A Type-theoretic Reconstruction of the Visitor Pattern

Peter Buchlovsky[1]

Computer Laboratory
University of Cambridge
Cambridge CB3 0FD, United Kingdom

Hayo Thielecke[2]

School of Computer Science
University of Birmingham
Birmingham B15 2TT, United Kingdom

Abstract

In object-oriented languages, the Visitor pattern can be used to traverse tree-like data structures: a visitor object contains some operations, and the data structure objects allow themselves to be traversed by accepting visitors. In the polymorphic lambda calculus (System F), tree-like data structures can be encoded as polymorphic higher-order functions. In this paper, we reconstruct the Visitor pattern from the polymorphic encoding by way of generics in Java. We sketch how the quantified types in the polymorphic encoding can guide reasoning about visitors in general.

Key words: Visitor pattern, polymorphic types, object-oriented programming, Generic Java

1 Introduction

Tree-like data structures, such as abstract syntax trees or binary trees, and their traversal (often called tree walking) are ubiquitous in programming. Modern functional languages, such as ML and Haskell, provide constructs in the form of datatype definitions and pattern matching to deal with trees and more general recursive datatypes. However, in object-oriented languages such as Java, the situation is more complicated. Testing class membership and branching on it is widely considered a violation of object-oriented style.

[1] Email: `Peter.Buchlovsky@cl.cam.ac.uk`
[2] Email: `H.Thielecke@cs.bham.ac.uk`

Instead, the Visitor pattern [6] has been proposed to define operations on tree-like inductive datatypes.

The simplest example using visitors is that of a sum type of the form $A+B$. Since Java lacks a sum type, the Visitor pattern implements a class doing the job of a sum type by a sort of double-negation transform. Concretely, the class has a method for accepting a visitor; the visitor itself has two methods, one accepting arguments of type A, the other of type B. If we take a very idealized view by taking methods as functions and objects as tuples of such functions, we can regard the above as an instance of a well-known isomorphism in the polymorphic λ-calculus [8]:

$$A + B \cong \forall\alpha.((A \to \alpha) \times (B \to \alpha)) \to \alpha$$

These isomorphisms are firmly grounded in programming language theory. Via the Curry-Howard correspondence, they also form part of a bigger picture in terms of the definability of logical connectives such as disjunction in higher-order logics.

The present paper aims to flesh out this type-theoretic view of visitors. Specifically, a Design Pattern (such as the aforementioned Visitor pattern), by its very nature, is not so much a single unambiguous definition, but a variety of related instances. Hence we aim to clarify how variants of the Visitor pattern relate to the more idealized type-theoretic picture. We classify visitors as internal or external, depending on whether the visitor itself or the tree specifies the traversal. Moreover these can be either functional (by returning a value) or imperative (having a `void` return type and side-effects instead).

The resemblance of variants of the Visitor pattern to the encoding of algebraic types into System F may be part of the type-theoretic folklore. However, formulating this precisely seems to be very useful, given the possibility of some technology transfer from the highly developed theory of polymorphic lambda calculi to less rigorous, but widely used design patterns. Our point is not to translate object-oriented languages into functional ones or conversely. Rather, we can see that the same notion has a manifestation in both of these very different scenarios.

The contributions of this paper include the following:

- We present a stylized polymorphic λ-calculus to make the relation of polymorphic encodings to visitors perspicuous.
- We show the type soundness of the resulting visitors in Featherweight Generic Java [9].
- We sketch how this abstract view of visitors could be useful for reasoning about visitors.

For completeness, an appendix (Section A) gives more details on Featherweight Generic Java. These details are not essential for understanding the paper.

2 Background

We briefly recall the two areas of relevant background between which we aim to bridge: visitors as presented in the Design Patterns literature [6], and polymorphic lambda calculi [7].

We find it useful to give some object-oriented terminology:

Interface – A fully abstract class. Defines a set of methods but does not give their bodies. This corresponds to an ML signature.

Class – A class may *implement* an interface by providing a body for each method header in the interface. This corresponds to an ML structure.

We will consider Generic Java [3] (Java extended with type parameters on classes, interfaces and methods) throughout. Briefly, the syntax is as follows. An interface or class definition of the form `class C<`α`>{...}` defines a class `C` parameterized by the type variable α. The type `C<Int>` instantiates the type parameter of class `C` to `Int`. A method definition of the form `<`α`> T m(...){...}` defines a method `m` in which the type variable α is universally quantified. A call to method `m` of the form `o.m<Int>(...)` instantiates α to `Int`.

The purpose of the Visitor pattern is to organize a program around the operations on a datatype as opposed to the constructors. The canonical example of the Visitor pattern consists of abstract syntax trees and their traversal by various phases of a compiler; this approach is used in SableCC [5]. We will consider a simpler example based on binary trees of integer leaves and the operation of summing up the leaves.

The standard object-oriented implementation of binary trees is based around the Composite pattern [6]. The datatype signature (sometimes called "element") is represented as an interface and the constructor for each variant of the datatype becomes a class (sometimes called "concrete element") which implements the interface. The interface includes method headers that specify the signatures of all operations on the data. This forces each constructor class to provide a method to handle the appropriate case of the operation.

This approach permits new datatype variants to be added without modifying existing code, something that is not possible in ML. The disadvantage is that adding new operations is difficult as every existing class has to be amended. The Visitor pattern turns the situation around. It becomes easy to add new operations but difficult to add new variants.

The Visitor pattern is used as follows. Every operation on the datatype is packaged into a *concrete visitor* class. The case for handling each variant is contained in a method typically named `visitCons` where `Cons` is the name of the constructor class for that variant. Every concrete visitor implements a visitor interface. This specifies the types of visit methods that must be present in a concrete visitor. It can be seen as a signature for concrete visitors.

The constructor classes of the datatype are modified to include an *accept* method. Its role is to accept a visitor and call the visit method for the variant which the class implements. This is essentially a form of double dispatch on the datatype variant and the visit method. Any components of the variant stored in fields inside the class must be passed to the visit method. It is also necessary to parameterize the accept methods and visitor interface since we cannot know in advance the type yielded by any concrete visitor class.

