  ... MyX ...  }[redirect .MyX to x]
    }
}
```

We emphasize that, for instance, the following code is instead ill-formed:

```
{
    class m(cpath x){
      return {  ... x ... }
    }
}
```

This is illustrated more extensively by the example below, a “stack producer” providing, for each class `typeName`, a stack of `typeName`.

```
StackProducer = {
  class apply(cpath typeName){
    class c=
      abstract{
        Elem= abstract{}
        abstract Elem top();
        abstract <> pop();
        abstract <> push(Elem e);
        EmptyException = implements RunTimeException{}
        Empty = implements outer{
          Elem top(){
            throw new EmptyException();
          }
          outer pop(){
            throw new EmptyException();
          }
          outer push(outer.Elem e){
            return new outer.NonEmpty(e,this);
          }
        }
        NonEmpty = implements outer{
          outer.Elem elem;outer tail;
          constructor(outer.Elem e, outer t){
            this.elem=e;
            this.tail=t;
          }
          Elem top(){
            return this.elem;
          }
          outer pop(){
            return this.tail;
          }
          outer push(outer.Elem e){
            return new <>(e,this);
          }
        }
      };
    return c[redirect .Elem to typeName];
  }
}
```

This stack producer can be easily used in other generic code, for example:

```
TaskPerformerProducer= {
  class apply(cPath t, iName taskName){
    class c=
      {
        Elem=abstract {abstract void operation();}
        InternalStack=new outer.StackProducer().apply(Elem)
        InternalStack stack;
        constructor(){this.stack=new InternalStack.Empty();}
        void add(Elem e){
          this.stack=this.stack.push(e);
        }
        void performTasks(){
          try{
            for(InternalStack aux=stack;;aux=aux.pop())
              aux.top().operation();
          }catch(InternalStack.EmptyException e){}
        }
      };
    return c[alias operation to taskName in .Elem]
        [redirect operation of .Elem to taskName]
        [redirect .Elem to t];
  }
}
```

Note the role of class `Elem` in `c`: it is first used to generate class `InternalStack`, and at the end it is redirected itself to the required class `t`.

We have presented a very simple example of a generic container. In the following example we model a class able to generate different kinds of containers, taking as input the requirements of the user. That is, we do not provide a finite set of possible containers, but, instead, we generate on the fly the best container that fulfils the user needs.

In this way containers (and generic data structures in general) could be managed in a much more abstract way w.r.t. what is currently done in other languages.

In this example we take in consideration only *sequence containers*, that is, we exclude trees, graphs, maps and other more complex types of containers. For simplicity, all sequences have a zero-argument constructor generating an empty sequence. Users may perform different kinds of operations over sequences: insertion, removal and access. We use `Enum` to define such kinds, and `Operation` as a common supertype. Finally, we define a nested class `UserData` able to keep associations between **`iname`** and `Operation`.

```
Containers={
```

```
Operation = abstract{}
Insert = new Enum().implementing(Operation).with(new path[]{
  .Top,.Bottom,.Random,.NotRelevant,...})
Removal = new Enum().implementing(Operation).with(new path[]{
  .Top,.Bottom,.Random,.ByElem,...})
Access = new Enum().implementing(Operation).with(new path[]{
  .Top,.Bottom,.Random,.NotRelevant,.Iterator,.Size,...})

UserData = {
  ...
  constructor(){... intialize all to empty ...}
}
UserData userData;
constructor(){this.userData=new UserData();}
```

Again we use fluent interfaces, so `addOperation` returns the current instance and method `end` produces the result.

Method `end` declares a variable `result` with a nested class representing a (for now empty) implementation of the data structure, and a method produce that instantiate such generic class over the user provided type.

```
<> addOperation(iname n,Operation kind){/*save the input*/}
class end(){
 class result=
   {
    BasicImpl = abstract{Elem = abstract{}};
    class produce(type t){
      return copy BasicImpl[redirect .Elem to t];}
   };
  return result [+] adaptResult(getBasicImpl(this.userData));
}
```

The implementation for the data structure is provided by two methods:

- `getBasicImpl` according with the user requirements chooses or composes the best implementation. For example, if the user asks for random access, a linked list is not a good implementation; and a field maintaining the size of the structure is kept if the utility method for finding the size is required.[3]

  ```
  class getBasicImpl(UserData userData){...}
  ```

  It returns classes similar to the following:

[3] In order to allow easier reuse, `getBasicImpl` takes a parameter even if such data is present in a field, see the following example.

```
{
  BasicImpl = {
    Elem = abstract{}
    ...
    void insertTop(Elem e){...}
    ...
  }
}
```

- `adaptResult` applies a lot of renaming and aliasing in order to adapt the basic implementations according to the required shape: for example the user may want the insertion on top to be called `push`.

  ```
  class adaptResult(class c){...}
  ```

Now the user can define her own sets in this way:

```
Sets = new Collections()
  .addOperation(add,new Collections.Insert.NotRelevant())
  .addOperation(getElem,new Collections.Access.NotRelevant())
  .addOperation(remove,new Collections.Removal.ByElem())
  .addOperation(iterator,new Collections.Access.Iterator()).end()

IntSet=new Sets().produce(int)
```

Now it is easy to define stacks, queues, vectors and so on. This is, of course, a very minimal example, there are many more kinds of *user needs*, like (insertion) ordering criteria. Moreover, in the example we do not ask the user about the kind of type the container can support. In the `Sets` example; we require `Removal.ByElem`; does this mean we require methods `equals` (and maybe `hashCode`) to be provided by the element types? However providing a well-designed class for generating collections is out of the scope of this simple proof of concept.

The criteria to choose the best implementation (depending from the user requirements) can be freely changed or updated. That is, each company can have its own preferred optimization strategy for collections. This can be obtained in an elegant way trough the following code:

```
CustomizeContainers = {
  class produce(class criteria){
    return copy outer.Containers
      [.UserData redirect outer.Containers.UserData]
      [restrict getBasicImpl] [+] criteria;
  }
}
```

That can be used in this way:

```
MyContainers = new CustomizeContainers().produce(
  { class getBasicImpl(outer.Containers.UserData userData){...} })
```

Expression problem revisited

In FJIG$_\star$ we show how to solve the expression problem using sum. However, it is common that many operations share a simple *propagation* behaviour for the big majority of the data variant. It seems reasonable to provide only the relevant code, without repeating the propagation one.

In the context of language implementations some common operations are, for example:

- to collect all the free variables,
- to collect all the used types,
- to remove a layer of syntactic sugar,
- to rename all the `foo` names into `bar`.

Now we will show how to add a method to collect all the numbers occurring in an expression. To keep the example simple and understandable, we require the data structure to follow very strict rules:

- the top level is abstract,
- there are no second level nested classes,
- there is a special nested abstract class named `Param`,
- all the other nested classes implement the top level class,
- the top level, and all the nested except `Param` provides method named `operation` implementing the desired propagation strategy.

Here we provide an example of such a class hierarchy.

```
Exp = abstract{
  Param = abstract{}
  void operation(Param p){}
  Num = implements outer{
    int n;
```

```
    constructor(int n){this.n=n;}
    void operation(outer.Param p){}
    }
  Sum = implements outer{
    outer left;
    outer right;
    constructor(outer left,outer right){
      this.left=left;
      this.right=right;
    }
    void operation(outer.Param p){
      this.left.operation(p);
      this.right.operation(p);
    }
  }
  ...
}
```

It is possible to write an `AddOperation` class such that

```
ExpCollectNum = new AddOperation().produce(abstract{
  abstract void collectNum(IntSet x);
  Num = implements outer{
    abstract int n;
    constructor(int n){}
    void collectNum(IntSet x){
      x.add(this.n);
    }
  }
})
```

will produce

```
ExpCollectNum = abstract{
  Param = abstract{}
  void operation(Param p){}
  Num = implements outer{
    int n;
    constructor(int n){this.n=n;}
    void operation(outer.Param p){}
    void collectNum(IntSet x){
      x.add(this.n);
    }
  }
  Sum = implements outer{
```

```
    outer left;
    outer right;
    constructor(outer left,outer right){
      this.left=left;
      this.right=right;
    }
    void operation(outer.Param p){
      this.left.operation(p);
      this.right.operation(p);
    }
    void collectNum(IntSet p){
      this.left.collectNum(p);
      this.right.collectNum(p);
    }

  }
  ...
}
```

Note that also the code of the operation has to follow strict rules:

- the top level is abstract,
- there are no second level nested classes,
- all the nested classes implement the top level class,
- the top level provides exactly one method, not named `operation`,
- all the nested classes do not provide methods in conflict with some other method already declared in the data structure.

Here is presented the code for `AddOperation`:

```
AddOperation = {
  void _check(class c,class op){...}
  iname _opName(class op){...}
  type _opType(class op){...}
```

Method `_check` verifies that both `c` and `op` follow the pattern. Methods `_opName` and `_opType` examine the *only* method present at top level and extract the name and the parameter type of such method, respectively.