To summarize, the visitor pattern consists of the following classes:

Visitor – This is an interface for visitors. It declares visit methods named visitCons for each Cons class.

ConcreteVisitor – One class for each operation on the data. It implements the Visitor interface and has to provide implementations for each of the visit methods declared there.

Data/Element – This is an interface naming a datatype (e.g. `BinTree`). It declares an accept method which takes a Visitor object as an argument.

ConcreteData/ConcreteElement – A Cons class for each variant of the datatype (e.g. `Leaf`). This corresponds to an ML datatype constructor named Cons. It implements the accept method that calls visitCons in the Visitor object.

The Visitor pattern does not prescribe where a visitor should store intermediate results or how it should return the result to the caller. We will distinguish between functional visitors which return intermediate results through the result of the call to accept and imperative visitors which accumulate results in some field inside the visitor.

Another aspect of the Visitor pattern is the choice of traversal strategies for composite objects. We could put the traversal code in the datatype. To do this we ensure that the accept method is called recursively on any component objects and passes the results to the visitor in the call to visit. Alternatively, we could put the traversal code in the visitor itself. We will refer to these as "internal" and "external" visitors respectively, by analogy with internal and external iterators [6].

We will use the polymorphic lambda calculus (System F) to encode data types. In fact, we need a more powerful extension of System F with polymorphic type constructors, called F_ω, since it allows us to approximate generics and interfaces better than we could with System F alone. Most features of this system will not be new to anyone familiar with advanced languages like Haskell or ML: intuitively, F_ω contains polymorphic functions and also polymorphic type constructors.

See Figure 1 for a fairly standard presentation of F_ω extended with finite products. We write $\alpha \notin \Gamma$ to mean that α is not among the free variables in Γ. We abbreviate $\forall\alpha{::}{*}.T$ as $\forall\alpha.T$ and similarly for $\Lambda\alpha{::}{*}.T$. Empty products are written as $\mathbf{1}$. We will also write t_i for the projection $\pi_i\, t$ and $\lambda\langle a, b\rangle{:}A \times B.t$ for $\lambda x{:}A \times B.[a \mapsto \pi_1\, x,\, b \mapsto \pi_2\, x]\, t$.

Terms

$$t, s ::= x \mid \lambda x{:}T.t \mid t\,t \mid \Lambda\alpha{::}K.t \mid t\,[T] \mid \langle t_i\rangle_{i\in 1..n} \mid \pi_i\,t$$

Kinds

$$K ::= * \mid K \Rightarrow K$$

Types

$$T ::= \alpha \mid T \to T \mid \forall\alpha{::}K.T \mid \Lambda\alpha{::}K.T \mid T\,[T] \mid \textstyle\prod_{i\in 1..n} T_i$$

Contexts

$$\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, \alpha :: K$$

Typing

$$(\text{Var})\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$(\text{Abs})\ \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2} \qquad (\text{App})\ \frac{\Gamma \vdash t : T_1 \to T_2 \quad \Gamma \vdash s : T_1}{\Gamma \vdash t\,s : T_2}$$

$$(\text{TAbs})\ \frac{\Gamma, \alpha :: K \vdash t : T}{\Gamma \vdash \Lambda\alpha{::}K.t : \forall\alpha{::}K.T}\ \ \alpha \notin \Gamma$$

$$(\text{TApp})\ \frac{\Gamma \vdash t : \forall\alpha{::}K.T_1 \quad \Gamma \vdash T_2 :: K}{\Gamma \vdash t\,[T_2] : [\alpha \mapsto T_2]\,T_1}$$

$$(\text{Tuple})\ \frac{\forall i \in 1..n \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \langle t_i\rangle_{i\in 1..n} : \prod_{i\in 1..n} T_i} \qquad (\text{Proj})\ \frac{\Gamma \vdash t : \prod_{i\in 1..n} T_i}{\Gamma \vdash \pi_j\,t : T_j}$$

Kinding

$$(\text{TVar})\ \frac{\alpha :: K \in \Gamma}{\Gamma \vdash \alpha :: K}$$

$$(\text{KAbs})\ \frac{\Gamma, \alpha :: K_1 \vdash T :: K_2}{\Gamma \vdash \Lambda\alpha{::}K_1.T :: K_1 \Rightarrow K_2}\ \ \alpha \notin \Gamma$$

$$(\text{KApp})\ \frac{\Gamma \vdash T_1 :: K_1 \Rightarrow K_2 \quad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1\,[T_2] :: K_2}$$

$$(\text{KArrow})\ \frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \qquad (\text{KAll})\ \frac{\Gamma, \alpha :: K \vdash T :: *}{\Gamma \vdash \forall\alpha{::}K.T :: *}\ \ \alpha \notin \Gamma$$

$$(\text{KTuple})\ \frac{\forall i \in 1..n \quad \Gamma \vdash T_i :: *}{\Gamma \vdash \prod_{i\in 1..n} T_i :: *}$$

Reductions

$$\begin{aligned}
(\lambda x{:}T.t)\,s &\ \rightsquigarrow_\beta\ [x \mapsto s]\,t \\
(\Lambda\alpha{::}K.t)[T] &\ \rightsquigarrow_\beta\ [\alpha \mapsto T]\,t \\
\pi_j\,\langle t_i\rangle_{i\in 1..n} &\ \rightsquigarrow_\beta\ t_j \\
(\Lambda\alpha{::}K.T_1)[T_2] &\ \rightsquigarrow_\beta\ [\alpha \mapsto T_2]\,T_1
\end{aligned}$$

Fig. 1. System F_ω with products.

3 Visitors and algebraic type encodings

In this section, we recall the encoding of algebraic types in System F [14] (sometimes called the Böhm-Berarducci encoding [2]). The standard description is in terms of F-algebra. However, we find it useful to present it in a slightly different style: while isomorphic to the usual account, it makes the connection to the Visitor pattern more explicit.

3.1 Internal visitors

We consider algebraic types of the form

$$\mu\alpha.\sum_{i\in 1..n} F_i[\alpha]$$

and we further restrict attention to F_i that are products of the recursive type variable α and type constants. Thus the types defined this way are various forms of trees, and we will see how visitors are tree walkers.

Definition 3.1 We define internal visitors to be pairs of the form $\langle A, \langle a_i\rangle_{i\in 1..n}\rangle$, where A is a type (called the result type), and $\langle a_i\rangle_{i\in 1..n}$ is a tuple of functions $a_i : F_i[A] \to A$ (called the visit methods).

In the presence of sums, we could define $F[X] = \sum_{i\in 1..n} F_i[X]$. Visitors are essentially F-algebras, and their visit methods give the structure map.

Next, we consider the encoding of algebraic types (that is to say initial F-algebras), but rephrased in terms of weakly initial visitors.

We define an object T as follows:

$$T = \forall\alpha.(\prod_{i\in 1..n}(F_i[\alpha] \to \alpha)) \to \alpha$$

T comes equipped with visit methods:

$$\begin{aligned} \mathsf{cons}_i &: F_i[T] \to T \\ \mathsf{cons}_i &= \lambda x{:}F_i[T].\Lambda\alpha.\lambda v{:}\prod_{j\in 1..n}(F_j[\alpha] \to \alpha).v_i(F_i[\lambda t{:}T.t[\alpha]v]x) \end{aligned}$$

Intuitively, we think of the cons_i as the constructors of the datatype T.

Then T with $\langle \mathsf{cons}_i\rangle_{i\in 1..n}$ is weakly initial in the following sense. Let A together with $\langle a_i : F_i[A] \to A\rangle_{i\in 1..n}$ be any visitor. Then there is a function from T to A defined by:

$$\lambda t{:}T.t[A]\,\langle a_i\rangle_{i\in 1..n}$$

This explains the definition of T: any element t of T has the ability to accept any visitor with result A and visit methods visit$_i$, and to yield an element of A.

$$T = \forall\alpha.\underbrace{(\prod_{i\in 1..n}(\underbrace{F_i[\alpha] \to \alpha}_{\mathrm{visit}_i[\alpha]}))}_{\mathrm{Visitor}[\alpha]} \to \alpha$$

More intuitively, the elements of T can be thought of as trees; and they use the visit methods to collapse themselves recursively into a single element of A. This is achieved by letting the visitor visit any subtrees, thereby collapsing them into elements of the result type, and then calling the appropriate visit methods for the topmost node.

$$\mathsf{cons}_i = \underbrace{\lambda x{:}F_i[T].}_{\text{constructor arguments}} \overbrace{\Lambda\alpha.\lambda v{:}\prod_{j\in 1..n}(F_j[\alpha]\to\alpha).}^{\text{accept visitor}} \underbrace{v_i}_{\text{call visit method}} \overbrace{(F_i[\lambda t{:}T.t[\alpha]v]x)}^{\text{walk over subtrees}}$$

Visitor morphisms that witness the initiality of T can be seen as calling accept on a T object and passing it a concrete visitor:

$$\widehat{a} = \lambda t{:}T.\ \underbrace{t[A]}_{\text{call accept}}\ \overbrace{\langle a_i\rangle_{i\in 1..n}}^{\text{concrete visitor}}$$

3.2 External visitors

An external visitor consists of a pair $\langle A, \langle v_i\rangle_{i\in 1..n}\rangle$, where A is a type and $\langle v_i\rangle_{i\in 1..n}$ is a tuple of functions $v_i : F_i[T] \to A$. Note the T in the position where an internal visitor would have another occurrence of A. Intuitively, an external visitor has visit methods just like an internal one; the difference is that these methods may accept trees of type T as arguments, rather than automatically collapsing the trees into elements of result type A.

We define a structure S that accepts external visitors in the same way that T accepts internal ones:

$$S = \forall\beta.(\prod_{i\in 1..n}(F_i[T]\to\beta))\to\beta$$

This object has the structure of visit methods:

$$\mathsf{p}_i \ :\ F_i[S] \to S$$

$$\mathsf{p}_i = \lambda x{:}F_i[S].\Lambda\beta.\lambda m{:}\prod_{j\in 1..n}(F_j[T]\to\beta).m_i(F_i[\lambda s{:}S.s[T]\,\langle\mathsf{cons}_i\rangle_{i\in 1..n}]\,x)$$

The visitor structure on S induces a function from the weakly initial visitor T.

$$\widehat{\mathsf{p}} : T \to S$$

Intuitively, this map takes a tree and pattern matches it, so that it can be visited by an external visitor.

External visitors can themselves use this map for further pattern matching of subtrees and thus traversal. However, without adding recursion, an external visitor can not traverse trees of arbitrary depth. Since the traversal of the whole tree is no longer built-in the way it was in internal visitors, an external visitor would have to recur under its own steam, as it were, which requires a fixpoint combinator in the visitor.

4 Factorizing the encoding

The overall view on visitors in this section is as follows. Our aim is to bring out the connection between the Visitor pattern and type encodings in System F by factoring various translations. We present an encoding $\mathcal{O}[\![\cdot]\!]$ of algebraic types into a polymorphic λ-calculus that is stylistically close to object-oriented languages. The calculus is a restricted subset of F_ω (with products), and the encoding corresponds to the classical encoding $\mathcal{F}[\![\cdot]\!]$ of algebraic types in System F (up to some reductions and type isomorphisms). On the other hand, because our calculus resembles object-oriented languages, there is a straightforward embedding $[\![\cdot]\!]$ into Featherweight Generic Java (which is itself a subset of Generic Java); and this lets us recover the internal variant of the Visitor pattern (by composition).

$$\begin{tikzcd}
 & & \text{Alg. Types} \arrow[r, "\mathcal{F}[\![\cdot]\!]"] \arrow[d, "\mathcal{O}[\![\cdot]\!]"] \arrow[dl, "\text{Visitor}"'] & \text{F} \arrow[d, hook] \\
\text{GJ} & \text{FGJ} \arrow[l, hook'] & \lambda^{oo}_\omega \arrow[l, "[\![\cdot]\!]"'] \arrow[r, hook] & F_\omega
\end{tikzcd}$$