```
  class produce(class c, class op){
    this._check(c,op);
```

```
    outer.Introspection i=new outer.Introspection(op);
    iname opN=this._opName(op);
    type opT=this._opType(op);
    class s=c;
    for(path p:c[cnames .<>])
      for(inames n:c[inames p])
        if(n!=operation || i.hasName(p,opN))
          s=s[restrict n in p];
    s=new outer.Rename(s[redirect .Param  to opT])
      .onMultipleName(.<>,operation,opN).end();
    return op [+] s [+] c;
  }
}
```

Variable `s` is a class obtained from `c`, where all (except the `operation` methods not implemented in `op`) is turned to abstract. `.Param` is redirected to the type used for the operation, and we set the operation name decided by the user. Finally, we compose the partial implementation `op`, provided by the user, the propagation implementation `s`, calculated by the algorithm, and the initial code.

In this example we require the propagation code to be hand written by the programmer. Another approach is to automatically generate such propagation code following some default pattern.

Moreover, we are aware that, with a well-designed hierarchy of *visitors* [GHJV95] it is possible to obtain a similar amount of code reuse. However, here we avoid the run time penalties of the visitor pattern.

6.3 Syntax and semantics

Syntax In Figure 6.1 we give the syntax of MetaFJig$_\star$, which is obtained by adding a meta-level to FJig$_\star$ analogously to how MetaFJig$_1$ is obtained by adding a meta-level to FJig$_1$.

Recall that in FJig$_\star$ an application is a class expression, which is expected to reduce to a class value by flattening. Correspondingly, in MetaFJig$_\star$ an application is a (meta-)expression, which is expected to reduce to a class value by compile-time execution.

We assume that, in a well-formed top-level expression, all class paths refer to existing classes.

Expressions are either conventional expressions and pre-objects, as in FJig$_\star$, or meta-expressions, or *error*.

MetaFJig$_\star$ supports meta-expressions denoting classes, instance member names, class paths and paths, which are expressions of type **`class`**, **`iname`**, **`cpath`**, and **`path`**, respectively.

e	::=		expression
		$x \mid e.f \mid e.m\,(\overline{e}) \mid \texttt{new}\ C\,(\overline{e})$	conventional
		$\mid C\,(\overline{fe})$	pre-object
		$\mid b$	basic class
		$\mid \texttt{copy}\ e$	class path (as class expression)
		$\mid e_1\,[+]\,e_2$	sum
		$\mid e\,[\texttt{restrict}\ e_2\ \texttt{in}\ e_1]$	restrict
		$\mid e\,[\texttt{alias}\ e^s\ \texttt{to}\ e^t\ \texttt{in}\ e']$	alias
		$\mid e\,[\texttt{alias}\ e^s\ \texttt{to}\ e^t]$	class alias
		$\mid e\,[\texttt{redirect}\ e^s\ \texttt{of}\ e'\ \texttt{to}\ e^t]$	redirect
		$\mid e\,[\texttt{redirect}\ e^s\ \texttt{to}\ e^t]$	class redirect
		$\mid i$	instance member name constant
		$\mid C$	class path constant
		$\mid \pi$	path constant
		$\mid error$	error
b	::=	$ch\,\{\,k\ \overline{d}\,\}$	basic class
ch	::=	$\mu\ \texttt{implements}\ \overline{C}$	class header
k	::=	$kh\,\{\overline{fe}\}$	constructor
kh	::=	$\texttt{constructor}\,(\,\overline{T\ x}\,)$	constructor header
fe	::=	$\texttt{this}.f = e\texttt{;}$	field expression
d	::=	$fd \mid md \mid cd$	(member) declaration
fd	::=	$\mu\ T\ f\texttt{;}$	field declaration
md	::=	$\texttt{abstract}\ mh\texttt{;} \mid mh\,\{\,\texttt{return}\ e\texttt{;}\,\}$	method declaration
cd	::=	$c = e$	class declaration
mh	::=	$T\ m(\overline{T\ x})$	method header
μ	::=	$\epsilon \mid \texttt{abstract}$	abstract modifier
n	::=	$i \mid c$	(member) name
f, m	::=	i	
T	::=	$C \mid \texttt{class} \mid \texttt{iname} \mid \texttt{cpath} \mid \texttt{path}$	type
C	::=	$\overline{\texttt{outer}}.\overline{c}$	class path
π	::=	$.\overline{c}$	path
$error$	::=	$\texttt{errorC}$	composition error
		$\mid \texttt{errorT}$	typechecking error
v	::=	$C\,(\overline{fv}) \mid cv \mid i \mid C \mid \pi$	value
fv	::=	$\texttt{this}.f = v\texttt{;}$	field value
σ	::=	$\overline{b}$	enclosing classes

Figure 6.1: MetaFJig$_\star$ syntax

$$
\begin{array}{lcl}
\mathcal{E}^{\mathrm{c}} & ::= & \Box \mid \mathcal{E}[\![\mathcal{B}]\!] \\
\mathcal{B} & ::= & ch\ \{\, k\ \overline{d}\ c = \Box \,\} \mid ch\ \{\, k\ \overline{d}\ c = \mathcal{E}[\![\mathcal{B}]\!] \,\} \\
 & & \mid ch\ \{\, k\ \overline{d}\ mh\ \{\, \texttt{return}\ \mathcal{E}[\![\mathcal{B}]\!]\texttt{;} \,\} \,\} \mid ch\ \{\, kh\ \{\, \overline{fe}\ \texttt{this}.f = \mathcal{E}[\![\mathcal{B}]\!]\texttt{;} \,\}\ \overline{d} \,\} \\
\mathcal{E} & ::= & \Box \mid \mathcal{E}.f \mid \mathcal{E}.m\,(\overline{e}) \mid e.m\,(\overline{e}, \mathcal{E}, \overline{e}') \mid \texttt{new}\ C\,(\overline{e}, \mathcal{E}, \overline{e}') \mid \mathcal{E}\,[+]\,e \mid e\,[+]\,\mathcal{E} \mid \texttt{copy}\ \mathcal{E} \\
 & & \mid \mathcal{E}\,[\,\texttt{restrict}\ e'\ \texttt{in}\ e\,] \mid e\,[\,\texttt{restrict}\ e'\ \texttt{in}\ \mathcal{E}\,] \mid e\,[\,\texttt{restrict}\ \mathcal{E}\ \texttt{in}\ e'\,] \\
 & & \mid \ldots
\end{array}
$$

Figure 6.2: Evaluation contexts for METAFJIG$_\star$ checked compile-time execution

Meta-expressions denoting classes correspond to class expressions in FJIG$_\star$. However, we write **copy** _ to explicitly denote the injection from class paths to class expressions, which is silent in FJIG$_\star$. This is necessary to distinguish the expression C of type **cpath** from the expression **copy** C of type **class**. To avoid confusion we will always use **copy** _ in the code examples in the rest of the thesis.

As in FJIG$_\star$, a basic class consists in an optional **abstract** modifier, a sequence of supertypes, a constructor, and a set of declarations. Supertype declarations, constructors, field and method declarations are exactly as in FJIG$_\star$, while class declarations can associate to class names c arbitrary expressions.

Types T are class paths C and primitive types for classes, instance member names, class paths and paths.

Correspondingly, values are objects, class values, and instance member name, class path and path constants.

The formal definition of class values cv is not given directly in the grammar. As motivated in previous section, a class value is a basic class where all right-hand sides of class declarations are basic classes. Formally, a class value cv is a basic class b such that $b = \mathcal{E}^{\mathrm{c}}[\![e]\!]$ implies $e = b'$, where the evaluation contexts $\mathcal{E}^{\mathrm{c}}$ for compile-time execution are defined in Figure 6.2.

The subterms of an expression which can be reduced by compile-time execution are either the whole expression (first case), or, for each subterm which is a basic class, the expressions appearing as right-hand side of class declarations either in the basic class itself (first case of $\mathcal{B}$), or, recursively, in basic classes which are subterms of class definitions, method bodies and field expressions (second, third and last case of $\mathcal{B}$, respectively). Here and in the following $\mathcal{E}$ is the one hole conventional context on expressions.

Reduction In Figure 6.3, Figure 6.4 and Figure 6.5 we define the reduction arrow for expressions. The relation is of the form $e \underset{\sigma}{\rightarrow} e'$, where we recall that σ is an environment, that is, a stack of enclosing basic classes.

$$\mathcal{E}^{\mathrm{r}} \;::=\; \Box \mid \mathcal{E}^{\mathrm{r}}.f \mid \mathcal{E}^{\mathrm{r}}.m\,(\overline{e}) \mid v.m\,(\overline{v}, \mathcal{E}^{\mathrm{r}}, \overline{e}) \mid \mathbf{new}\,C\,(\overline{v}, \mathcal{E}^{\mathrm{r}}, \overline{e}) \mid \mathcal{E}^{\mathrm{r}}\,[+]\,e \mid v\,[+]\,\mathcal{E}^{\mathrm{r}} \mid \mathbf{copy}\,\mathcal{E}^{\mathrm{r}} \mid \mathcal{E}^{\mathrm{r}}\,[\,\mathbf{restrict}\,e\,\mathbf{in}\,e'\,] \mid v\,[\,\mathbf{restrict}\,\mathcal{E}^{\mathrm{r}}\,\mathbf{in}\,e\,] \mid v\,[\,\mathbf{restrict}\,v'\,\mathbf{in}\,\mathcal{E}^{\mathrm{r}}\,] \mid \ldots$$

$$\boxed{e_1 \underset{\sigma}{\rightarrow} e_2}$$

$$\text{(CTX)}\ \frac{e_1 \underset{\sigma}{\rightarrow} e_2}{\mathcal{E}^{\mathrm{r}}[\![e_1]\!] \underset{\sigma}{\rightarrow} \mathcal{E}^{\mathrm{r}}[\![e_2]\!]} \qquad \text{(CTX-ERROR)}\ \frac{}{\mathcal{E}^{\mathrm{r}}[\![error]\!] \underset{\sigma}{\rightarrow} error}$$

$$\text{(FIELD-ACCESS)}\ \frac{}{C\,(\overline{fv})\,.f \underset{\sigma}{\rightarrow} v}\ \overline{fv}(f) = v$$

$$\text{(INVK)}\ \frac{}{C\,(\overline{fv})\,.m\,(\overline{v}) \underset{\sigma}{\rightarrow} e[\mathsf{from}\ C][{}^{\overline{v}}/_{\overline{x}}][{}^{C\,(\overline{fv})}/_{\mathbf{this}}]}\ \mathsf{mBody}(\sigma, C, m) = \langle \overline{x}, e \rangle$$

$$\text{(OBJ-CREATION)}\ \frac{}{\mathbf{new}\,C\,(\overline{v}) \underset{\sigma}{\rightarrow} C\,(\overline{fe}[\mathsf{from}\ \pi][{}^{\overline{v}}/_{\overline{x}}])}\ \begin{array}{l}\mathsf{kBody}(\sigma, C) = \langle \overline{x}, \overline{fe} \rangle \\ \mathsf{nonAbs}(\sigma, C)\end{array}$$

Figure 6.3: METAFJIG$_\star$ reduction rules 1/3

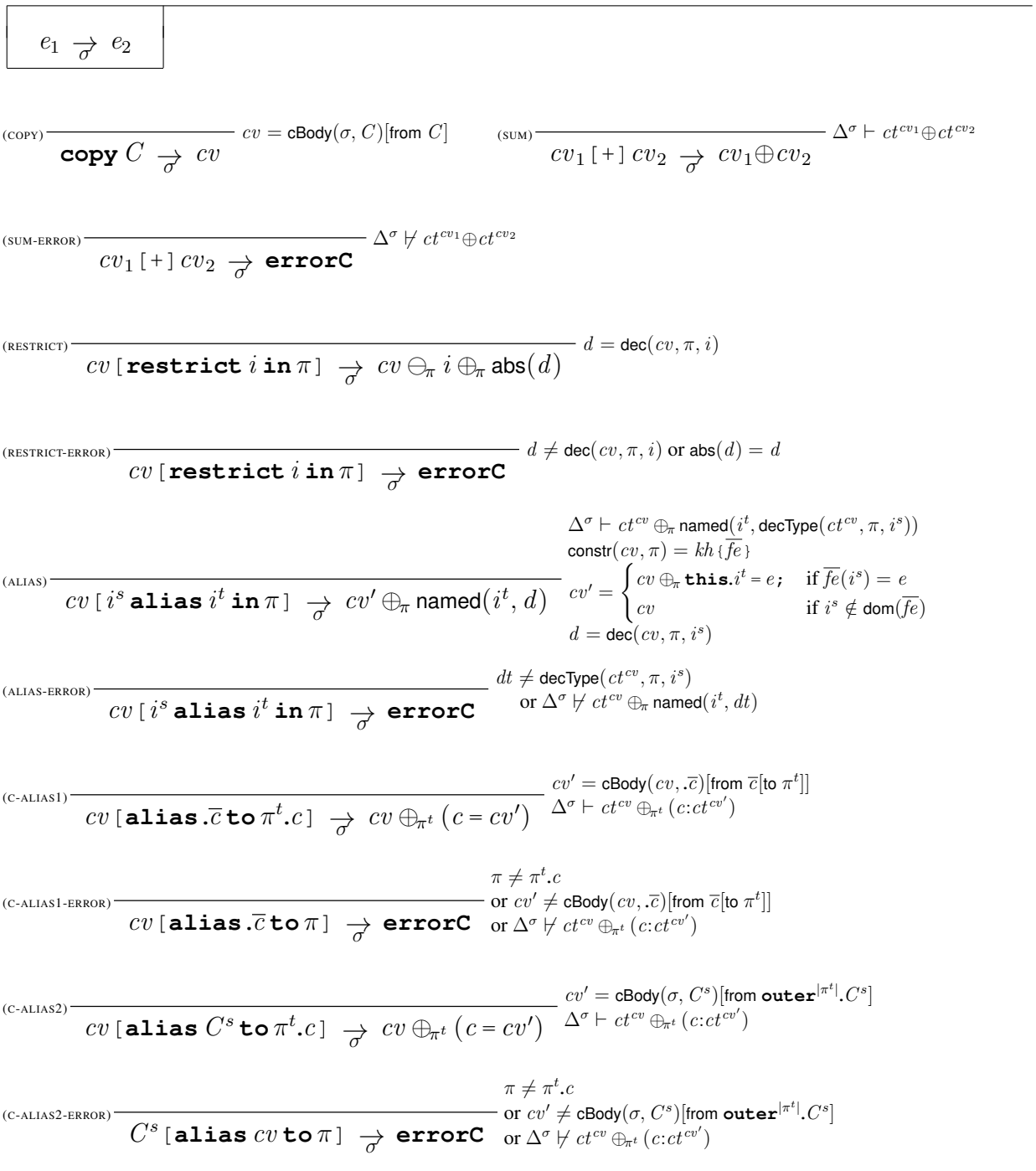

Figure 6.4: METAFJIG$_\star$ reduction rules 2/3

Figure 6.5: METAFJIG$_\star$ reduction rules 3/3

Evaluation contexts for reduction are the obvious extension of those of FJIG$_\star$.

Rule (CTX-ERROR) is the usual propagation of errors, while the other rules in Figure 6.3 are those of FJIG$_\star$.

Rules in Figure 6.4 and Figure 6.5 model reduction of meta-expressions denoting classes. Other meta-expressions have no reduction rules since they are constants.

Rule (COPY) can be applied when the class expression occurring in position C in σ is a class value (METAFJIG$_\star$ has a call-by-value semantics). The notations $\mathsf{cBody}(\sigma, C)$ and $cv[\mathsf{from}\ C^s]$ are defined analogously as in FJIG$_\star$. As in METAFJIG$_1$, for each composition operator there are two rules.

The former reduces the term analogously to the corresponding reduction rule in FJIG$_\star$, and the side conditions are analogous to the ones of the corresponding typing rule in FJIG$_\star$.

The latter corresponds to the case when the operator cannot be performed, hence a composition error is raised. In the prototype compiler, of course, different predefined exceptions are thrown.

We denote by Δ^σ the type environment extracted from σ and by ct^b the class type extracted from b, formally defined by:
$\Delta^{b_1 \dots b_n} = ct^{b_1} \dots ct^{b_n}$
$ct^b = [\mu \mid \overline{C} \mid kt^k \mid dt^d_1 \dots dt^d_n]$ if $cv = \mu\ \texttt{implements}\ \overline{C}\ \{\, k\ d_1 \dots d_n \,\}$
$dt^{\mu\ T\ f\texttt{;}} = f{:}\mu\ T$
$dt^{\mu\ T\ m(T_1\ x_1 \dots T_n\ x_n)\_} = m{:}\mu\ T_1 \dots T_n \to T$
$dt^{c\,=\,b} = c{:}ct^b$
$dt^{c\,=\,e} = \emptyset$ with $e \neq b$
$kt^{\texttt{constructor}\,(\,T_1\ x_1,\dots,T_n\ x_n\,)\ \{\,\overline{fe}\,\}} = T_1 \dots T_n$

All other notations are exactly the same of FJIG$_\star$. The only difference is that the auxiliary operators $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$ and $cv[\pi^s \rightsquigarrow C^t]$ have to be extended to address the following two issues:

- Since now a class value can occur inside a method body or field expression, deciding whether an internally written C denotes some π is more complex: in FJIG$_\star$ any cv' inside a cv is reachable with a path, and such path was used as accumulation parameter. Here over constructors or method declarations we use a form of the operator with another accumulation parameter j keeping trace of the number of enclosing basic class encountered in the expression. This difference is common to both operators,

- Since we use incremental typing as in METAFJIG$_1$, it is no longer possible to suppose method invocations and field accesses to be already annotated with the static type of the receiver. We use, instead, partial function $\mathsf{typeOf}(\sigma, \Gamma, e)$, inferring the *expected type* of e when such type is a class path internal w.r.t. σ. Intuitively "expected type" means that if e

$cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$

$cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t] = cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\epsilon\Lambda}$

$cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi'}$

$ch\ \{\, k\ \overline{fd}\ \overline{md}\ \overline{cd}\,\}[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi'} = ch\ \{\, k[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma'\pi'0}\ \overline{fd}\ \overline{md}[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma'\pi'0}\ \overline{cd}[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma'\pi'}\,\}$
with $\sigma' = cv \cdot \sigma$

$d[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi'}$

$T\ m(\overline{T\ x})\{\,\texttt{return}\ e\texttt{;}\,\}[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j} = T\ m(\overline{T\ x})\{\,\texttt{return}\ e[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j\Gamma}\texttt{;}\,\}$
with $\overline{T\ x} = T_1\ x_1, \ldots, T_n\ x_n$ and $\Gamma = x_1{:}T_1 \ldots x_n{:}T_n$

$(c = cv)[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi'} = c = (cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi'.c})$

$k[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi'}$

$\texttt{constructor}\,(\,\overline{T\ x}\,)\ \{\overline{fe}\}[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j} = \texttt{constructor}\,(\,\overline{T\ x}\,)\ \{\overline{fe}[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j\Gamma}\}$
with $\overline{T\ x} = T_1\ x_1, \ldots, T_n\ x_n$ and $\Gamma = x_1{:}T_1 \ldots x_n{:}T_n$

$e[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j\Gamma}$

...
$.i[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j\Gamma} = .i^t$ if $i = i^s, \mathsf{typeOf}(\sigma, \Gamma, e) = \texttt{outer}^j.C$ and $C[\mathsf{from}\ \pi'] = \pi$
$.i[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j\Gamma} = .i$ otherwise
...
$ch\ \{\, k\ \overline{d}\,\}[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j\Gamma} = ch\ \{\, k[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j+1}\ \overline{d}[i^s \underset{[\pi]}{\rightsquigarrow} i^t]_{\sigma\pi' j+1}\,\}$

Figure 6.6: MetaFJig$_\star$ auxiliary function for redirect

is well-typed, e will have such (static) type. Formally:
$\mathsf{typeOf}(\_, \Gamma, x) = \Gamma(x)$ and $\mathsf{typeOf}(\_, \_, \texttt{new}\ C\ (\_)\ ) = C$; if $\mathsf{cBody}(\sigma, C)$ is defined
with $\mathsf{typeOf}(\sigma, \Gamma, e) = C$, if $\mathsf{cBody}(\sigma, C')$ is defined:
$\mathsf{typeOf}(\sigma, \Gamma, e.f) = C'[\mathsf{from}\ C]$ if $\mathsf{cBody}(\sigma, C) = \_\ \{\_\_\ \mu\ C'\ f\ \texttt{;}\ \}$ and
$\mathsf{typeOf}(\sigma, \Gamma, e.m\ (\_)\ ) = C'[\mathsf{from}\ C]$ if $\mathsf{cBody}(\sigma, C) = \_\ \{\_\_\ \mu\ C'\ m(\_)\_\}$

The formal definition of $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$ is given in three logical steps:

- recursively descend along the nested classes of the starting class value cv, keeping trace in two accumulation parameters σ and π' of the enclosing basic classes and the explored path, respectively. Propagation inside k and md uses $j = 0$,
- exploring an expression, we keep trace of the variable environment Γ this expression can access.
- finally, propagates through all the kind of expressions; over field accesses or a method invocations we use partial function $\mathsf{typeOf}(\sigma, \Gamma, e)$ to infer the expected (static) type of e, and we change the referred name to i^t if the original name and the expected type denote the source name and the redirected type, respectively. We explore basic classes using $j+1$.

The formal definition of $cv[\pi^s \rightsquigarrow C^t]$ is similar to the one of $cv[i^s \underset{[\pi]}{\rightsquigarrow} i^t]$ and is given in two logical steps:

- recursively descend along the nested classes of the starting class value cv, keeping trace in an accumulation parameter π of the explored path. Formally:
  - $cv[\pi^s \rightsquigarrow C^t] = cv[\pi^s \rightsquigarrow C^t]_\Lambda$
    $(c = cv)[\pi^s \rightsquigarrow C^t]_\pi = c = (ce[\pi^s \rightsquigarrow C^t]_{\pi.c})$
    $ch\ \{\ k\ \overline{fd}\ \overline{md}\ \overline{cd}\ \}[\pi^s \rightsquigarrow C^t]_\pi = ch[\pi^s \rightsquigarrow C^t]_{\pi 0}\ \{\ k[\pi^s \rightsquigarrow C^t]_{\pi 0}\ (\overline{fd}\ \overline{md})[\pi^s \rightsquigarrow C^t]_{\pi 0}\ \overline{cd}[\pi^s \rightsquigarrow C^t]_\pi\ \}$

    note how propagation inside ch, k, fd and md uses $j = 0$
- finally, propagates through all the kind of expressions. Formally:
  - $\texttt{new}\ C\ (\overline{e})\ [\pi^s \rightsquigarrow C^t]_{\pi j} = \texttt{new}\ C[\pi^s \rightsquigarrow C^t]_{\pi j}\ (\overline{e}[\pi^s \rightsquigarrow C^t]_{\pi j})$
    $ch\ \{\ k\ \overline{d}\ \}[\pi^s \rightsquigarrow C^t]_{\pi j} = ch[\pi^s \rightsquigarrow C^t]_{\pi j+1}\ \{\ k[\pi^s \rightsquigarrow C^t]_{\pi j+1}\ \overline{d}[\pi^s \rightsquigarrow C^t]_{\pi j+1}\ \}$

    note how we explore basic classes using $j + 1$.

    When we reach a C that represents a path at top level, that is $C = \texttt{outer}^j.C'$ we replace C' with the shortest class path that correspond to C^t in position π, that is $C^t[\mathsf{to}\ \pi]$; if the (top level) path denoted by C' in π is π^s, that is $C'[\mathsf{from}\ \pi] = \pi^s$.

Δ	$::=$	$\overline{pct}$	class type environment
pct	$::=$	$ct \mid \overline{c{:}pct}$	partial class type
ct	$::=$	$[\mu \mid \overline{C} \mid kt \mid \overline{dt}]^{\lambda}$	class type
λ	$::=$	$E \mid \epsilon$	executable label
kt	$::=$	$\overline{T}$	constructor type
dt	$::=$	$i{:}\mu\ imt \mid c{:}ct$	declaration type
imt	$::=$	$T \mid \overline{T} \rightarrow T$	field or method type
Γ	$::=$	$\overline{x{:}T}$	parameter type environment
L	$::=$	$0 \mid \star$	type system

Figure 6.7: METAFJIG$_\star$ types and type environments

– $C[\pi^s \rightsquigarrow C^t]_{\pi j} = \begin{cases} \texttt{outer}^j.(C^t[\text{to } \pi]) & \text{if } C = \texttt{outer}^j.C' \text{ and } C'[\text{from } \pi] = \pi^s \\ C & \text{otherwise} \end{cases}$

6.4 Type system

Types and type environments are defined in Figure 6.7. There are two main differences w.r.t. FJIG$_\star$.

The first difference is that, since our typechecking is incremental, the type information currently available about a class can be incomplete. This is formally modelled by *partial* class types. A partial class type can be either a standard FJIG$_\star$ class type (enriched by the "executable" label explained below) or just a map from names of nested classes to their types. The former case means that we have all the type information about the class. Note that this is only possible if the class is a class value. The latter case means that we do not have the whole type information about the class yet, but some of its nested classes could have been typechecked already.

For instance, the following basic class cannot be typechecked yet:

```
class{
  abstract int f1;
  A = {
    B = {}
    C = {} [+] {}
  }
  D = {
    abstract int f2;
```

```
    E = {}
    }
}
```

can be typed as follows:

A:(B:[ϵ | $\emptyset$ | $\emptyset$ | $\emptyset$]),
D:[ϵ | $\emptyset$ | $\emptyset$ | f2:(**abstract int**) E:[ϵ | $\emptyset$ | $\emptyset$ | $\emptyset$]]

The second difference is that class types can be labelled by E, for "executable". A class with an executable type must have no abstract fields or methods and its method bodies and field expressions can only instantiate, recursively, executable classes, see (BASIC-T) and (NEW-T) in the following. As the name suggests, a main expression can only instantiate executable classes of an application, as formally stated in Theorem 41. Of course, non executable classes can be safely used as types and for code reuse, that is, as arguments of composition operators.

Note that in FJIG$_\star$ we use a much coarse-grained requirement, that is, the whole application must be "executable" in the sense that all non abstract classes have no abstract fields or methods, see the definition of predicate isComplete before Theorem 31. Here this would be not adequate, since we could need to reduce expressions inside basic classes that are incomplete.

Typing rules for environments and basic classes are given in Figure 6.8.

The judgment $\vdash \sigma : \Delta$ is defined in the first three rules.

(ENV-T) typechecks a stack of basic classes using either (BASIC-T) or (BASIC-PARTIAL-T): the first one is used when a basic class is a well-typed class value cv, while the latter is (non deterministically) used when the basic class is still not a class value, or it is impossible to type such class value.

The other rules of Figure 6.8 define the type judgement for basic classes, and are exactly as in FJIG$_\star$. The only difference is that in (BASIC-T) the class type can be labelled executable only if the class contains no abstract fields or methods.

Typing rules for expressions are given in Figure 6.9 and Figure 6.10.

Rules for conventional constructs and pre-objects are as in FJIG$_\star$, moreover, as in METAFJIG$_1$, new rules are needed for the new kinds of expression. The only difference is that (NEW-T) and (OBJ-T) ensure that an executable class can only instantiate executable classes.

Exactly as in METAFJIG$_1$, we consider two different type judgements for expressions: *weak well-typedness*, denoted by the label 0, and *strong well-typedness* denoted by the label $\star$. Note that only class values cv can be (strongly or weakly) well-typed, that is, an expression containing basic class that are not class values can not be (strongly or weakly) well-typed. In checked compile-time execution this is ensuring the correct order of reduction.

As in METAFJIG$_1$, the only difference between the weak and the strong type system is about

$\vdash \sigma : \Delta$

$$\text{(ENV-T)}\ \frac{pct_{i+1} \ldots pct_n \vdash b_i : pct_i \quad \forall i \in 0..n}{\vdash \sigma : \Delta} \quad \begin{array}{l} \sigma = b_0 \ldots b_n \\ \Delta = pct_0 \ldots pct_n \end{array}$$

$\Delta \vdash b : pct$

$$\text{(BASIC-PARTIAL-T)}\ \frac{pct \cdot \Delta \vdash b_i : pct_i \quad \forall i \in 1..n}{\Delta \vdash b : pct} \quad \begin{array}{l} b = \_\,\{\_\_\,c_1 = b_1 \ldots c_n = b_n\} \\ pct = c_1{:}pct_1 \ldots c_n{:}pct_n \end{array}$$

$$\text{(BASIC-T)}\ \frac{\begin{array}{c} ct \cdot \Delta \vdash k : kt \\ ct \cdot \Delta \vdash \overline{d} : \overline{dt} \end{array}}{\Delta \vdash b : ct} \quad \begin{array}{l} \lambda \neq E \text{ if } \overline{dt}(i) = \texttt{abstract}\ \_ \text{ for some } i \\ b = \mu\ \texttt{implements}\ \overline{C}\ \{k\ \overline{d}\} \\ ct = [\mu \mid \overline{C} \mid kt \mid \overline{dt}]^{\lambda} \\ \Delta \vdash ct \end{array}$$

$\Delta \vdash k : kt$

$$\text{(CONS-T)}\ \frac{\begin{array}{c} \Delta; x_1{:}T'_1, \ldots, x_n{:}T'_n \vdash^{\star} e_i : T''_i \quad \forall i \in 1..n \\ \Delta \vdash T_i \leq T''_i \quad \forall i \in 1..n \end{array}}{\Delta \vdash kh\,\{\overline{fe}\} : T'_1 \ldots T'_n} \quad \begin{array}{l} \overline{fe} = \texttt{this.}f_1 = e_1\texttt{;} \ldots \texttt{this.}f_k = e_k\texttt{;} \\ \mathsf{exists}(\Delta, T'_i) \quad \forall i \in 1..n \\ kh = \texttt{constructor}\,(T'_1\ x_1, \ldots, T'_n\ x_n) \\ \mathsf{defFields}(\Delta, \Lambda) = f_1{:}T_1, \ldots, f_k{:}T_k \end{array}$$

$\Delta \vdash d : dt$

$$\text{(FIELD-T)}\ \frac{}{\Delta \vdash (\mu\ T\ f\texttt{;}) : (f{:}\mu\ T)} \quad \mathsf{exists}(\Delta, T)$$

$$\text{(ABS-METHOD-T)}\ \frac{}{\Delta \vdash (\texttt{abstract}\ mh\texttt{;}) : mt} \quad \begin{array}{l} mt = m{:}\texttt{abstract}\ T_1 \ldots T_n \rightarrow T_0 \\ mh = T_0\ m(T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n \end{array}$$

$$\text{(METHOD-T)}\ \frac{\begin{array}{c} \Delta; \texttt{this}{:}\Lambda, x_1{:}T_1, \ldots, x_n{:}T_n \vdash^{\star} e : T \\ \Delta \vdash T \leq T_0 \end{array}}{\Delta \vdash mh\,\{\texttt{return}\ e\texttt{;}\} : mt} \quad \begin{array}{l} mt = m{:}\epsilon\ T_1 \ldots T_n \rightarrow T_0 \\ mh = T_0\ m(T_1\ x_1, \ldots, T_n\ x_n) \\ \mathsf{exists}(\Delta, T_i) \quad \forall i \in 0..n \end{array}$$

$$\text{(CLASS-T)}\ \frac{\Delta \vdash cv : ct}{\Delta \vdash (c = cv) : (c{:}ct)}$$

Figure 6.8: METAFJIG$_\star$ typing rules for environments and basic classes

$\Delta; \Gamma \vdash^{L} e : T$

$$\text{(VAR-T)}\ \frac{}{\Delta; \Gamma \vdash^{L} x : T}\ \Gamma(x) = T \qquad \text{(FIELD-ACCESS-T)}\ \frac{\Delta; \Gamma \vdash^{L} e : C}{\Delta; \Gamma \vdash^{L} e.f : T[\mathsf{from}\ C]}\ \mathsf{mType}(\Delta, C, f) = T$$

$$\text{(INVK-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash^{L} e : C \\ \Delta; \Gamma \vdash^{L} \overline{e} : \overline{T}' \\ \Delta \vdash \overline{T} \leq (\overline{T}'[\mathsf{from}\ C])\end{array}}{\Delta; \Gamma \vdash^{L} e.m\,(\overline{e}) : T[\mathsf{from}\ C]}\ \mathsf{mType}(\Delta, C, m) = \overline{T} \to T$$

$$\text{(NEW-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash^{L} \overline{e} : \overline{T}' \\ \Delta \vdash \overline{T} \leq (\overline{T}'[\mathsf{from}\ C])\end{array}}{\Delta; \Gamma \vdash^{L} \mathbf{new}\ C\,(\overline{e}) : C}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, C) \\ \mathsf{isE}(\Delta, \Lambda)\ \text{implies}\ \mathsf{isE}(\Delta, C) \\ \mathsf{kType}(\Delta, C) = \overline{T}'\end{array}$$

$$\text{(OBJ-T)}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash^{L} e_i : T_i' \quad \forall i \in 1..n \\ \Delta \vdash T_i \leq (T_i'[\mathsf{from}\ C]) \quad \forall i \in 1..n\end{array}}{\Delta; \Gamma \vdash^{L} C\,(\mathbf{this}.f_1 = e_1\mathbf{;}\ \ldots\ \mathbf{this}.f_n = e_n\mathbf{;}) : C}\ \begin{array}{l}\mathsf{nonAbs}(\Delta, C) \\ \mathsf{isE}(\Delta, \Lambda)\ \text{implies}\ \mathsf{isE}(\Delta, C) \\ \mathsf{defFields}(\Delta, C) = f_1{:}\epsilon\ T_1, \ldots, f_n{:}\epsilon\ T_n\end{array}$$

$$\text{(BASIC-META-T}\star\text{)}\ \frac{\Delta \vdash cv : ct}{\Delta; \Gamma \vdash^{\star} cv : \mathbf{class}} \qquad \text{(BASIC-META-T0)}\ \frac{}{\Delta; \Gamma \vdash^{0} cv : \mathbf{class}}\ \Delta \vdash ct^{cv}$$

Figure 6.9: METAFJIG$_\star$ typing rules for expressions 1/2

class values: weak type system provide rule (BASIC-E-T0), and only well-formedness of the extracted type is required, while strong type system provide rule (BASIC-E-T$\star$), and requires the class value to be (strongly) well-typed.

All the notations are the same as in FJIG$_\star$, except $\mathsf{exists}(\Delta, T)$, which holds if either T has a standard class type in Δ, or is a primitive type. Formally:
$\mathsf{exists}(\Delta, T)$ iff $\mathsf{cType}(\Delta, T) = ct$ or $T \in \{\texttt{class}, \texttt{iname}, \texttt{path}, \texttt{cpath}\}$

Figure 6.10 provides the trivial rules for typechecking literals and class composition operators.

Subtyping and class type well formedness are exactly as in FJIG$_\star$, so they are omitted here.

6.5 Checked compile-time execution

Recall that in METAFJIG$_1$ a *meta-program* is a sequence of class declarations where arbitrary expressions, rather than basic classes, are associated to class names. A meta-program can be reduced to a program by *compile-time execution*. Analogously, in METAFJIG$_\star$, a meta-application is an expression where, in any inner class declaration, arbitrary expressions, rather than class values, are associated to class names, and a meta-application can be reduced to an application, that is, a class value, by compile-time execution. This is formally modelled by the following rule:

$$\text{(UNCHECKED-META-RED)}\ \frac{e \xrightarrow[\sigma]{+} e'}{\mathcal{E}^{\mathrm{c}}[\![e]\!] \Rightarrow \mathcal{E}^{\mathrm{c}}[\![e']\!]}\ \mathsf{enclosing}(e, \mathcal{E}^{\mathrm{c}}) = \sigma$$

Here, $\mathcal{E}^{\mathrm{c}}[\![e]\!]$ is an expression where e occurs as right-hand side of a class declaration at any inner level, as we have formally defined in Figure 6.2, and $\mathsf{enclosing}(e, \mathcal{E}^{\mathrm{c}})$ returns the stack of basic classes enclosing the hole in $\mathcal{E}^{\mathrm{c}}$. Formally:

$$\begin{array}{l}
\mathsf{enclosing}(e, \Box) = \epsilon \\
\mathsf{enclosing}(e, \mathcal{E}[\![\mathcal{B}]\!]) = \mathsf{enclosing}(e, \mathcal{B}) \\
\mathsf{enclosing}(e,\ ch\ \{\, k\ \overline{d}\ c = \Box \,\}) = ch\ \{\, k\ \overline{d}\ c = e \,\} \\
\mathsf{enclosing}(e, \mathcal{B}) = \mathsf{enclosing}(e, \mathcal{B}') \cdot \mathcal{B}[\![e]\!] \quad \text{with } \mathcal{B} = \begin{cases} ch\ \{\, k\ \overline{d}\ c = \mathcal{E}[\![\mathcal{B}']\!] \,\} \\ ch\ \{\, k\ \overline{d}\ mh\ \{\, \texttt{return}\ \mathcal{E}[\![\mathcal{B}']\!]\texttt{;} \,\}\,\} \\ ch\ \{\, kh\ \{\, \overline{fe}\ \texttt{this}.f = \mathcal{E}[\![\mathcal{B}']\!]\texttt{;} \,\}\ \overline{d} \,\} \end{cases}
\end{array}$$

However, exactly as in METAFJIG$_1$, soundness of compile-time execution is not guaranteed, that is, reduction could get stuck, or produce as right-hand side of a class declaration a value different from a class value, or a class value denoting an ill-typed class. To avoid this, we extend to METAFJIG$_\star$ checked compile-time execution, as formally defined in Figure 6.11.

In the rules we use the *clientship* $\xrightarrow[\sigma]{\mathsf{client}}$ and *dependency* $\xrightarrow[\sigma]{\mathsf{dep}}$ relations defined in Figure 6.12;

$\Delta; \Gamma \vdash^{L} e : T$

$$\text{(COPY-T)}\ \frac{\Delta; \Gamma \vdash^{L} e : \mathbf{cpath}}{\Delta; \Gamma \vdash^{L} \mathbf{copy}\ e : \mathbf{class}}\ \mathsf{exists}(\Delta, C)$$

$$\text{(SUM-T)}\ \frac{\Delta; \Gamma \vdash^{L} e_1 : \mathbf{class} \quad \Delta; \Gamma \vdash^{L} e_2 : \mathbf{class}}{\Delta; \Gamma \vdash^{L} e_1\ [+]\ e_2 : \mathbf{class}}$$

$$\text{(RESTRICT-T)}\ \frac{\Delta; \Gamma \vdash^{L} e : \mathbf{class} \quad \Delta; \Gamma \vdash^{L} e' : \mathbf{iname} \quad \Delta; \Gamma \vdash^{L} e'' : \mathbf{path}}{\Delta; \Gamma \vdash^{L} e\,[\mathbf{restrict}\ e'\ \mathbf{in}\ e'']\ : \mathbf{class}}$$

$$\text{(ALIAS-T)}\ \frac{\Delta; \Gamma \vdash^{L} e : \mathbf{class} \quad \Delta; \Gamma \vdash^{L} e^s : \mathbf{iname} \quad \Delta; \Gamma \vdash^{L} e' : \mathbf{path} \quad \Delta; \Gamma \vdash^{L} e^t : \mathbf{iname}}{\Delta; \Gamma \vdash^{L} e\,[\mathbf{alias}\ e^s\ \mathbf{to}\ e^t\ \mathbf{in}\ e']\ : \mathbf{class}}$$

$$\text{(CLASS-ALIAS-T)}\ \frac{\Delta; \Gamma \vdash^{L} e : \mathbf{class} \quad \Delta; \Gamma \vdash^{L} e^t : \mathbf{path} \quad \Delta; \Gamma \vdash^{L} e^s : T}{\Delta; \Gamma \vdash^{L} e\,[\mathbf{alias}\ e^t\ \mathbf{to}\ e^s]\ : \mathbf{class}}\ T \in \{\mathbf{path}, \mathbf{cpath}\}$$

$$\text{(REDIRECT-T)}\ \frac{\Delta; \Gamma \vdash^{L} e : \mathbf{class} \quad \Delta; \Gamma \vdash^{L} e^s : \mathbf{iname} \quad \Delta; \Gamma \vdash^{L} e' : \mathbf{path} \quad \Delta; \Gamma \vdash^{L} e^t : \mathbf{iname}}{\Delta; \Gamma \vdash^{L} e\,[\mathbf{redirect}\ e^s\ \mathbf{of}\ e'\ \mathbf{to}\ e^t]\ : \mathbf{class}}$$

$$\text{(CLASS-REDIRECT-T)}\ \frac{\Delta; \Gamma \vdash^{L} e : \mathbf{class} \quad \Delta; \Gamma \vdash^{L} e^s : \mathbf{path} \quad \Delta; \Gamma \vdash^{L} e^t : T}{\Delta; \Gamma \vdash^{L} e\,[\mathbf{redirect}\ e^s\ \mathbf{to}\ e^t]\ : \mathbf{class}}\ T \in \{\mathbf{path}, \mathbf{cpath}\}$$

$$\text{(INAME-T)}\ \frac{}{\Delta; \Gamma \vdash^{L} n : \mathbf{iname}} \qquad \text{(CLASS-PATH-T)}\ \frac{}{\Delta; \Gamma \vdash^{L} C : \mathbf{cpath}}\ \mathsf{exists}(\Delta, C)$$

$$\text{(PATH-T)}\ \frac{}{\Delta; \Gamma \vdash^{L} \pi : \mathbf{path}}$$

Figure 6.10: METAFJIG$_\star$ typing rules for expressions 2/2

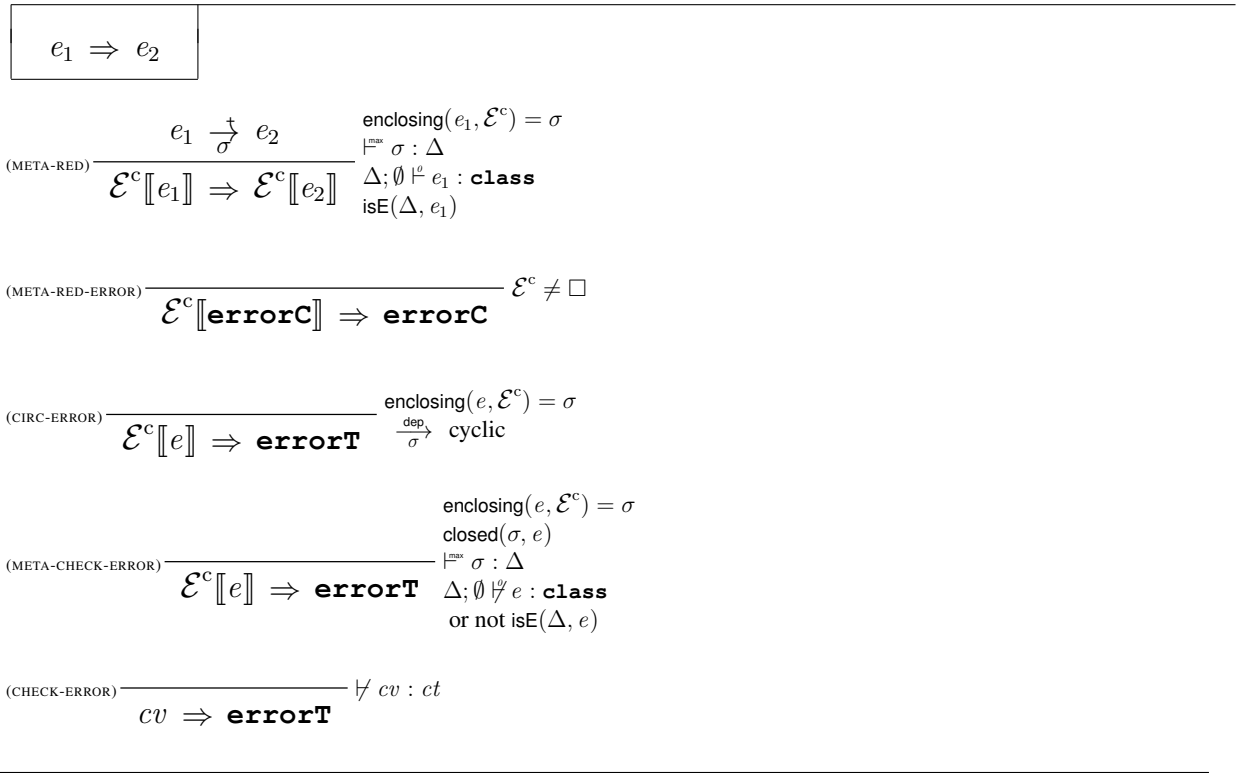

Figure 6.11: METAFJIG$_\star$ checked compile-time execution

those concept are analogously to the corresponding concepts in METAFJIG$_1$. However, auxiliary function cnames* is adapted to manage nested classes.

That is, cnames$^\star(e)$ denotes the set of all the class paths that appear in e that have an external representation. For instance, if e is