We need a calculus for expressing visitors in such a way that we can then transform them into both System F and FGJ. To do so, we will approximate "objects" as tuples of methods (of a restricted function type). Another ingredient is a form of parameterized let on *types* that we will use for approximating interfaces with generics. We define:

$$\mathsf{let}\ \alpha\texttt{<}\beta\texttt{>}\ \mathsf{be}\ T_1\ \mathsf{in}\ T_2 \equiv (\Lambda\alpha{::}{*}\Rightarrow{*}.T_2)\,[\Lambda\beta.T_1]$$

Where there are no type parameters we omit <> and define:

$$\mathsf{let}\ \alpha\ \mathsf{be}\ T_1\ \mathsf{in}\ T_2 \equiv (\Lambda\alpha{::}{*}\Rightarrow{*}.T_2)\,[T_1]$$

This construct could be extended to include an arbitrary number of parameters β by adding a tuple kind to the definition of F_ω.

We define some conventions: we write '+' to mean "one or more occurrences"; $\vec{\beta}$ for $\beta_1, \ldots, \beta_p$; $[\vec{S}]$ for $[S_1] \ldots [S_p]$; and $[\![\vec{S}]\!]$ for $[\![S_1]\!]\texttt{,} \ldots \texttt{,}[\![S_p]\!]$. Our calculus for defining visitor types is then as follows.

Definition 4.1 Types of λ^{oo}_ω are *well-kinded* types of F_ω (with let) given by J in

$$\begin{array}{rcll}
J & ::= & (\mathsf{let}\ \alpha\texttt{<}\beta\texttt{>}\ \mathsf{be}\ O\ \mathsf{in})^+\ \alpha[\vec{S}] & \textit{interface definitions} \\
O & ::= & \prod_{i\in 1..n} M_i & \textit{object type} \\
M & ::= & \forall\vec{\beta}.(\prod_{j\in 1..m} S_j) \rightarrow S & \textit{method type} \\
S & ::= & \alpha[\vec{S}] & \textit{interface instantiation} \\
 & | & \mathsf{int} & \textit{integer type}
\end{array}$$

where $\alpha, \beta \in \mathrm{TyVar}$ and int is a constant type. Note that this grammar restricts all type variables to the kinds $*$ or $*\Rightarrow*$. We also insist that all type variables are bound and that all bound variables are distinct.

Using Definition 4.1, we can reformulate visitors as let types.

Definition 4.2 If T is a data type with constructors of type $F_i[T] \to T$, its visitor encoding is as follows:

$$T = \mathsf{let}\ \overbrace{\gamma\texttt{<}\beta\texttt{>}\ \mathsf{be}\ \prod_{i\in 1..n}(\underbrace{F_i[\beta]\to\beta}_{\mathrm{visit}_i[\beta]})}^{\mathrm{Visitor}[\beta]}\ \mathsf{in}\ \mathsf{let}\ \overbrace{\delta\ \mathsf{be}\ \forall\alpha.\gamma[\alpha]\to\alpha}^{\mathrm{accept}}\ \mathsf{in}\ \delta$$

It is easy to see that this is equivalent to the System F encoding by a series of reductions in F$_\omega$:

$$\begin{aligned}
T &= \mathsf{let}\ \gamma\texttt{<}\beta\texttt{>}\ \mathsf{be}\ \prod_{i\in 1..n}(F_i[\beta]\to\beta)\ \mathsf{in}\ \mathsf{let}\ \delta\ \mathsf{be}\ \forall\alpha.\gamma[\alpha]\to\alpha\ \mathsf{in}\ \delta\\
&= (\Lambda\gamma{::}{*}\Rightarrow{*}.(\Lambda\delta.\delta)[\forall\alpha.\gamma[\alpha]\to\alpha])[\Lambda\beta.\prod_{i\in 1..n}(F_i[\beta]\to\beta)]\\
&\leadsto_\beta (\Lambda\gamma{::}{*}\Rightarrow{*}.\forall\alpha.\gamma[\alpha]\to\alpha)[\Lambda\beta.\prod_{i\in 1..n}(F_i[\beta]\to\beta)]\\
&\leadsto_\beta \forall\alpha.(\Lambda\beta.\prod_{i\in 1..n}(F_i[\beta]\to\beta))[\alpha]\to\alpha\\
&\leadsto_\beta \forall\alpha.(\prod_{i\in 1..n}(F_i[\alpha]\to\alpha))\to\alpha
\end{aligned}$$

But we can also recover visitors in Generic Java. To do so, we define a translation.

Definition 4.3 The translation $[\![\cdot]\!]$ from types of λ^{oo}_ω to a sequence of FGJ interface definitions as follows: (To render this more compactly we slightly abuse EBNF notation. Repeated occurrences on the LHS correspond to those on the RHS.)

$$\begin{aligned}
[\![(\mathsf{let}\ \alpha\texttt{<}\beta\texttt{>}\ \mathsf{be}\ O\ \mathsf{in})^+\ \alpha[\vec{S}]]\!] &= (\texttt{interface}\ \alpha\texttt{<}\beta\texttt{>}\ \texttt{\{}[\![O]\!]\texttt{\}})^+\\
[\![\textstyle\prod_{i\in 1..n} M_i]\!] &= ([\![M_i]\!])^{i\in 1..n}\\
[\![\forall\vec{\beta}.(\textstyle\prod_{j\in 1..m} S_j)\to S]\!] &= \texttt{<}\vec{\beta}\texttt{>}\ [\![S]\!]\ \texttt{m(}([\![S_j]\!]\ \texttt{x}_j)^{j\in 1..m}\texttt{);}\\
[\![\alpha[\vec{S}]]\!] &= \alpha\texttt{<}[\![\vec{S}]\!]\texttt{>}\\
[\![\mathsf{int}]\!] &= \texttt{Int}
\end{aligned}$$

where $\texttt{m}$ and $\texttt{x}_j$ are fresh.