```
new C().m({
  H = {}
  outer.D n(H h){return new outer.D();}
  })
```

then only C, D $\in$ cnames$^\star(e)$.

The formal definition of cnames$^\star(b)$ is straightforward and given in Figure 6.12.

Note how a class paths depends to the (unique) prefix C'' denoting an expression that is not yet reduced to a basic class.

closed(σ, e) holds if, for all (clients of) class paths directly occurring in e, either a class value is available in σ or we know that no class value will be ever available, formally:
closed(σ, e) iff, for all $C \in \mathsf{cnames}^0(e)$, $C \xrightarrow[\sigma]{\text{client}} C'$ implies
$\begin{cases} \text{either } \mathsf{cBody}(\sigma, C') = cv \\ \text{or } C' = C''.c.\_, \mathsf{cBody}(\sigma, C'') = \_\ \{\ \_\ \overline{d}\ \} \text{ and } c \notin \mathsf{dom}(\overline{d}) \end{cases}$

hence we can determine whether or not e is a well-typed expression of type **class** w.r.t. σ.

In (META-RED), a meta-reduction step can only be performed if the expression to be reduced is a well-typed meta-expression of type **class** w.r.t. the available type information, and, moreover, the expression is *executable*, that is, only instantiates executable classes. Note that available type information can only increase during compile-time execution, hence once a class can be typechecked, such type will not change in the following steps of the compile time execution. This make caching the result a very effective optimization method.

We use the following notations:

- $\vdash^{\max} \sigma : \Delta$ denotes the maximal Δ such that $\vdash \sigma : \Delta$, where the ordering relation between type environments is formally defined here:
  $(pct_1 \dots pct_n) \geq (pct'_1 \dots pct'_n)$ iff $pct_1 \geq pct'_1 \dots pct_n \geq pct'_n$
  $[\mu \mid \overline{c} \mid kt \mid \overline{d}\ c_1 = ct_1 \dots c_n = ct_n] \geq c_1 = pct_1 \dots c_n = pct_n$ iff $ct_1 \geq pct_1 \dots ct_n \geq pct_n$
  $[\mu \mid \overline{c} \mid kt \mid \overline{d}]^E \geq [\mu \mid \overline{c} \mid kt \mid \overline{d}]^\lambda$
- isE(σ, e) holds if all the class instantiated inside e are executable. Formally:
  isE(σ, e) holds iff for all C such that $e = \mathcal{E}[\![\textbf{new}\ C\ (\_)\ ]\!]$ or $e = \mathcal{E}[\![C\ (\_)\ ]\!]$ isE(σ, C) holds.

The other rules models abnormal termination:

$C_1 \xrightarrow[\sigma]{\text{client}} C_2$

$$\text{(DIRECT-C)}\ \frac{\text{cBody}(\sigma, C)_1 = b \quad C_2 \in \text{cnames}^\star(b)}{C_1 \xrightarrow[\sigma]{\text{client}} C_2} \qquad \text{(REFL-C)}\ \frac{}{C \xrightarrow[\sigma]{\text{client}} C}$$

$$\text{(TRANS-C)}\ \frac{C_1 \xrightarrow[\sigma]{\text{client}} C_2 \quad C_2 \xrightarrow[\sigma]{\text{client}} C_3}{C_1 \xrightarrow[\sigma]{\text{client}} C_3}$$

$\text{cnames}^\star(\mu\ \texttt{implements}\ \overline{C}\ \{\, k\ \overline{d}\,\}) = \text{removeOuter}(\overline{C} \cup \text{cnames}^\star(k) \cup \text{cnames}^\star(\overline{d}))$
$\text{cnames}^\star(\texttt{constructor}\,(\,T_1\ x_1 \ldots T_n\ x_n\,)\ \{\,\overline{fe}\,\}) = \text{cnames}^\star(\,T_1 \ldots T_n) \cup \text{cnames}^\star(\overline{fe})$
$\text{cnames}^\star(\texttt{this.}f = e\texttt{;}) = \text{cnames}^\star(e)$
$\text{cnames}^\star(\mu\ C\ f\texttt{;}) = C$
$\text{cnames}^\star(\texttt{abstract}\ T_0\ m(\,T_1\ x_1 \ldots T_n\ x_n)\texttt{;}) = \text{cnames}^\star(\,T_0 \ldots T_n)$
$\text{cnames}^\star(\,T_0\ m(\,T_1\ x_1 \ldots T_n\ x_n)\{\,\texttt{return}\ e\texttt{;}\,\}) = \text{cnames}^\star(\,T_0 \ldots T_n) \cup \text{cnames}^\star(e)$
$\text{cnames}^\star(\,c = e) = \text{cnames}^\star(e)$
$\text{cnames}^\star(x) = \emptyset$
$\text{cnames}^\star(e.f) = \text{cnames}^\star(e)$
$\text{cnames}^\star(e.m\,(\,\overline{e}\,)\,) = \text{cnames}^\star(e) \cup \text{cnames}^\star(\overline{e})$
$\text{cnames}^\star(\texttt{new}\ C\,(\,\overline{e}\,)\,) = C \cup \text{cnames}^\star(\overline{e})$
$\text{cnames}^\star(\,C\,(\,\overline{e}\,)\,) = C \cup \text{cnames}^\star(\overline{e})$
$\text{cnames}^\star(\,C) = C$
$\text{cnames}^\star(\,T) = \emptyset$ if $T \neq c$
...
$\text{removeOuter}(\texttt{outer.}C) = C$
$\text{removeOuter}(\pi) = \emptyset$

$C_1 \xrightarrow[\sigma]{\text{dep}} C_2$

$$\text{(DIRECT-D)}\ \frac{C_2 \xrightarrow[\sigma]{\text{client}} C_3\texttt{.\_} \quad \text{cBody}(\sigma, C)_1 = e \quad C_2 \in \text{cnames}^0(e) \quad \text{cBody}(\sigma, C_3) \neq b}{C_1 \xrightarrow[\sigma]{\text{dep}} C_3} \qquad \text{(TRANS-D)}\ \frac{C_1 \xrightarrow[\sigma]{\text{dep}} C_2 \quad C_2 \xrightarrow[\sigma]{\text{dep}} C_3}{C_1 \xrightarrow[\sigma]{\text{dep}} C_3}$$

$\text{cnames}^0(x) = \emptyset$
$\text{cnames}^0(e.f) = \text{cnames}^0(e)$
$\text{cnames}^0(e.m\,(\,\overline{e}\,)\,) = \text{cnames}^0(e) \cup \text{cnames}^0(\overline{e})$
$\text{cnames}^0(\texttt{new}\ C\,(\,\overline{e}\,)\,) = C \cup \text{cnames}^0(\overline{e})$
$\text{cnames}^0(\,C\,(\,\overline{e}\,)\,) = C \cup \text{cnames}^0(\overline{e})$
$\text{cnames}^0(\texttt{\_ implements}\ \overline{C}\ \{\texttt{\_}\,\overline{fd}\ \overline{md}\ \overline{cd}\,\}) = \text{removeOuter}(\overline{C} \cup \text{cnames}^0(\overline{cd}))$
$\text{cnames}^0(\,c = b) = \text{cnames}^0(b)$
$\text{cnames}^0(\,C) = C$
...

Figure 6.12: METAFJIG$_\star$ clientship and dependency relations

- (META-RED-ERROR) manage the case one expression reduces to **errorC**,
- (CIRC-ERROR) is applied when we detect a cyclic dependency[4],
- (META-CHECK-ERROR) represent a compilation terminated by a typechecking error **errorT**. This rules requires closed(σ, e), that is, all the class paths that are required to safely execute e are already compiled, but e is is either not well-typed or not executable.
- (CHECK-ERROR) can be applied at the end of the checked compile time execution, ensuring the result to be a well-typed class value. Note that in checked compile time execution of METAFJIG$_1$ rule (CHECK-ERROR) could be executed in any moment. Indeed here we typecheck classes only on need.

Reduction examples The following examples clarify the details of checked compile-time execution for METAFJIG$_\star$.

Example 1 First we give an example of successful reduction.

```
{
  A = { class m1(class x){return {};} }
  B = new A().m1(
    {
      C = copy outer.A
      D = new C().m1(copy E)
      E = { abstract int f; }
      F = { int m(){return new E().k;} }
    })
}
```

reduces by two applications of (META-RED) to

```
{
  A = { class m1(class x){return {};} }
  B = new A().m1(
    {
      C = { class m1(class x){return {};} }
      D = {}
      E = { abstract int f; }
      F = { int m(){return new E().k;} }
    })
}
```

[4]For example `class C = new C().m()`.

First we reduce **copy outer.**A and then **new** C().m1(**copy** E). Note that class E is not executable, however it is safe to perform the operation **copy** E. By another application of (META-RED) we obtain

```
{
  A = { class m1(class x){return {};} }
  B = {}
}
```

Note how the code of nested classes C, D, E and F has disappeared. Moreover, the code of F is ill-typed, since E.k does not exist. Formally,
$\Delta;\emptyset \vdash^{o}$ **new** A().m1({...}) : **class** holds, while
$\Delta;\emptyset \vdash^{\star}$ **new** A().m1({...}) : **class** does not.

Example 2 The second example shows a case when checked compile-time execution terminates with an **errorT** caused by a not executable class. The only difference w.r.t. the first example is in the nested classes declared inside the declaration of class B.

```
{
  A = { class m1(class x){return {};} }
  B = new A().m1(
    {
      C = copy outer.A [+] { abstract int f; }
      D = new C().m1({})
    })
}
```

By an application of (META-RED) we obtain

```
{
  A = { class m1(class x){return {};} }
  B = new A().m1(
    {
      C = {
        class m1(class x){return {};}
        abstract int f;
      }
      D = new C().m1({})
    })
}
```

that reduces to **errorT** by (META-CHECK-ERROR). Class C is not executable, hence isE(Δ, **new** C().m1({})) does not hold.

Example 3 The last example shows a case when checked compile-time execution terminates with an *error* caused by an ill-typed class. Again the only difference w.r.t. the first example is in the nested classes declared inside the class initialization expression for class `B`.

```
{
  A = { class m1(class x){return {};} }
  B = new A().m1(
    {
      C = copy outer.A [+] { int m2(){return true;} }
      D = new C().m1({})
    })
}
```

By application of (META-RED) we obtain

```
{
  A = { class m1(class x){return {};} }
  B = new A().m1(
    {
      C = {
        class m1(class x){return {};}
        int m2(){return true;}
      }
      D = new C().m1({})
    })
}
```

Now we reduce to error by (META-CHECK-ERROR). Class `C` does not typecheck, hence $\Delta;\emptyset \vdash^{o}$ `new C().m1({}) : class` does not hold (since not $\mathsf{exists}(\Delta, \texttt{C})$).

6.6 Results

Properties of reduction The type system is sound, as formally stated by the following standard theorem.

Theorem 38 (METAFJIG$_\star$ soundness)**.** *If* $\vdash \sigma : \Delta$, $\Delta;\emptyset \vdash^{o} e : T$, $\mathsf{isE}(\sigma, e)$ *and* $e_1 \xrightarrow[p]{}{}^{*} e_2$ *then either* e_2 *is a value, or* e_2 *is error, or* $e_2 \xrightarrow[\sigma]{} \_$.

As usual, soundness can be derived from progress and subject reduction properties.

Theorem 39 (METAFJIG$_\star$ progress)**.** *If* $\vdash \sigma : \Delta, \Delta;\emptyset \vdash^{o} e : T$ *and* $\mathsf{isE}(\sigma, e)$ *then either* e *is a value, or* e *is error, or* $e \xrightarrow[\sigma]{} \_$.

Theorem 40 (METAFJIG$_\star$ subject reduction)**.** *If* $\vdash \sigma : \Delta$, $\Delta; \emptyset \vdash^{o} e : T_1$ *and* $e_1 \xrightarrow[\sigma]{} e_2$*, then* $\Delta; \emptyset \vdash^{o} e_2 : T_2$ *and* $\Delta \vdash T_2 \leq T_1$.

The proofs are a straightforward extension of those for FJIG$_\star$. Indeed, METAFJIG$_\star$ programs can be roughly seen as FJIG$_\star$ programs with four primitive types.

Properties of checked compile-time execution As in METAFJIG$_1$ we can state two significant properties for METAFJIG$_\star$ checked compile-time execution: *soundness* (Theorem 41) and *meta-level soundness* (Theorem 42). The former means that checked-compile time execution never gets stuck, while the latter allows the programmer to safely use compiled libraries.

A *value* w.r.t. checked compile-time execution is a well-typed class value cv, and an *error* is either **`errorC`** or **`errorT`**.

Theorem 41 (METAFJIG$_\star$ soundness w.r.t. compilation)**.** *If* $e_1 \stackrel{\star}{\Rightarrow} e_2$*, and* e_1 *is a well-formed top-level expression, then either* e_2 *is a value or* e_2 *is an error or* $e_2 \stackrel{\star}{\Rightarrow} \_$.

Proof. The proof reduces to verify that there is always a rule applicable to a well-formed top-level expression e if e is neither a value nor an error.

If e is neither a value nor an error, then e is necessarily of the form $\mathcal{E}^{\mathrm{c}}[\![e']\!]$.

Set $\sigma = \mathsf{enclosing}(e', \mathcal{E}^{\mathrm{c}})$, and consider the graph denoted by relation $\xrightarrow[\sigma]{\mathsf{dep}}$. If the graph is not acyclic, then $C \xrightarrow[\sigma]{\mathsf{dep}} C$ and (CIRC-ERROR) is applicable.

Otherwise, if e is neither a value, nor **`errorC`**, nor **`errorT`**, and the graph w.r.t. σ is acyclic, then there is a C that is a sink node in the graph.

In this case, e is of the form $\mathcal{E}^{\mathrm{c}\prime}[\![e'']\!]$, set $\sigma' = \mathsf{enclosing}(e'', \mathcal{E}^{\mathrm{c}})'$ and $e'' = \mathsf{cBody}(\sigma, C)$.

This means that there is no C' such that $C \xrightarrow[\sigma]{\mathsf{dep}} C'$, hence $\mathsf{closed}(\sigma', e'')$ holds, and one of rules (META-RED), (META-RED-ERROR) or (META-CHECK-ERROR) is applicable. □

The next theorem use notation $b_1 \subseteq b_2$, with intuitive meaning that every class inside b_2 is exactly as in b_1, but b_1 can be richer. Formally:

- $b_1 \subseteq b_2$ holds iff $\mathsf{cBody}(b_2, C) = ch\ \{\, k\ \overline{d}\,\}$ implies $\mathsf{cBody}(b_1, C) = ch\ \{\, k\ \overline{d}\ \overline{cd}\,\}$.

Theorem 42 (METAFJIG$_\star$ meta-level soundness)**.** *If* $\vdash cv : ct$*,* $cv \subseteq b$ *and* $b \stackrel{\star}{\Rightarrow}$ **`errorT`** *then* $\exists C \in \mathsf{dom}(b) \setminus \mathsf{dom}(cv)$ *such that* $ct[\mathsf{in}\ C]; \emptyset \not\vdash^{\star} \mathsf{cBody}(b, C) :$ **`class`**.

Proof. The thesis follows from the following property:
if $\vdash \sigma : \Delta$, $\Delta; \emptyset \vdash^{\star} e : T$ and $e \xrightarrow[\sigma]{\star} b$, then $\Delta \vdash cv : ct$. By (META-BASE-T$\star$) this can be derived from progress and subject reduction properties. □

Lemma 43 states that by (successfully) applying a composition operator to strongly well-typed classes we always get a strongly well-typed class.

Lemma 43. *If* $\vdash \sigma : \Delta$, $\Delta; \emptyset \vdash^{\star} cv_1 : \texttt{class}$ *and* $\Delta; \emptyset \vdash^{\star} cv_2 : \texttt{class}$ *then if*

1. $cv_1\,[+]\,cv_2 \underset{\sigma}{\rightarrow} cv$ *or*
2. $cv_1\,[\texttt{restrict}\,i\,\texttt{in}\,\pi] \underset{\sigma}{\rightarrow} cv$ *or*
3. $cv_1\,[\texttt{redirect}\,i_1\,\texttt{of}\,\pi\,\texttt{to}\,i_2] \underset{\sigma}{\rightarrow} cv$ *or*
4. $cv_1\,[\texttt{alias}\,i_1\,\texttt{to}\,i_2\,\texttt{in}\,\pi] \underset{\sigma}{\rightarrow} cv$ *or*
5. $cv_1\,[\texttt{redirect}\,\pi^s\,\texttt{to}\,\pi^t] \underset{\sigma}{\rightarrow} cv$ *or*
6. $cv_1\,[\texttt{alias}\,\pi^s\,\texttt{to}\,\pi^t] \underset{\sigma}{\rightarrow} cv$ *or*
7. $cv_1\,[\texttt{alias}\,C^s\,\texttt{to}\,\pi^t] \underset{\sigma}{\rightarrow} cv$ *or*
8. $cv_1\,[\texttt{redirect}\,\pi^s\,\texttt{to}\,C^t] \underset{\sigma}{\rightarrow} cv$

then $\Delta; \emptyset \vdash^{\star} cv : \texttt{class}$.

Proof. This lemma is analogous to the subject reduction for flattening of FJIG$_\star$. □

Theorem 44 (METAFJIG$_\star$ meta-level progress)**.** *If* $\vdash \sigma : \Delta$, $\mathsf{isE}(\sigma, e)$ *and* $\Delta; \emptyset \vdash^{\star} e : T$ *then either* e *is a value or* e *is* `errorC` *or* $e \underset{\sigma}{\rightarrow} \_$.

Proof. Since $\Delta; \emptyset \vdash^{\star} e : T$ implies $\Delta; \emptyset \vdash^{o} e : T$ the thesis follows by Theorem 39. □

Theorem 45 (METAFJIG$_1$ meta-level subject reduction*)**.** *If* $\vdash \sigma : \Delta$, $\Delta; \emptyset \vdash^{\star} e : T_1$ *and* $e_1 \underset{\sigma}{\rightarrow} e_2$ *then* $\Delta; \emptyset \vdash^{\star} e_2 : T_2$ *and* $\Delta \vdash T_2 \leq T_1$.

Proof. Analogously to Theorem 40 but with application of Lemma 43. □

6.7 Implementation

As already mentioned, our approach allows a modular implementation, relying on typechecking and execution of the conventional language. This is effectively shown by our prototype compiler, which is built on top of the standard Java compiler and virtual machine. In this section, we describe in some more detail how this modular implementation works.

We will use the metavariables e_J and p_J for plain Java expressions and programs, respectively.

First of all, in order to reuse the standard Java compiler, it is clear that we need a translation step which transforms MetaFJig$_\star$ expressions e and class values cv into plain Java expressions e_J and programs p_J, respectively.

Notably, primitive types **`class`**, **`iname`**, **`path`** and **`cpath`** are not available in plain Java, but are encoded by classes `MFJClass`, `MFJIName`, `MFJPath` and `MFJCPath`. The first offers a method for each composition operator, and the last class offers a **`copy`** method, that retrieves the corresponding class value from the Java program. We denote by $\widetilde{cv}$, $\widetilde{i}$, $\widetilde{\pi}$ and $\widetilde{C}$ the values of type `MFJClass`, `MFJIName`, `MFJPath` and `MFJCPath` which are the Java representation of a class value cv, a name i, a path π and a class path C, respectively. We omit the details of this representation, which are not relevant, and only assume that it is invertible.

More precisely, the compilation uses two different translation functions, formally defined in Figure 6.13: $[\![C, e]\!]$ and $[\![\Delta, \sigma]\!]$. They are triggered when rule (META-RED) is applied, and translates expressions e occurring in C into the corresponding e_J, and class values of σ whose class type is available in Δ into the corresponding p_J, respectively.

We model a MetaFJig$_\star$ class with a Java interface and a Java class. More precisely, this is true for classes that are executable, while for the others only the interface is produced.

The two translations ensure that the following properties hold:

- $\Delta[\mathsf{in}\ C]; \emptyset \vdash^{\varrho} e$: **`class`** implies $\Delta'; \emptyset \vdash [\![C, e]\!]$: `MFJClass` for some Δ', where $\Delta; \emptyset \vdash e_J : T$ is the typing judgement for Java expressions.
- $\vdash \sigma : \Delta$ implies $\vdash [\![\Delta, \sigma]\!]$ where $\vdash p_J$ is the typing judgement for Java programs.

Figure 6.14 contains the implementation-oriented version of the rule (META-RED) of checked compile-time execution. The rule is analogous to the one in Figure 6.11, except that the Java (bytecode) expression $[\![\Lambda, e]\!]$ is evaluated by the JVM. We use the $\xrightarrow{+}$ arrow to model that JVM execution continues until getting a final result or an exception.

We use the notations $\mathcal{C}$ and $\mathcal{I}$ with the following meaning: for each MetaFJig$_\star$ class path C' occurring C w.r.t. the execution expression, $\mathcal{I}^C_{C'}$ and $\mathcal{C}^C_{C'}$ are a Java interface name and class name, respectively, where the functions $\mathcal{I}$ and $\mathcal{C}$ have disjoint codomains. In practice such name is obtained by encoding in valid Java identifiers the result of the $C'[\mathsf{from}\ C]$ operation.

Translation for expressions

$$
\begin{array}{lcl}
[\![C, \texttt{new}\ C'\,(\overline{e})\,]\!] & = & \texttt{new}\,\mathcal{C}^C_{C'}\,([\![C, \overline{e}]\!]) \\
[\![C, e.m\,(\overline{e})\,]\!] & = & [\![C, e]\!].m\,([\![C, \overline{e}]\!]) \\
[\![C, e.f]\!] & = & [\![C, e]\!].f \\
[\![C, cv]\!], [\![C, i]\!], [\![C, \pi]\!] & = & \widetilde{cv}\ ,\widetilde{i}\ ,\widetilde{\pi} \\
[\![C, \texttt{copy}\ e]\!] & = & [\![C, e]\!]\texttt{.copyClass()} \\
[\![C, e_1\,\texttt{[+]}\,e_2]\!] & = & [\![C, e_1]\!]\texttt{.sum(}[\![C, e_2]\!]\texttt{)} \\
[\![C, e_1\,\texttt{[}\texttt{redirect}\ e^s\ \texttt{of}\ e_2\ \texttt{to}\ e^t\texttt{]}]\!] & = & [\![C, e_1]\!]\texttt{.redirectM(}[\![C, e_2]\!]\texttt{,}[\![C, e^s]\!]\texttt{,}[\![C, e^t]\!]\texttt{)} \\
[\![C, e_1\,\texttt{[}\texttt{redirect}\ e_2\ \texttt{to}\ e_3\texttt{]}]\!] & = & [\![C, e_1]\!]\texttt{.redirectC(}[\![C, e_2]\!]\texttt{,}[\![C, e_3]\!]\texttt{)} \\
[\![C, e_1\,\texttt{[}\texttt{alias}\ e^s\ \texttt{to}\ e^t\ \texttt{in}\ e_2\texttt{]}]\!] & = & [\![C, e_1]\!]\texttt{.aliasM(}[\![C, e_2]\!]\texttt{,}[\![C, e^s]\!]\texttt{,}[\![C, e^t]\!]\texttt{)} \\
[\![C, e_1\,\texttt{[}\texttt{alias}\ e_2\ \texttt{to}\ e_3\texttt{]}]\!] & = & [\![C, e_1]\!]\texttt{.aliasc(}[\![C, e_2]\!]\texttt{,}[\![C, e_3]\!]\texttt{)} \\
[\![C, e_1\,\texttt{[}\texttt{restrict}\ e_2\ \texttt{in}\ e_3\texttt{]}]\!] & = & [\![C, e_1]\!]\texttt{.restrictM(}[\![C, e_2]\!]\texttt{,}[\![C, e_3]\!]\texttt{)} \\
[\![C, C']\!] & = & \widetilde{C''} \text{ iff } C'' = C'[\mathsf{from}\ C]
\end{array}
$$

Translation for classes

$$
\begin{array}{lcl}
[\![\Delta, \sigma]\!] & = & [\![C_1, \mathsf{cBody}(\sigma, C_1)]\!]^{\mathcal{I}} \ldots [\![C_n, \mathsf{cBody}(\sigma, C_n)]\!]^{\mathcal{I}} [\![C_1, \mathsf{cBody}(\sigma, C_1)]\!]^{\mathcal{C}} \ldots [\![C_k, \mathsf{cBody}(\sigma, C_k)]\!]^{\mathcal{C}} \\
 & & \text{with } C_1 \ldots C_n = \mathsf{dom}(\Delta) \text{ and } \mathsf{isE}(\Delta, C_i) \text{ iff } i < k
\end{array}
$$

$$
\begin{array}{lcl}
[\![C, cv]\!]^{\mathcal{C}} & = & \texttt{public}\ \mu\ \texttt{class}\ \mathcal{C}^C_C\ \texttt{implements}\ \mathcal{I}^C_C\ \texttt{\{}\ [\![C, k]\!]^{\mathcal{C}}\ [\![C, \overline{fd}]\!]^{\mathcal{C}}\ [\![C, \overline{md}]\!]^{\mathcal{C}}\ \texttt{\}} \\
 & & \text{with } cv = \mu\,\texttt{implements}\,\overline{C}\,\{k\,\overline{fd}\,\overline{md}\,\overline{cd}\} \\
[\![C, cv]\!]^{\mathcal{I}} & = & \\
 & & \texttt{public interface}\ \mathcal{I}^C_C\ \texttt{extends}\ \mathcal{I}^C_{\overline{C}}\ \texttt{\{}\ [\![C, \overline{fd}]\!]^{\mathcal{I}}\ [\![C, \overline{md}]\!]^{\mathcal{I}} \\
 & & \quad \texttt{public static class Static\{} \\
 & & \quad\quad \texttt{public static MFJClass thisRepr=}\widetilde{cv}\texttt{;\}} \\
 & & \texttt{\}} \\
 & & \text{with } cv = \mu\,\texttt{implements}\,\overline{C}\,\{k\,\overline{fd}\,\overline{md}\,\overline{cd}\}
\end{array}
$$

Translation for instance members

$$
\begin{array}{lcl}
[\![C, k]\!]^{\mathcal{C}} & = & \mathcal{C}^C_C\,(\ \mathcal{I}^C_{T_1}\ x_1\ \ldots\ \mathcal{I}^C_{T_n}\ x_n)\{\ f_1\ =\ [\![C, e_1]\!];\ \ldots\ f_k\ =\ [\![C, e_k]\!];\} \\
 & & \text{with } k = \texttt{constructor}\,(\,T_1\,x_1 \ldots T_n\,x_n)\,\{\texttt{this.}f_1 = e_1\texttt{;}\ldots\texttt{this.}f_k = e_k\texttt{;}\} \\
[\![C, T\,f]\!]^{\mathcal{C}} & = & \mathcal{I}^C_T\ f\texttt{;}\ \texttt{public}\ \mathcal{I}^C_T\ f\,\texttt{()}\,\{\texttt{return}\ f\texttt{;}\} \\
[\![C, \texttt{abstract}\ T\,f]\!]^{\mathcal{C}} & = & \texttt{public abstract}\ \mathcal{I}^C_T\ f\,\texttt{();} \\
[\![C, \texttt{abstract}\ T\,m(T_1\,x_1 \ldots T_n\,x_n)\texttt{;}]\!]^{\mathcal{C}} & = & \texttt{public abstract}\ \mathcal{I}^C_T\ m\,(\mathcal{I}^C_{T_1}\ x_1\ \ldots\ \mathcal{I}^C_{T_n}\ x_n\ )\texttt{;} \\
[\![C, T\,m(T_1\,x_1 \ldots T_n\,x_n)\{\texttt{return}\,e\texttt{;}\}]\!]^{\mathcal{C}} & = & \mathcal{I}^C_T\ m\,(\mathcal{I}^C_{T_1}\ x_1\ \ldots\ \mathcal{I}^C_{T_n}\ x_n\ )\{\texttt{return}\ [\![C, e]\!]\ \texttt{;}\} \\
[\![C, \texttt{abstract}\ T\,f]\!]^{\mathcal{I}} & = & \mathcal{I}^C_T\ f\,\texttt{();} \\
[\![C, T\,f]\!]^{\mathcal{I}} & = & \mathcal{I}^C_T\ f\,\texttt{();} \\
[\![C, \texttt{abstract}\ T\,m(T_1\,x_1 \ldots T_n\,x_n)\texttt{;}]\!]^{\mathcal{I}} & = & \mathcal{I}^C_T\ m\,(\mathcal{I}^C_{T_1}\ x_1\ \ldots\ \mathcal{I}^C_{T_n}\ x_n)\texttt{;} \\
[\![C, T\,m(T_1\,x_1 \ldots T_n\,x_n)\{\texttt{return}\,e\texttt{;}\}]\!]^{\mathcal{I}} & = & \mathcal{I}^C_T\ m\,(\mathcal{I}^C_{T_1}\ x_1\ \ldots\ \mathcal{I}^C_{T_n}\ x_n)\texttt{;}
\end{array}
$$

Figure 6.13: METAFJIG$_\star$ to Java translation functions

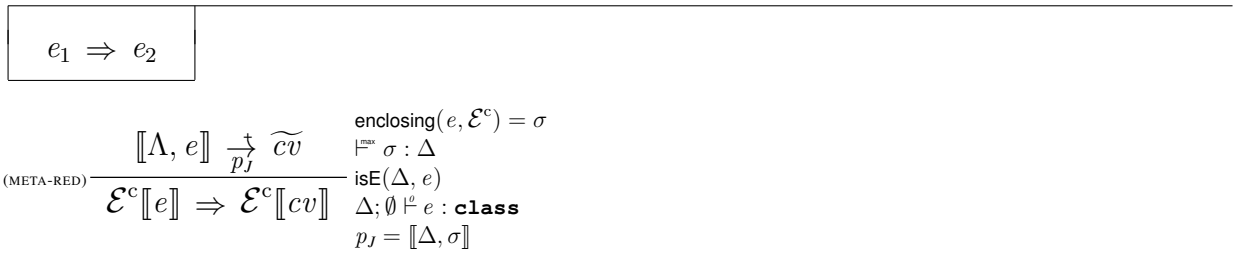

Figure 6.14: Checked compile-time execution for implementation

Chapter 7

Related work

In this chapter provide a survey of the most influential proposals in the following three branches of language design, relevant for this thesis: module/class composition, meta-programming and hierarchical composition.

7.1 Class composition languages

Standard inheritance Traditional class-based object-oriented programming is based on the notions of *class*, *object* and *inheritance*. Classes are templates that define *members*, typically *fields* and *methods*, and can be instantiated to produce objects, which are runtime entities. A new class `H` can be defined by adding new members to an existing class `P`, called *parent*, as shown below in Java syntax:

```
class H extends P {
  decs
}
```

where `decs` denotes a set of member declarations. This mechanism, called (single) inheritance, allows to obtain, roughly, the same effect programmers would get by copying the code of the parent class in the heir, without actually duplicating any code or requiring the source code of the parent class. However, the key advantage is that source code duplication is avoided, and the code of the parent is, instead, found on demand, through a runtime procedure called *method look-up*. In other words, two different semantics of inheritance can be given: *flattening* semantics, that is, by translation into a language with no inheritance, and *direct* semantics, that is, by formalizing dynamic method look-up. These two different ways for expressing the semantics can be applied, in principle, to any class composition language.

Inheritance has been extremely successful and has caused a profound change in the development of software systems, to the point that most mainstream programming languages now embrace the object-oriented paradigm. However, the need of going beyond has been recognized for a long time, leading to a variety of proposals for improving flexibility and expressive power of object-oriented programming. We mention, among others, mixin classes [FKF98, ALZ03], virtual classes [Ern01, EOC06], generic classes as in Java 5, mixin modules [FF98, ALZ06] and other proposals for adding a module/component level [MFH01, ACN02], traits [SDNB03, FR04, LS08b], multimethods [BC97, CMLC06] and aspect-oriented programming [KLM+97]. In the following, we focus on proposals that can be more directly seen as class composition languages.

Mixin classes The notion of *parametric heir class* or *mixin class* has been originally introduced in [Moo86, Kee89].[1] As the first name suggests, a mixin is a uniform extension of many different parent classes with the same set of fields and methods, i.e., a class-to-class function. To be more concrete, let us consider the following class declaration.

```
class H1 extends P1 {
  decs
}
```

In languages that only support standard inheritance, if we want to extend another parent class, say `P2`, with *the same* set of fields and methods, then we have to write a new independent declaration, duplicating the code in `decs`.

```
class H2 extends P2 {
  decs
}
```

In languages supporting mixin classes, instead, we can give a name, say `M`, to `decs`, and then instantiate `M` on different parent classes, e.g., `P1` and `P2`, obtaining different heir classes equivalent to `H1` and `H2` above.

```
mixin M { decs }
class H1 = M extends P1
class H2 = M extends P2
```

Declarations in `decs` can refer to fields/methods, which should be provided by the parent (we will call these members *required* or *abstract* in the following).

Mixin classes for Java-like languages have been proposed, e.g., in [FKF98, ALZ00, ALZ03]. Other proposals for extensions of object-oriented languages with mixins are [BG96], which ex-

[1] The abbreviated term "mixin" is used sometimes; however, this term more appropriately refers to a language design where classes can be combined by a rich set of operators (see Jigsaw framework in the following), whereas mixin classes can typically only be combined by instantiation (function application).

tends Smalltalk, and [BPS99], which presents an imperative typed calculus for classes and mixins defined on top of a simple functional language with references and no polymorphic types.

Semantics of mixin classes has been given in the literature both in flattening and direct style. For instance, in [ALZ03] the language design has been guided by flattening semantics, there called the "copy principle", that is, that the behaviour of a class obtained by mixin instantiation should be the same that one would get by copying the code of the mixin into the heir. In [FKF98], instead, a direct semantics is defined which significantly deviates from that we would get by flattening (translation into plain Java), since, roughly, in case of unexpected conflicts both version of a method are kept.

Traits A trait is a collection of members, usually only methods. Using a syntax analogous to the one above, a trait could be written:

```
trait T { decs }
```

where methods used and not defined in `decs` should be provided later when composing the trait with others. As the reader has probably noted, a (basic) trait is formally equal to a mixin class with only methods; the difference is given by the richer set of operators that can be used to compose traits, notably: *Sum* merges two traits (with disjoint sets of defined methods) into one, by taking the union of required and defined methods; *Override* forms a new trait by layering, over an existing trait, additional methods that replace the original ones with the same name, if any; *Alias* forms a new trait by adding a new name for an existing offered method; *Exclusion* forms a new trait by removing an offered method.

Traits were originally introduced in [SDNB03] (see also [DNS$^+$06]), as a way to obtain fine grained reuse, avoiding problems which are typical of other techniques like multiple inheritance and mixin classes. Indeed, traits can be imported by classes, hence can be used to add composable features without changing the inheritance hierarchy. Moreover, a great simplification is given by the fact that they do not have state, so they leave to classes the responsibility to define fields.

Whereas the original work is focused on an untyped context, various proposals for using traits in connection with static typing can be found in literature [FR04, SD05, LS08b, LS08a], see also the survey in [NDS06]. The basic idea is that a trait type is, roughly, a pair of sets of method signatures, one for the required methods and one for the offered ones. In this way, it is possible to check whether a class can safely import a trait, that is, when all the requests are satisfied.

Semantics of traits has been generally given by flattening style, that is, by translation into a language without traits, with the notable exception of [LS08b, LS08a]. Given a flattening semantics, soundness of the type system can be proved by relying on soundness of the language without traits, that is, showing that flattening preserves well-typedness.

Finally, [BDG07, BDG08, BDL+10, BDS10] propose, for the first time, a language where traits are not an add-on to the object-oriented paradigm, but the unique mechanism for code reuse[2]. In other words, problems about inheritance are solved by simply removing inheritance. In this way, traits can fully demonstrate their advantages over conventional object-oriented development. This proposal goes into the direction of a class composition language where inheritance is replaced by a more powerful mechanism, as we fully exploit in this thesis.

One important advantage that our approach shares with traits is that the sum operator is symmetric and has a flexible explicit conflict resolution, whereas implicit precedence rules for method invocation become hard to maintain in the case of mixin chains (and even more complex for deep mixin composition, discussed later). This is effectively shown in [Sch05], where Schärli performs a refactoring of the Smalltalk library using traits, that offer only symmetric composition.

Jigsaw framework In the seminal work in Bracha's thesis [Bra92], the key idea is to consider classes as modularity units that can be combined by a rich set of operators (e.g., merge, override, restrict, hide, freeze, ...), leading to a programming paradigm based on *composition rather than inheritance* as the primary mechanism for structuring code. This paradigm is also called *mixin-based* programming, since it is possible to "mix" classes in different ways, as suggested by the puzzle metaphor. Standard inheritance can be subsumed by a certain combination of these operators see page 37. The resulting framework, called Jigsaw, is largely independent from the underlying language (*core* language) used for writing basic classes. Even more, the mixin notion is not strictly related to object-oriented programming but can be formulated in general in the context of module composition (*mixin module*), allowing a clean and unifying view of different linguistic mechanisms for composing modules.

Mixin-based programming has been extensively studied both on the methodological and foundational point of view [BC90, Bra92, BL92, BL96, AZ98, AZ02]. In particular, CMS (Calculus of Module Systems) [AZ02] is a kernel calculus (parametric in the core) providing a minimal set of primitive operators (sum, freeze and reduct) allowing to express all Jigsaw operators.

The choice of primitive composition operators in this thesis originates indeed from [AZ02]. The three CMS operators were taken as primitives in the FJIG version in [LSZ09b], where, as in Jigsaw, defined members can be *virtual*, *frozen* or *private*. In this thesis, as in [SZ10], fields and methods are all implicitly virtual, hence we do not include the *freeze* primitive operator which allows to express, e.g., hiding. Moreover, the *reduct* operator, handling maps from names into names, has been replaced by three operators which handle single names (restrict, alias and redirect), which provide the same expressive power and are more convenient for the meta-level.

[2]Their proposal has a wider scope: provide a linguistic mechanism for each type of reuse: traits for behaviour, records for state and interfaces for type reuse.

7.2 Meta-programming

There are many well-known strategies for code generation. The simplest one is to treat code as a string, which can be manipulated by the conventional language. This approach is very powerful, but gives no warranty on the quality of the produced code, including syntactic well-formedness. A well-known way to ensure the latter is to manipulate code, rather than as a string, as an abstract syntax tree (AST). This makes still possible for programmers to write arbitrary terms, but enforcing them to follow at least syntactic rules.

On the other hand, it is possible to restrict the user to a domain-specific language, rather than to the language itself, to manipulate code. Such approaches often ensure also to generate well-typed code, but such meta-languages are usually far from being Turing complete and have a limited control flow.

Meta-AspectJ Meta-AspectJ [ZHS04] is a Java extension providing facilities to write methods that manipulate (and emit as `String`) AspectJ code. Syntactic well-formedness of the produced code is guaranteed. As a matter of fact, Meta-AspectJ is a user friendly layer of syntactic sugar that hides the use of an ordinary data structure implementing an AST.

SafeGen and MorphJ SafeGen [HZS05] is a domain-specific language allowing to modify Java classes. More precisely, a SafeGen program takes a Java class as input and produces a new class as output. A theorem prover ensures that, for any well-typed class in input, a well-typed class is produced.

MorphJ [HZS07] is another domain-specific language which takes in input well-typed Java classes. Syntactically, it consists in a template-like mechanism added on top of Java. This mechanism is called *class morphing* and allows a class to abstract over the structure of other types. For instance, one can define a parametric class `Log<X>` which provides, for each method of `X`, a method with the same signature which invokes the original method and logs its result on a database.

MorphJ does not require generation of specialized code, but follows a direct semantics approach.

OpenJava OpenJava [TCKI00] offers the ability to define new language constructs, on top of Java, using metacircular compile-time execution. Programmers can define new constructs by writing *meta-classes*, that is, particular Java classes which instruct the OpenJava compiler on how to perform a type-driven translation. These meta-classes use the reflection-based *Meta Object Protocol (MOP)* to manipulate the source code and provide its translation. While there are similarities with our work, their approach is definitely lower level and we have a very different long-term goal: our aim is to bring compile-time execution in the realm of an already familiar

programming language, not to allow programmers to define their own extensions of an existing language.

Groovy Groovy [KGK+07] is an untyped language that allows both run time and compile-time (meta-circular) code generation. It uses a MOP as OpenJava does.

Meta-traits A meta-trait [RT07] is a trait that can have *place-holders* to be later filled with types, values, or method names. To generate an instance of a meta-trait, one gives actual values for the place-holders. Formally, a meta-trait is a function from some parameters to traits. This technique allows to emulate rename operators as well as generic classes/traits. This proposal supports separate typechecking of trait definitions. That is, there is no need to perform meta-trait expansion in order to typecheck classes which use a trait.

C++, D C++ [Int03] and D [Ale10] support template meta-programming, which is a very powerful, yet hard-to-debug technique. Template expansion is different from macro expansion, in fact a macro, which is also a compile-time language feature, generates code in-line using text manipulation and substitution. Macro systems often have limited compile-time process flow abilities and usually lack awareness of the semantics and type system of their companion language [3].

Template meta-programming is generally Turing-complete, meaning that any computation expressible by a conventional program can be computed, in some form, by a template meta-program. This implies that compilation may not terminate[4].

MetaML MetaML [TS00] is an extension of ML that allows a program to manipulate (MetaML) code fragments as data, and to execute such code during its own execution.

Multi-stage programming (MSP) provides a disciplined approach to run time code generation. MetaML shows how MSP can be used to reduce the overhead of abstractions, allowing clean, maintainable code without paying performance penalties.

The MetaML type system ensures that all the code that can be produced will be well-typed. In other words, a multi-stage program is typechecked once and for all before execution, ensuring the safety of all computations.

Meta-ML is tightly integrated, that is, programs are constructed, combined, compiled, and executed all under a single paradigm.

[3] With the notable exception of Lisp macros, which are written in Lisp itself, and allow more than simple text manipulation.

[4] This is true from a theoretical point of view, in practice termination is guaranteed by fixing a maximum level of recursion on template instantiations.

The MetaML approach is in many way specular to the approach of METAFJIG: They perform delayed compilation,that is, compilation during execution, whereas METAFJIG performs early execution, that is, execution during compilation.

JavaMint JavaMint [WRI+10] is strongly inspired by MetaML but proposes a new approach combining MSP with imperative features. JavaMint allows to dynamically generate expressions using the same techniques used in MetaML. However, since Java expressions can contain in-line anonymous inner classes, this approach can be used to dynamically generate classes. However, the generated classes have all the same structure, that is, only the implementation of the methods can be customized. They show staging having good effects on the performance of Java programs.

dynλ The paper [SSPJ98] details a technique called staged type inference that allows to emulate dynamic typing. Moreover, staged type inference also extends the expressive power of Meta-ML, allowing the generation of terms of (statically) unpredictable structural type.

Their approach is however different from our incremental typing: expressions of the form `run(`e`,`e'`)` require code generated by e to be of the same type of e', allowing the writer of e' to make precise requirement over the resulting code, or equivalently, leaving the burden of predicting the type to the writer of e' (who is, as usual, helped by the type inference mechanism).

Dynamic errors are modelled in dynλ with the expression `error`, that can have any type. However, even in the case of expressions of the form `run(`e`,error)` the call environment is used to infer the expected type for e.

In order to avoid requiring the programmer to explicitly denote the expected type, their work relies heavily on the underlying type inference algorithm, that is usually not available in a Java-like context. Moreover, the generalization to mutually recursive records (or classes) looks non trivial.

The philosophy of our approach is different: the (structural) type of a class produced by meta-programming is simply accepted into the program, and if the clients of such a class use it without respecting its type, they are ill-typed. In the dynλ approach, instead, the expected (structural) type of the produced element is extracted from the context, and if the produced type is not acceptable, the meta programming process is considered to be "wrong".

The language dynλ offers no property like our meta-level soundness, however, it is possible to check if some code is already in *fail* status, that is, wrong code can be detected. It is however unclear what kind of useful errors can be provided by the meta-programmer, since fail state is caused by failure in the unification / inference algorithm. In METAFJIG$_1$, instead, the composition exceptions are caused by a small set of conditions, depending only by the (public) interface of composed classes.

Template Haskell Template Haskell [SJ02] is an Haskell extension that supports algorithmic construction of programs at compile-time. The ability to generate code at compile time allows the programmer to implement even advanced features like the generation of supporting data structures and functions from existing data structures and functions. A staged type-checking algorithm interleaves type checking and compile-time execution. The language is expressive and simple, but still type-safe, because all run-time computations (either hand written or computed) are always type-checked before they are executed. However their approach is less safe w.r.t. METAFJIG$_1$ since there is nothing similar to meta level soundness. Template Haskell provides an algebraic data type for representing Haskell code as an abstract syntax tree, and *splice* operation invokes the compilation of such represented code. Trees can be explicitly built, moreover the language supports quotations[5] as a convenient syntactic sugar.

Template Haskell does allow compile-time IO.

Template Haskell has a very minimal management of dependency: you can not write a *splice* involving a function defined in the same file; you have to import that function to use it. If it happens, the programmer will simply get a "stage restriction" error.

A good survey, to better understand different metaprogramming approaches, is "DSL Implementation in MetaOCaml, Template Haskell, and C++" [COST04]

Comparison

The following table classifies meta-programming approaches w.r.t. several dimensions, that is, whether they:

A are homogeneous, that is, the language coincides with the meta-language,

B allow to execute generated code,

C statically ensure well-formedness w.r.t. the type system,

D perform compile-time execution,

E perform runtime compilation,

F ensure termination of code generation,

G code produced by libraries is always well typed,

H shape and type of generated code depend from external input.

	A	B	C	D	E	F	G	H
Meta-AspectJ	YES	NO	NO	NO	YES	NO	NO	YES
SafeGen	NO	NO	YES	NO	YES	YES	YES	NO
MJ	NO	YES	YES	NO	NO	YES	YES	NO
Open Java	YES	YES	NO	YES	NO	NO	NO	YES
Groovy	YES	YES	NO	YES	YES	NO	NO	YES
Meta-traits	NO	YES	YES	YES	NO	YES	YES	NO
C++,D	NO	YES	NO	YES	NO	NO	NO	NO
Meta-ML	YES	YES	YES	NO	YES	NO	YES	YES
Java-Mint	YES	YES	YES	NO	YES	NO	YES	NO
dynλ	YES	YES	NO	NO	YES	NO	NO	NO
Template Haskell	YES	YES	NO	YES	NO	NO	NO	NO
MetaFJig	YES	YES	NO	YES	NO	NO	YES	YES

Any of these features has advantages and disadvantages.

A Homogeneous approaches do not require programmers to understand two different languages, but a domain-specific language for code generation can be easier and safer. However, in such existing domains-specific languages there is usually no way to read external input.

B Allowing to execute generated code avoids many explicit compile-and-run iterations, but encourages involved programming.

C Statically ensuring well-formedness w.r.t. the type system increases reliability, but can reduce the expressive power. MetaFJig provides meta-level soundness: a good compromise between reliability and expressive power.

D Performing code generation at compile-time speeds up runtime, but slows down compilations. When the code generation is interleaved with the type-checking, it allows to use such type information to provide better error-checking for the non yet compiled part of the program, as C++ does.

E In the other way, performing code generation at run time allows to create program that continuously adapt itself w.r.t. the input.

F Enforcing the termination of the code generation process is good, but of no practical use if termination is not guaranteed to take place in a reasonable amount of time. Moreover, this requires to reduce the expressive power of the meta-language.

[5]It support also the interesting *quasi-quotation* feature, refer to [SJ02] for the details.

G Usually languages that does not statically ensure well-formedness w.r.t. the type system allows the generation of ill typed code. Instead, METAFJIG provides meta-level soundness: a good compromise between reliability and expressive power.

H Being able to produce code whose shape/type depends from external input allows to use meta-programming for a wide range of tasks, as shown in Section 4.2 and Section 6.2. However this prevent the possibility of statically ensuring well-formedness w.r.t. the type system.

7.3 Hierarchical composition

Many languages provide some mechanism to declare classes "inside" other classes, that is, nested classes. However, the semantic of a nested class is not obvious. Moreover, a (nested) class name can be used either for the "direct use", that is, as type annotation or instance generator, or for the "reuse", that is, inside class expressions. In the following, example (written with Java syntax and scoping rules)