When considering the transform to FGJ it will be necessary to consider the factors of $F_i[\beta]$ separately. We will take $F_i[\beta]$ to be $\prod_{j\in 1..r-1}\mathsf{int}\times\prod_{k\in r..n}\beta$.

We define the following abbreviations:

$$\begin{aligned}
\texttt{Int}\ \vec{\texttt{f}} &= \texttt{Int f}_1\texttt{, ... , Int f}_{r-1} \quad \text{(similarly for } \texttt{Int}\ \vec{\texttt{x}}\text{)}\\
\texttt{D}\ \vec{\texttt{g}} &= \texttt{D g}_r\texttt{, ... , D g}_n \quad \text{(similarly for } \beta\ \vec{\texttt{g}} \text{ and } S\ \vec{\texttt{y}}\text{)}\\
\texttt{Int}\ \vec{\texttt{f}}\texttt{;} &= \texttt{Int f}_1\texttt{; ... ; Int f}_{r-1}\texttt{;}\\
\texttt{D}\ \vec{\texttt{g}}\texttt{;} &= \texttt{D g}_r\texttt{; ... ; D g}_n\texttt{;} \quad \text{(similarly for } \beta\ \vec{\texttt{g}}\texttt{;}\text{)}\\
\texttt{this.}\vec{\texttt{f}}\texttt{=}\vec{\texttt{f}}\texttt{;} &= \texttt{this.f}_1\texttt{=f}_1\texttt{; ... ; this.f}_{r-1}\texttt{=f}_{r-1}\texttt{;}\\
\texttt{this.}\vec{\texttt{g}}\texttt{=}\vec{\texttt{g}}\texttt{;} &= \texttt{this.g}_r\texttt{=g}_r\texttt{; ... ; this.g}_n\texttt{=g}_n\texttt{;}\\
\texttt{this.}\vec{\texttt{f}} &= \texttt{this.f}_1\texttt{, ... , this.f}_{r-1}\\
\texttt{this.}\vec{\texttt{g}}\texttt{.accept<}\alpha\texttt{>(v)} &= \texttt{this.g}_r\texttt{.accept<}\alpha\texttt{>(v), ... , this.g}_n\texttt{.accept<}\alpha\texttt{>(v)}
\end{aligned}$$

Applying $[\![\cdot]\!]$ to the let encoding of visitors results in

$$[\![\mathsf{let}\ \gamma\texttt{<}\beta\texttt{>}\ \mathsf{be}\ \prod_{i\in 1..n}(F_i[\beta]\to\beta)\ \mathsf{in}\ \mathsf{let}\ \delta\ \mathsf{be}\ \forall\alpha.\gamma[\alpha]\to\alpha\ \mathsf{in}\ \delta]\!]$$

=

```
interface γ<β> {
  (β visit_i(Int x⃗, β y⃗);)^{i∈1..n}
}
interface δ {
  <α> α accept(γ<α> v);
}
```

Proposition 4.4 *The translation from λ_ω^{oo} to FGJ is type-preserving.*

Proof (Sketch) The proof is by induction on the structure of types, keeping track of the interface table generated. □

We have shown that a sequence of Java interfaces can be seen as types in System F_ω. If interfaces are like types then classes implementing interfaces should correspond to terms. This is indeed the case: for each constructor cons_i there is a class $\texttt{Cons}_i$. Similarly, tuples of "visit methods" in F_ω correspond to classes implementing the visitor interface. See Figure 2 for an overview of this correspondence.

We would also like to be sure that both the interface and class definitions are well-typed.

Proposition 4.5 (Internal visitors are well-formed) *Assuming a class table CT and an interface table IT with the class and interface definitions shown in Figure 2, for any Δ, for each $i \in 1..n$,*

(i) $\Delta \vdash U$ *ok*

(ii) $\Delta; \vec{\texttt{x}} : \texttt{Int}, \vec{\texttt{y}} : U, \texttt{this} : \texttt{ConcVis} \vdash e_i \in V$ *and* $\vdash V$ **<:** U

the class and interface definitions in Figure 2 are well-formed.

Proof (Sketch) Straightforward type-checking using the type system of FGJ.□

Internal visitor encoding	**Internal visitor in FGJ**
$T =$ let γ<β> be $\prod_{i\in 1..n}(F_i[\beta]\rightarrow\beta)$ in let δ be $\forall\alpha.\gamma[\alpha]\rightarrow\alpha$ in δ	`interface Visitor<`β`> {` (β `visit`$_i$`(Int` $\vec{\texttt{x}}$`,` β $\vec{\texttt{y}}$`);`)$^{i\in 1..n}$ `}` `interface D {` `<`α`>` α `accept(Visitor<`α`> v);` `}`
$\mathsf{cons}_i \;:\; F_i[T]\rightarrow T$ $\mathsf{cons}_i = \lambda x{:}F_i[T].\;\Lambda\alpha.\lambda v{:}\prod_{j\in 1..n}(F_j[\alpha]\rightarrow\alpha).\; v_i(F_i[\lambda t{:}T.t[\alpha]v]x)$	`class Cons`$_i$ `implements D {` `Int` $\vec{\texttt{f}}$`; D` $\vec{\texttt{g}}$`;` `Cons`$_i$`(Int` $\vec{\texttt{f}}$`, D` $\vec{\texttt{g}}$`) {` `this.`$\vec{\texttt{f}}$ `=` $\vec{\texttt{f}}$`; this.`$\vec{\texttt{g}}$ `=` $\vec{\texttt{g}}$`;` `}` `<`α`>` α `accept(Visitor<`α`> v) {` `return v.visit`$_i$`(this.`$\vec{\texttt{f}}$`,` `this.`$\vec{\texttt{g}}$`.accept<`α`>(v));` `}` `}`
$\mathsf{s} \;:\; \prod_{i\in 1..n}(F_i[U]\rightarrow U)$ $\mathsf{s} = \langle\lambda x{:}F_i[U].e_i\rangle_{i\in 1..n}$	`class ConcVis implements Visitor<`U`> {` (U `visit`$_i$`(Int` $\vec{\texttt{x}}$`,` U $\vec{\texttt{y}}$`) {return` e_i`;}`)$^{i\in 1..n}$ `}`

Fig. 2. Correspondence between type encodings and internal visitors in FGJ.

As a worked example, we consider the type B of binary trees with integers at the leaves, as given by the least fixed point of $F[X] = \mathbb{Z} + X \times X$. This is encoded as

$$\mathsf{B} = \mathsf{let}\ \gamma\ \mathsf{be}\ \Lambda\beta.(\mathbb{Z}\rightarrow\beta)\times((\beta\times\beta)\rightarrow\beta)\ \mathsf{in\ let}\ \delta\ \mathsf{be}\ \forall\alpha.\gamma[\alpha]\rightarrow\alpha\ \mathsf{in}\ \delta$$

which is transformed into

```
interface Visitor<β> {
  β visitLeaf(Int x);
  β visitNode(β x, β y);
}
interface BinTree {
  <α> α accept(Visitor<α> v);
}
```

The datatype constructors are

$$\begin{aligned}
\mathsf{leaf} &: \mathbb{Z}\rightarrow\mathsf{B}\\
\mathsf{leaf}(n) &= \Lambda\alpha.\lambda\langle p,q\rangle{:}(\mathbb{Z}\rightarrow\alpha)\times((\alpha\times\alpha)\rightarrow\alpha).pn\\
\mathsf{node} &: (\mathsf{B}\times\mathsf{B})\rightarrow\mathsf{B}\\
\mathsf{node}(l,r) &= \Lambda\alpha.\lambda\langle p,q\rangle{:}(\mathbb{Z}\rightarrow\alpha)\times((\alpha\times\alpha)\rightarrow\alpha).q\langle l[\alpha]\langle p,q\rangle, r[\alpha]\langle p,q\rangle\rangle
\end{aligned}$$

The corresponding FGJ classes `Leaf` and `Node` are

```
class Leaf implements BinTree {
  Int n;
  Nil(Int n) { this.n = n; }
  <α> α accept(Visitor<α> v) {
    return v.visitLeaf();
  }
}
class Node implements BinTree {
  BinTree l; BinTree r;
  Cons(BinTree l, BinTree r) { this.l = l; this.r = r; }
  <α> α accept(Visitor<α> v) {
    return v.visitNode(l.accept<α>(v), r.accept<α>(v));
  }
}
```

The type of a concrete visitor for binary trees is

$$\mathsf{let}\ \gamma\ \mathsf{be}\ (\mathbb{Z} \rightarrow \mathbb{Z}) \times ((\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z})\ \mathsf{in}\ \gamma$$

An operation for summing up the leaves of a tree can defined as follows:

$$\mathsf{sum}\ :\ \mathsf{B} \rightarrow \mathbb{Z}$$

$$\mathsf{sum} = \lambda t{:}T.t[\mathbb{Z}]\langle \lambda x{:}\mathbb{Z}.x, \lambda\langle x,y\rangle{:}\mathbb{Z}\times\mathbb{Z}.x+y\rangle$$

which is equivalent to a call to accept on a `BinTree` with an instance of the following concrete visitor

```
class SumVisitor implements Visitor<Int> {
  Int visitLeaf(Int x) {
    return x;
  }
  Int visitNode(Int x, Int y) {
    return x + y;
  }
}
```

The generated code is valid, well-typed FGJ with interfaces code. This means it is also almost correct Generic Java. Indeed, after minor modifications (adding the keyword `public` to method definitions, discarding type parameter instantiation on calls to accept and adding a `main` method), it compiles correctly using the Sun Java 1.5 compiler.

5 From functional to imperative internal visitors

The visitors in Figure 2 were purely functional and relied on generics. A more typical rendition of an internal visitor using internal state instead is given in Figure 3 (it is debatable whether `visitNode()` should be omitted, since there

```
interface Visitor {
    void visitLeaf(int n);
    void visitNode();
}
interface BinTree {
    void accept(Visitor v);
}
class Leaf implements BinTree {
    int n;
    Leaf(int n) { this.n = n; }
    public void accept(Visitor v) {
        v.visitLeaf(n);
    }
}
class Node implements BinTree {
    BinTree left; BinTree right;
    Node(BinTree left, BinTree right) {
        this.left = left;
        this.right = right;
    }
    public void accept(Visitor v) {
        left.accept(v);
        right.accept(v);
    }
}
class SumVisitor implements Visitor {
    int s;
    SumVisitor(int s) { this.s = s; }
    public void visitLeaf(int n) { s = s + n; }
    public void visitNode() { }
}
```

Fig. 3. An imperative internal visitor in Java (without generics)

is no information at internal nodes).

In this section, we address the equivalence of functional and imperative visitors, albeit in a very idealized setting. Consider the example of summing all the leaves of a binary tree. Functional visitors would do this by performing additions at all the internal nodes. On the other hand, a more typical imperative use of the Visitor pattern would use a field in the visitor to accumulate the sum. The field is initialized to 0, and at each leaf, its value is added to the field; at an internal node, the left and right subtrees are simply traversed one after the other. After the traversal, the field holds the result. Since such an imperative visitor uses state rather than a result value, its return type is `void`.

We will model a visitor that updates a piece of state S by a function $S\to S$, as one does in the semantics of imperative languages, or the monadic view of computation approach. Given (the functional internal visitor encoding of) a binary tree t, we define its imperative counterpart $\widehat{t}$ as follows:

$$\widehat{t} = \Lambda\alpha.\lambda c{:}\mathbb{Z}\to\alpha\to\alpha.t[\alpha\to\alpha]\langle c,\circ_\alpha\rangle$$

where $\circ$ abbreviates function composition,

$$\circ_\alpha = \lambda\langle f,g\rangle{:}(\alpha\to\alpha)\times(\alpha\to\alpha).\lambda x{:}\alpha.g(fx)$$

Compared to t, $\widehat{t}$ is more specialized: it only allows one to specify an operation at the leaves, which needs to transform a state of type α, while the only operation at internal nodes is to compose the state transformers of the left and right subtrees; this composition of state transformations corresponds to calling two `void`-returning methods in succession.