```
class A{
  class B{ int mb(){return 1;} }
  class C extends B{}
  int ma1(){ return new B().mb();}
  int ma2(){ return new C().mb();}
}
class AA extends A{
  class B{ int m(){return 2;} }
}
```

the marked occurrence of class name `B` is an example of the second kind of use, while **new** `B()` and **new** `C()` are examples of the first kind of use. Both kinds of use can be either virtual or not. If the first kind of use is virtual then **new** `AA().ma1()` will evaluate to `2`, otherwise it will evaluate to `1`. On the other way, if the second kind of use is virtual then **new** `AA().ma2()` will evaluate to `2`, otherwise it will evaluate to `1`.

Language designers are free to choose whether the former or the latter kind of use is allowed to be virtual.

In detail:

- in order to see nested classes as virtual components, in the sense of the Jigsaw framework, both kinds of use have to be virtual.
- in Java nested classes have static binding, as fields and static methods. That is, nested classes are not virtual.

- in the first versions of beta [MMPN93] only the first kind of use, is virtual
- in literature of of family polymorphism (also known as virtual classes) [Ern01, EOC06, ISV05, IV07, ISV08] or deep mixin composition [OZ05, Hut06] usually also the second kind of use is virtual, and this is called "supporting virtual superclasses",
- our approach does not support virtual superclasses, since flattening removes all the information about the way a class was defined, that is, (in a class expression) references to other classes (formally, subexpressions which are paths) are all implicitly frozen. This allows to keep a simple type system, while keeping, as our examples show, the main advantages.

We stress that our choice is the one that more naturally fit flattening and substitutability principle, and allows a type system simple and compositional.

Indeed, supporting virtual superclasses would require the type of A to maintain information about `C extends B`. This exposes the inheritance hierarchy and, as Bracha pointed out [Bra92], breaks modularity.

This problem is already present in all the proposals supporting virtual superclasses (that usually offer only the extends operator) and would be even worse in our (richer) composition language.[6]

Analogously, providing virtual semantics for the first kind of use match well with flattening and substitutability principle; For example, in

```
A = { B = {...} B m(){ return new B(); } }
AA = copy A [+] { B = {...} }
```

class `AA` is a subclass of class `A`, and expression the `new B()` in `AA` clearly refers to the resulting class `AA.B`.

Our approach is sound since our sum and alias operators only add members, while restrict operator simple changes the kind, leaving the member type unaltered. We also have redirect operator, which removes a member completely; this operation is sound since references to such a member are redirected to another one.

Virtual classes and inheritance have a non trivial interaction. In proposal where subtyping and subclassing are not totally disjoint concepts, a naive approach is unsound. As example, in

```
class A{
  class B{int f1;}
  int k(B x){ return x.f1;}
}
class AA extends A{
  class B{int f2;}
```

[6] A compositional type system should likely use constraints and type variables as in [MW05].

```
  int k(B x){return x.f2+new B().f2; }
}

new AA().k(new AA.B())//ok
new A().k(new A.B())//ok
A a=new AA();
a.k(new A.B())//error: A.B.f2 does not exist
```

the the last method call is incorrect. Here `B` in `AA` *further extends* `B` in `A`, adding the field `f2`.

Intrigued by this problem, many authors [BOW98, IV07] recognize the need to break the coincidence of subtyping and subclassing in some controlled way. For example, in those works, to be more similar to the Java extends relation, subtyping and subclassing coincide whenever this does not directly brings to unsoundness.

We choose a more radical approach, that is, subclassing and subtyping are totally unrelated and the latter is declared by the programmer in a way that resembles implementation of interfaces in Java, that is, there are no a priori (subtype) relation between `A` and `AA`.

Many approaches offering "automatic" subtype relation impose that extending relation is declared only between nested classes of the same outer class (i.e. family), while here we have no such limitations.

Another criteria that can be used to classify proposals on nested classes is whether are members of instances [Ern01, OZ05, Hut06, CDNW07, BvdAB+10] or members of classes [ISV05, IV07, ISV08, NCM04, NQM06].

The former choice is more expressive, but requires a complex type-system usually involving dependent types. Our model follows the latter choice, mainly resembling nested classes of C++ and C#, and static nested classes of Java.

Jx and J& Nystrom et al. develop Jx [NCM04], a language providing a composition expressive power similar to our deep **`override`** operator, defined in page 81. They introduce an "hypothetical extension of Jx with abstract types" allowing an encoding for generics very similar to ours with the **`redirect`** operator. However, in our work **`redirect`** can be applied to any type, not only to ad-hoc "abstract types"[7]. J& [NQM06] is an extension of Jx with a composition mechanism similar to our sum operator.

Jx provide a non trivial formalization involving dependent types, while we do not need sophisticated types. Our types are just class paths, which have a very intuitive meaning for the programmer. In the following example (a simplified version of an example from [NCM04], Figure 1, rephrased in our syntax):

[7] Not to be confused with abstract classes.

```
A = {
  B = {...}
  int m(B b){...}
}
A2 = copy A [override] {
  B = { ... int y;}
  int m(B b){ ... b.y ... }
}
```

A2 is not a subtype of A, and*cannot* be declared to be a subtype of A. That is, it would be a type error to write

```
A2 = copy A [override] implements outer.A{
  B = { ... int y;}
  int m(B b){ ... b.y ... }
}
```

since the parameter types of method m are non compatible (they denote the classes A.B and A2.B, respectively). In other words, we have *no* notion of *family*.

However, code which works uniformly over families, if needed, for instance a method call x.m(y) which works with x and y of (static) type A, A.B and A2, A2.B, can be obtained as shown below:

```
C = {
  X = abstract{ abstract int m(Y y);}
  Y = abstract{}
  int k(){
    X x = new X();
    Y y = new Y();
    return x.m(y);
  }
}
CA = copy C [redirect .Y to A.B][redirect .X to A]
CA2 = copy C [redirect .Y to A2.B][redirect .X to A2]
```

which is equivalent to the following:

```
C = // as before
CA = {
  int k(){
    outer.A x = new outer.A();
    outer.A.B y = new outer.A.B();
    return x.m(y);
  }
}
```

```
CA2 = {
  int k(){
    outer.A2 x = new outer.A2();
    outer.A2.B y = new outer.A2.B();
    return x.m(y);
  }
}
```

^FJ ^FJ [ISV05, IV07], is a Java-like language where nested classes are members of classes.

As in our approach, ^FJ have no dependent types. Functionalities which works uniformly over families can be obtained using generics, analogously of what we do with **redirect**, see last example.

^FJ uses two kinds of types: absolute and relative, while we have chosen a simpler approach, and always use class paths, both in the syntax and in the formalization.

^FJ absolute types behaves like in Java, with conventional Java scoping rules. Our class paths are equivalents to ^FJ relative types: **outer**^{n}.$c_1 \ldots c_n\, c$ is roughly equivalent to $\hat{}^n$ `This@`c_1`@...@`c_n.c. For example **outer.outer**`.A.B.C` correspond to `^^This@A@B.C` and `<>` to `This`.

In [IV07], Figure 4, to better explain the meaning of ^FJ relative types, they provide an example of an `equals` method over an abstract syntax tree.

In order to show the similarity between relative types and class paths, we show the same example encoded in FJIG$_\star$.

```
Ast =class{
  Expr = abstract implements outer.Expr{...
    boolean equals(outer.Expr e){return false;}
    boolean eqLit(outer.Lit e){return false;}
    boolean eqPlus(outer.Plus e){return false;}
  }
  Lit =
    implements outer.Lit{...
      int i;
      boolean equals(outer.Expr e){
        return b.eqLit(this);
      }
      boolean eqLit(outer.Lit e){
        return this.i==e.i;
      }
      abstract boolean eqPlus(outer.Plus e);
    }
```

```
    [override] Expr
  Plus =
    implements outer.Plus{...
      outer.Expr op1;outer.Expr op2;
      boolean equals(outer.Expr b){
        return b.eqPlus(this);
      }
      abstract bolean eqLit(outer.Lit e);
      boolean eqPlus(outer.Plus e){
        return this.op1.equals(e.op1) && this.op2.equals(e.op2);
      }
    }
    [override] Expr
}
```

The fact that `Expr` implements **`outer.`**`Expr` ensures that any subclass of `Expr` will be also a subtype. Note that in FJIG$_\star$ we have to explicitly declare abstract methods `eqPlus` and `eqLit` to satisfy the **`implements`** clauses **`outer.`**`Lit` and **`outer.`**`Plus`. As future work we can develop a way to infer such (tedious) abstract declarations.

Deep and MixML Deep [Hut06] and MixML [DR08] are languages supporting deep mixin composition, that is, mutually recursive mixins with hierarchical namespace management. They offer an expressive power similar to our *sum* or *override* operators.

The semantic of Deep is given by flattening, while MixML translates into a core calculus.

Deep inspired us with the idea of hierarchical propagation of composition, built over the idea of deep mixin composition, as proposed in [OZ05].

Chapter 8

Conclusions

8.1 Current achievements

The contributions of this thesis are summarized and discussed below.

Design of the meta-level We have designed a new language schema for class-based languages, where classes are first-class values and new classes can be derived from existing ones by exploiting the full power of the language itself, used on top of a small set of primitive composition operators, instead of using a fixed mechanism like inheritance.

Our language is *meta-circular*, hence, differently from approaches such as, e.g., template meta-programming, programmers do not need to learn esoteric syntax and idioms. However, our design is also very different from other meta-circular approaches, where, typically, the meta-level is added on top of the computational language itself. For instance, in MetaML [TS00], the *bracket* operator returns, for any term e of the language, the corresponding meta-term $\langle e \rangle$, which intuitively represents a delayed computation, formally, is a value w.r.t. the reduction relation.

Here, instead, we consider class-based languages, where we can distinguish the computational language, that is, the language of *expressions* e which are executed at runtime (formally, reduced), from the composition language, that is, the language of *class expressions* ce which are compiled (formally, flattened), and are constructed by composition operators on top of basic classes. Adding the meta-level means that class expressions are no longer a separate syntactic category, but are merged into expressions, and composition operators are lifted at the level of expressions. A (silent) equivalent of the bracket operator is only used by the strong type system, since typechecking a basic class b (constant meta-expression) means typechecking b "as class expression".

This also implies a difference in how meta-terms are manipulated, that is, in how we can combine different pieces of code.

In MetaML there are no operators on meta-terms. The only possibility for constructing larger pieces of code from smaller ones is by using the *escape* annotation $\sim$, which allows to perform some reduction steps inside a meta-term. For instance, the term $(\lambda x.\langle 1 + \sim x\rangle)\langle 2\rangle$ reduces to $\langle 1 + 2\rangle$. This corresponds, in practice, to possibly include additional code "anywhere" (formally, in any subterm) in a meta-term, as analogously allowed by, apparently very different, approaches such as, e.g., the Meta Object Protocol. This flexibility opens the door for errors which can be difficult to understand.[1] In our approach we do not have any operator equivalent to escape, but, as mentioned above, *we do have* operators for composing code, that is, all the operators on class expressions ce lifted at the meta-level.

We believe this is a good choice, which makes the semantics of the meta-level easier to understand, since these operators for composing code have been designed on purpose to have a simple and intuitive meaning, basing on the experience of previous research summarized in Section 7.1. Notably, the main criticism which is usually made to meta-programming, that is, it is hard to debug code that is not written by a human, does not apply to our approach. Indeed, generated code is always obtained by a controlled manipulation (that is, only via the composition operators) of code originally written by the programmer.

Checked compile-time execution We have designed a lightweight technique to guarantee soundness: class composition errors are detected dynamically, and conventional typing errors are detected by *incremental typechecking*.

An alternative approach could be a purely static typechecking phase, as, e.g., in MetaML [TS00], taking place before compile-time execution. This would require a sophisticated type system with structural types for meta-expressions. This alternative direction is certainly interesting. However, the lightweight solution proposed in this thesis has the advantage of *simplicity* and *modularity*, since it allows to just reuse the type system of the conventional language, and the usual advantage of dynamic type systems that *more programs can be typed*, which is even more significant in a meta-programming context. For instance, no static type system could ever type a program where the shape of produced classes depends on external data-sources.

On the other hand, compared with approaches which check generated code only "a posteriori" [ZHS04, SJ02, TCKI00], our approach detects *all* composition errors dynamically, as soon as they appear , and conventional typechecking errors "on demand", as soon as the ill-typed class(es) are needed.[2] Hence, error detection is earlier and more informative error messages can be provided. Even more importantly, conventional reduction can be safely reused as it is, since

[1] Of course, in MetaML this is compensated by the purely static type system.

[2] An ill-typed unused class will silently reach the end of the compilation without raising any error. This class will, by definition, be of no influence in the process of compile-time execution.

no expression is executed before having been typechecked, hence dynamic errors like "message not understood" cannot be raised. Again, this is an advantage in terms of *modularity*, which allows the language to be implemented on top of (for example) the Java Virtual Machine. Moreover, since composition errors are modelled by exceptions, programmers can customize error messages and error handling, e.g., taking an alternative action when the composition of some classes fails.

In summary, the flexibility of our approach allows programmers to develop active libraries [CEG+00], that is, libraries performing additional abstraction steps, taking an active role and, using user input and external data-sources, adapting themselves to the specific user needs. At the same time, soundness and meta-level soundness (discussed below) are ensured.

Meta-level soundness We have formulated a new property which is significant for all meta-programming approaches: errors found by the conventional type-checker are always due to programmers' code, and not to the code of a library, differently from what happens, e.g., for C++ templates. This property holds in MetaFJig (Theorems 18 and 42) thanks to the fact that during compile-time execution we cannot generate arbitrary code, but only compose basic classes which were explicitly written in the library.

Design of FJig$_\star$ We have designed a class composition language which combines composition operators and nesting. In this way, a single class becomes an adequate unit of reuse, since it can embody a whole hierarchy of classes, which can be manipulated by the operators. We are not aware of any previous language or calculus supporting both Jigsaw-like operators and nesting. Thanks to this combination, we achieve more expressive power than other approaches [IV07, NQM06] which only provide , roughly, sum and overriding = restrict + sum operators, as the examples of generics, AOP and refactoring show. Moreover, we do not need sophisticated types, on the contrary our types (class paths) have a very intuitive meaning for the programmer.

Note that this last contribution is orthogonal to the others.

8.2 Future work

There are many possible directions for further research related to the work in this thesis.

First, we discuss some extensions of MetaFJig towards a real language. Indeed, it would not be sensible to extend in the way described in this thesis an existing language like Java, C++ or C#, since our approach is based on replacing inheritance by a more flexible composition mechanism, which also can encode generics. Hence, adding MetaFJig features on top of a language with inheritance and/or generics would lead to an over-complicated language.

Extending the expression language In this thesis we keep a Java-like flavour as much as possible, in order to make the presentation more understandable. However, it should be clear that our approach is largely independent from the particular language chosen for expressions and statements. Hence, a real language design could include useful features which are in principle orthogonal, such as, e.g., advanced typing features such as readonlyness [TE05], immutability [HP09], ownership [ZPL+10], exception safety [JP09, LS10] and safe parallelism [HMO10], or more conventional features like variable-length argument list and operator overloading.

Private members As mentioned in Section 1.1, in [LSZ09c, LSZ09b, LSZ10a] we have presented a language analogous to FJIG$_1$, but also including *frozen* and *private* members, and, correspondingly, operators such as *freeze* and *hide*, as in the original Jigsaw framework [Bra92]. However, the interaction of hiding with nesting is not trivial. Consider, for instance, the following program:

```
C = {
  A = { int ma(){return 1;}}
  B = { int mb(){return new outer.A().ma();}}
  }
D = copy C [hide ma in A]
```

We could expect this to reduce by flattening to:

```
C = //as before
D = {
  A = { private int ma(){return 1;}}
  B = { int mb(){return new outer.A().ma();}}
  }
```

But this would be unsound, since `D.B` would call the private method `D.A.ma`. A possible solution could be a "many level" private modifier, where a **`private`**`[`n`]` member is visible only in the first n enclosing classes. In particular, **`private`**`[0]` corresponds to the standard Java **`private`** modifier, denoting a member that is visible only in the current class (and its nested classes), **`private`**`[1]` denotes a member visible in the enclosing class (and its nested classes) and the **`public`** modifier is roughly equivalent to **`private`**`[`∞`]`. Following this idea, the example above would correctly reduce to

```
C = //as before
D = {
  A = { private[1] int ma(){return 1;}}
  B = { int mb(){return new outer.A().ma();}}
  }
```

Static members Adding static members seems easy at first look. However, there is a subtle interaction between nesting, static fields with mutable state and compile-time execution, leading to an error situation similar to scope extrusion, shown in the following example:

```
A = {
  static cpath foo=<>;
  static cpath id(cpath c){
    <>.foo=c;
    return c;
  }
}
B = {
  class m(){
    return {
        C = {...}
        D = copy outer.A.id(C)
      };
    }
E = copy B [+] copy A.foo
```

At the first step, we get for class `D` the same definition of `C`. However, `A.foo` keeps a reference to the class path `C`, having no meaning outside the unnamed class inside method `B.m`. The problem is made worse by the declaration of class `E`, requiring to recover the definition of `C`, that cannot be found anywhere. A drastic but sound solution is to reinitialize all the static fields after any application of (META-RED).

Annotations Annotations are a widely used mechanism to provide documentation and to inject flexible behaviour. Annotations in METAFJIG could be used as a descriptive language allowing to fire specialized behaviour in customized composition operators. For example the following class definition

```
C = new Instrument().apply(
  {
    @Getter @Setter int foo;
    @Pre(preM) @Post(postM) @PostExc(postM2)
    int m(int p){return e;}
    boolean preM(int p){...}
    boolean postM(int p, int result){...}
    boolean postM2(int p, Throwable e){...}
    ...
  })
```

with an opportune definition for `Instrument.apply` would be equivalent to

```
C = {
    int foo;
    int getFoo(){return this.foo;}
    void setFoo(int foo){this.foo=foo;}
    int m(int p){
      if(!this.preM(p))throw ...
      try{
        int result =e;
        if(!postM(p,result))throw ...
      }catch(Throwable t){
        if(!postM2(p,t))throw ...
        throw t;
      }
   }
    boolean preM(int p){...}
    boolean postM(int p, int result){...}
    boolean postM2(int p, Throwable e){...}
    ...
  }
```

The semantics of composition operators should be extended to handle annotations, in a non obvious way. For instance, when restricting an annotated method, should the annotation remain in place or be deleted together with the method implementation?

We conclude this section with two research directions which are related to the checked compile-time compilation process in itself, rather than to the language.

Parallel compilation Clearly, compile time execution can be time consuming. However, thanks to the dependency relation, it is possible to determine which class definitions can be reduced in parallel. To avoid unsound modification of static fields, any thread should have its own set of independent static fields.

Earlier error detection In the current model, typechecking during checked compile-time execution always requires code to be *closed*, that is, type information about all class names occurring in code must be available. Hence, typing errors can be detected only at this time.

A constraint-based type system, as in [ADDZ05], where classes can be typechecked separately, would allow earlier error detection. With this approach, we could give a stronger definition for the judgement $\vdash^o$, requiring the absence of intrinsic errors.

Bibliography

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In Boris Magnusson, editor, *ECOOP'02 - Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 334–367. Springer, 2002.

[ADDZ05] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2005*. ACM Press, January 2005.

[Ale10] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 1st edition, 2010.

[ALZ00] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of Java with mixins. In E. Bertino, editor, *ECOOP'00 - European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 154–178. Springer, 2000. An extended version is [ALZ03].

[ALZ03] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam–designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, September 2003. Extended version of [ALZ00].

[ALZ06] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Flexible type-safe linking of components for Java-like languages. In *JMLC'06 - Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2006.

[AZ98] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, August 1998.

[AZ01] Davide Ancona and Elena Zucca. True modules for Java-like languages. In J.L. Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Pro-*

gramming, number 2072 in Lecture Notes in Computer Science, pages 354–380. Springer, 2001.

[AZ02] Davide Ancona and Elena Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1990*, volume 25(10) of *SIGPLAN Notices*, pages 303–311. ACM Press, October 1990.

[BC97] John Boyland and Giuseppe Castagna. Parasitic methods: An implementation of multi-methods for Java. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1997*, pages 66–76. ACM Press, 1997.

[BDG07] Viviana Bono, Ferruccio Damiani, and Elena Giachino. Separating type, behavior, and state to achieve very fine-grained reuse. In *9th Intl. Workshop on Formal Techniques for Java-like Programs*, 2007.

[BDG08] Viviana Bono, Ferruccio Damiani, and Elena Giachino. On traits and types in a Java-like setting. In *TCS'08 - IFIP Int. Conf. on Theoretical Computer Science*. Springer, 2008.

[BDL$^+$10] Lorenzo Bettini, Ferruccio Damiani, Marco De Luca, Kathrin Geilmann, and Jan Schäfer. A calculus for boxes and traits in a java-like setting. In Dave Clarke and Gul A. Agha, editors, *COORDINATION'10 (Coordination Models and Languages)*, volume 6116 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2010.

[BDNW08] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Comput. Lang. Syst. Struct.*, 34(2-3):83–108, 2008.

[BDS10] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing software product lines using traits. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC'10 - ACM Symposium on Applied Computing*, pages 2096–2102. ACM, 2010.

[BF04] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *ECOOP'04 - Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 389–413, 2004.

[BG96] Gilad Bracha and David Griswold. Extending Smalltalk with mixins. In *OOPSLA96 Workshop on Extending the Smalltalk Language*, April 1996.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, April 1992.

[BL96] Guruduth Banavar and Gary Lindstrom. An application framework for module composition tools. In *ECOOP'96 - European Conference on Object-Oriented Programming*, number 1098 in Lecture Notes in Computer Science, pages 91–113. Springer, July 1996.

[BOSW98] Gilad Bracha, M. Odersky, D. Stoutmire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, pages 183–200. ACM Press, October 1998.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98 - European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 523–549, 1998.

[BPS99] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In R. Guerraoui, editor, *ECOOP'99 - European Conference on Object-Oriented Programming*, number 1628 in Lecture Notes in Computer Science, pages 43–66. Springer, 1999.

[Bra92] Gilad Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.

[BvdAB+10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In Theo D'Hondt, editor, *ECOOP'10 - Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2010.

[CDNW07] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In Brian M. Barry and Oege de Moor, editors, *AOSD'07 - Aspect-oriented software development*, volume 208, pages 121–134. ACM, 2007.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CEG+00] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, number 1766 in Lecture Notes in Computer Science, pages 25–39. Springer, 2000.

[CMLC06] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, 2006.

[COST04] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*. Springer, 2004.

[CSZ10] Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig - Modular composition of nested classes. In *FOOL 2010 - Intl. Workshop on Foundations of Object-Oriented Languages*, 2010.

[CSZ11] Andrea Corradi, Marco Servetto, and Elena Zucca. DeepFJig - Modular composition of nested classes. In *PPPJ'11 - Principles and Practice of Programming in Java*, 2011. To appear.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

[DR08] Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In James Hook and Peter Thiemann, editors, *Intl. Conf. on Functional Programming 2008*, pages 307–320. ACM Press, 2008.

[EOC06] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *ACM Symp. on Principles of Programming Languages 2006*, volume 41, pages 270–282. ACM Press, January 2006.

[Ern01] Erik Ernst. Family polymorphism. In J.L. Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 303–326. Springer, 2001.

[FF98] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with Units and Mixins. In *Intl. Conf. on Functional Programming 1998*, pages 94–104, 1998.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183. ACM Press, 1998.

[FR04] Kathleen Fisher and John Reppy. A typed calculus of traits. In *FOOL'04 - Intl. Workshop on Foundations of Object-Oriented Languages*, January 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[GJSB05] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. The Java series. Addison-Wesley, third edition, 2005.

[GS09] Joseph Gil and Tali Shragai. Are we ready for a safer construction environment? In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 495–519. Springer Berlin / Heidelberg, 2009.

[HMO10] Philipp Haller and Martin Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP'10 - Object-Oriented Programming*, pages 354–378. Springer, 2010.

[HP09] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *ECOOP'09 - Object-Oriented Programming*, Lecture Notes in Computer Science. Springer, 2009.

[Hut06] DeLesley Hutchins. Eliminating distinctions of class: using prototypes to model virtual classes. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2006)*, pages 1–20. ACM Press, 2006.

[HZS05] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with SafeGen. In Robert Glück and Michael R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 2005.

[HZS07] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *ECOOP'07 - Object-Oriented Programming*, pages 399–424. Springer, 2007.

[Int03] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages – C++*. International Organization for Standardization, 2003.

[IPW99] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146. ACM Press, 1999.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[ISV05] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In Kwangkeun Yi, editor, *APLAS 2005 - Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2005.

[ISV08] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. *Journ. of Functional Programming*, 18(3):285–331, 2008.

[IV07] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007)*, pages 113–132. ACM Press, 2007.

[JP09] Bart Jacobs and Frank Piessens. Failboxes: Provably safe exception handling. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 470–494. Springer, 2009.

[Kee89] S.C. Keene. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, 1989.

[KGK+07] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353. Springer, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[LS08a] Luigi Liquori and Arnaud Spiwack. Extending FeatherTrait Java with interfaces. *Theoretical Computer Science*, 398(1-3):243–260, 2008.

[LS08b] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 30(2), 2008.

[LS10] Giovanni Lagorio and Marco Servetto. Strong exception-safety for Java-like languages. In *12th Intl. Workshop on Formal Techniques for Java-like Programs*, ACM International Conference Proceedings Series, 2010.

[LSZ09a] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Customizable composition operators for Java-like classes (extended abstract). In *ICTCS'09 - Italian Conf. on Theoretical Computer Science*, 2009.

[LSZ09b] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw - a minimal core calculus for modular composition of classes. In Sophia Drossopoulou, editor, *ECOOP'09 - Object-Oriented Programming*, number 5653 in Lecture Notes in Computer Science. Springer, 2009.

[LSZ09c] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *FOOL'09 - Intl. Workshop on Foundations of Object-Oriented Languages*, 2009.

[LSZ10a] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw - replacing inheritance by composition in Java-like languages. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, May 2010. Submitted for journal publication.

[LSZ10b] Giovanni Lagorio, Marco Servetto, and Elena Zucca. A lightweight approach to customizable composition operators for Java-like classes. *Electronic Notes in Theoretical Computer Science*, 263:161–177, 2010. FACS'09 - International Workshop on Formal Aspects of Component Software.

[MFH01] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New age components for old fashioned Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*. ACM Press, 2001. SIGPLAN Notices.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[Moo86] David A. Moon. Object-oriented programming with flavors. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1986*, pages 1–8. ACM Press, 1986.

[MW05] Henning Makholm and J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In Olivier Danvy and Benjamin C. Pierce, editors, *Intl. Conf. on Functional Programming 2005*, pages 156–167. ACM Press, 2005.

[NCM04] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. *SIGPLAN Not.*, 39(10):99–115, 2004.

[NDS06] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *Journ. of Object Technology*, 5:66–90, 2006.

[NQM06] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. *SIGPLAN Not.*, 41(10):21–36, 2006.

[OZ05] M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *FOOL'05 - Intl. Workshop on Foundations of Object-Oriented Languages*, 2005.

[Par78] David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[RT07] John Reppy and Aaron Turon. Metaprogramming with traits. In Erik Ernst, editor, *ECOOP'07 - Object-Oriented Programming*, number 4609 in Lecture Notes in Computer Science, pages 373–398. Springer, 2007.

[Sch05] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Bern, February 2005.

[SD05] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-Like Languages. In *ECOOP'05 - Object-Oriented Programming*, volume 4406, pages 453–478. Springer, 2005.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP'03 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003.

[She00] Tim Sheard. Accomplishments and research challenges in meta-programming. In *2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer, 2000.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37:60–75, 2002.

[SSPJ98] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Symp. on Principles of Programming Languages 1998, pages 289–302. ACM Press, 1998.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Reading. Addison-Wesley, special edition, 2000.

[SZ10] Marco Servetto and Elena Zucca. MetaFJig - A meta-circular composition language for Java-like classes. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2010)*, pages 464–483. ACM Press, 2010.

[TCKI00] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Kilijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science, pages 117–133. Springer, 2000.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. *ACM SIGPLAN Notices*, 40(10):211–230, October 2005.

[Tor04] Mads Torgersen. The expression problem revisited. In Martin Odersky, editor, *ECOOP'04 - Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 123–143. Springer, 2004.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[WRI+10] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *PLDI'10 - ACM Conf. on Programming Language Design and Implementation*. ACM Press, 2010.

[WV00] J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *ESOP 2000 - European Symposium on Programming 2000*, number 1782 in Lecture Notes in Computer Science, pages 412–428. Springer, 2000.

[ZHS04] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 583–605. Springer, 2004.

[ZPL+10] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2010)*, pages 598–617. ACM Press, 2010.