To relate the state-transforming and the functional visitors, we want to show that the state-transforming visitor (with initial state 0) yields the same result as the functional one:

$$\widehat{t}\,[\mathbb{Z}]\,(\lambda n{:}\mathbb{Z}.\lambda s{:}\mathbb{Z}.s+n)\,0 \quad = \quad t[\mathbb{Z}]\langle \mathrm{id}_{\mathbb{Z}},+\rangle$$

We sketch a proof using relational parametricity [13], specifically the reasoning developed by Wadler as “theorems for free” [16]; see the latter paper for an introduction and the relevant definitions.

Let t be a binary tree for internal visitors, that is

$$t : \forall\alpha.((\mathbb{Z}\to\alpha)\times((\alpha\times\alpha)\to\alpha))\to\alpha$$

Now $\widehat{t}\,[\mathbb{Z}]\,(\lambda n{:}\mathbb{Z}.\lambda s{:}\mathbb{Z}.s+n) = t[\mathbb{Z}\to\mathbb{Z}]\,\langle(\lambda n{:}\mathbb{Z}.\lambda s{:}\mathbb{Z}.s+n),\circ_{\mathbb{Z}}\rangle$. Hence we want to show

$$t[\mathbb{Z}]\langle \mathrm{id}_{\mathbb{Z}},+\rangle = t[\mathbb{Z}\to\mathbb{Z}]\langle \mathrm{add}_{\mathbb{Z}},\circ_{\mathbb{Z}}\rangle\,0$$

where

$$\begin{aligned}\mathrm{id}_{\mathbb{Z}} &= \lambda n{:}\mathbb{Z}.n && : \mathbb{Z}\to\mathbb{Z}\\ \mathrm{add}_{\mathbb{Z}} &= \lambda n{:}\mathbb{Z}.\lambda s{:}\mathbb{Z}.s+n && : \mathbb{Z}\to\mathbb{Z}\to\mathbb{Z}\end{aligned}$$

Assuming parametricity [16], we have that for all relations R,

$$\langle t,t\rangle \in ((\mathbb{Z}\to R)\times((R\times R)\to R))\to R$$

We define a relation $R:\mathbb{Z}\leftrightarrow(\mathbb{Z}\to\mathbb{Z})$ by $\langle n,f\rangle\in R$ iff $f(x)=n+x$ for all x. Then we have:

$$\begin{aligned}\langle \mathrm{id}_{\mathbb{Z}},\mathrm{add}_{\mathbb{Z}}\rangle &\in \mathbb{Z}\to R\\ \langle +,\circ_{\mathbb{Z}}\rangle &\in (R\times R)\to R\end{aligned}$$

The latter of these holds because if $\langle\langle x,y\rangle,\langle f,g\rangle\rangle\in R\times R$, then $\langle x{+}y, f{\circ}g\rangle{\in}R$.

Since t maps related arguments to related results, we have

$$\langle t[\mathbb{Z}]\langle \mathrm{id}_{\mathbb{Z}},+\rangle, t[\mathbb{Z}\to\mathbb{Z}]\langle \mathrm{add}_{\mathbb{Z}},\circ_{\mathbb{Z}}\rangle\rangle\in R$$

Hence, by the definition of R, $t[\mathbb{Z}\to\mathbb{Z}]\langle \mathrm{add}_{\mathbb{Z}},\circ_{\mathbb{Z}}\rangle\,0 = t[\mathbb{Z}]\langle \mathrm{id}_{\mathbb{Z}},+\rangle$, as required.

Note that the proof made use of 0 being the neutral element of addition, and of the associativity of addition in establishing the relation R between $f \circ g$ and $x + y$:

$$(f \circ g)(z) = f(g(z)) = f(z + y) = (z + y) + x = z + (x + y)$$

The above argument of relating a functional and an imperative visitor by associativity is applicable to more substantial cases as well. Consider the standard example of abstract syntax trees, and suppose we need to traverse the tree to add information into a symbol table. The most evident specification in terms of a synthesized attribute would be essentially functional, merging the symbol table of the subtrees at the inner nodes. An imperative visitor could instead start off with an empty symbol table and add entries by updating a mutable symbol table during traversal. Showing the equivalence of the functional and the imperative version should be analogous to the parametricity argument above, with the empty symbol table being the neutral element, and merging of symbol tables as the associative operation.

6 Conclusions

We have reconstructed the internal Visitor pattern and a restricted form of the external variant within an idealized type-theoretic setting. To do so, we had to make simplifying assumptions, and the fit is not perfect; this may be inevitable, since Java was not designed on top of a lambda calculus, unlike functional languages. But if one grants these idealizations, it is possible to glean essential features of visitors more easily than from lengthy Java code or class diagrams. We summarize our abstract reconstruction of visitors with the following mapping from the terminology used in the patterns literature [6].

Patterns literature	Idealized view
Visitor interface	Type (determines a functor F)
Concrete visitor	Object of visitor type ($\cong$ F-algebra)
Visit method	Component of the structure map of a visitor
Data (or "Element")	Weakly initial visitor
Accept method	Witnesses initiality
Concrete data	Given by cons_i

There is a large literature on the Visitor pattern, most of it concerned with overcoming some of its inflexibility. This line of work goes back to Reynolds [12]. In the context of the present paper, the most relevant previous work is Felleisen and Friedman's textbook [4], in which variations on the Visitor pattern are developed, implicitly based on program transformations known from functional programming. Setzer [15] also observes a connection between visitors and functional programming.

As sketched in Section 5, for functional (particularly internal) visitors, the polymorphic typing immediately gives one reasoning principles, or "theorems for free". It should be possible to transfer the structure of such arguments to more realistic imperative visitors, where logical relations on parts of the heap would be used in place of the return type parametricity of the functional visitors. Instead of the result types of visitors, the relations could be built on an effect system [10] (adapted from functional to object-oriented languages [1]). In particular, the effect annotations tells us that the data structure to be traversed unleashes the effects of the visitor, but causes none itself. Apart from effect systems, Hoare logic may also be applicable to visitors. Specifically, it would be interesting to see whether abstract predicates [11] for visitors can be handled analogously to the quantified result type.

The visitor-style encoding in System F does not extend to datatypes that use subclassing. It may be worthwhile to consider similar encodings in $F_\omega^{<:}$ to see whether it fits such Visitor pattern variants more closely.

Acknowledgement

We thank Alan Mycroft and the anonymous referees for their comments.

References

[1] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: an imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.

[2] C. Böhm and C. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39(2/3):135–154, 1985.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98), Vancouver, British Columbia, 18–22 October 1998*, pages 183–200. ACM Press, New York, 1998.

[4] Matthias Felleisen and Daniel P. Friedman. *A Little Java, A Few Patterns.* MIT Press, Cambridge, Massachusetts, 1998.

[5] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems, Santa Barbara, California, 3–7 August 1998*, pages 140–154. IEEE Computer Society, Washington DC, 1998.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, Massachusetts, 1995.

[7] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

[8] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, Cambridge, 1989.

[9] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99), Denver, Colorado, 1–5 November 1999*, pages 132–146. ACM Press, New York, 1999.

[10] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL'88), San Diego, California, 13–15 January 1988*, pages 47–57. ACM Press, New York, 1988.

[11] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'05), Long Beach, California, 12–14 January 2005*, pages 247–258. ACM Press, New York, 2005.

[12] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages 1975*, pages 157–168. IRIA, Rocquencourt, France, 1976.

[13] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1983.

[14] John C. Reynolds and Gordon D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105:1–29, 1993.

[15] Anton Setzer. Java as a functional programming language. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs: International Workshop, TYPES 2002, Berg en Dal, The Netherlands, 24–28 April 2002*, number 2646 in Lecture Notes in Computer Science, pages 279–298. Springer, Berlin, 2003.

[16] Philip Wadler. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'89), London, 11–13 September 1989*, pages 347–359. ACM Press, New York, 1989.

A Featherweight Generic Java

Featherweight Generic Java [9] is a minimal Java-like calculus. It includes type parameterized classes and methods and its syntax is (almost) a subset of Java, so all FGJ programs are also valid Java programs. (To convert an

FGJ program into a Generic Java program it is necessary to add the keyword `public` before method implementations and to elide type instantiations on method calls.)

Our description of FGJ includes interfaces which were not present in the original definition. The extension is limited since it only permits single inheritance in interface hierarchies and a class is permitted to implement at most one interface. There are also some restrictions on type parameter bounds. Since we are not interested in casting we will omit that from our account.

The syntax and typing of FGJ with interfaces are shown in Figures A.1 and A.2. The computation rules are shown in Figure A.3. We omit some auxiliary rules due to lack of space. We abbreviate the keywords `extends` as $\triangleleft$, `implements` as $\diamond$ and `return` as $\uparrow$. The metavariables α, β, γ range over type variables; `T`, `U`, `V` range over types; `N` and `O` range over class types; and `Q` ranges over interface types.

Some abbreviations are also necessary for various sequences:

$$
\begin{aligned}
\vec{\mathtt{f}} &= \mathtt{f_0,\ldots,f_n} \quad \text{(similarly for } \vec{\mathtt{C}}, \vec{\mathtt{x}}, \vec{\mathtt{e}}, \vec{\alpha}, \vec{\mathtt{N}}, \vec{\mathtt{Q}} \text{ etc.)}\\
\vec{\mathtt{M}} &= \mathtt{M_0 \ldots M_n} \quad \text{(similarly for } \vec{\mathtt{H}}\text{)}\\
\vec{\mathtt{C}}\ \vec{\mathtt{f}} &= \mathtt{C_1\ f_1, \ldots, C_n\ f_n}\\
\vec{\mathtt{C}}\ \vec{\mathtt{f}}; &= \mathtt{C_1\ f_1; \ldots; C_n\ f_n;}\\
\mathtt{this.}\vec{\mathtt{f}}\mathtt{=}\vec{\mathtt{f}}\mathtt{;} &= \mathtt{this.f_1=f_1; \ldots; this.f_n=f_n;}
\end{aligned}
$$

The empty sequence is written as $\bullet$ and concatenation is denoted with a comma.

Unparameterized classes `C<>` and methods `m<>` can be abbreviated to `C` and `m`. As in Java, unbounded parameters are assumed to have a bound of `Object`. We also abbreviate `C`$\triangleleft$`Object` to `C`. A class does not have to implement an interface, in which case we can omit $\diamond$`Q` from its definition. Unlike the class hierarchy, the interface hierarchy has no root so we also allow $\triangleleft$`Q` to be omitted from interface definitions. We will omit empty calls to `super()` and empty constructors.

The class table *CT* is a mapping from class names to class declarations. The extended calculus also has an interface table *IT* which plays a similar role with respect to interfaces. The authors of FGJ define some "sanity conditions" on class tables and we assume that both *CT* and *IT* obey them. We assume the existence of the special variable `this`, which may not be used as the name of a field or method parameter. A program in FGJ with interfaces is a triple $(CT, IT, \mathtt{e})$ of a class table, an interface table and an expression.

Syntax

$$\begin{aligned}
\mathtt{CL} &::= \mathtt{class\ C<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\diamond\mathtt{Q}\ \{\vec{\mathtt{T}}\ \vec{\mathtt{f}}\mathtt{;}\ \mathtt{K}\ \vec{\mathtt{M}}\} \\
\mathtt{IN} &::= \mathtt{interface\ I<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>}\lhd\mathtt{Q}\ \{\vec{\mathtt{H}}\} \\
\mathtt{K} &::= \mathtt{C(}\vec{\mathtt{T}}\ \vec{\mathtt{f}}\mathtt{)}\ \{\mathtt{super(}\vec{\mathtt{f}}\mathtt{);\ this.}\vec{\mathtt{f}}\ \mathtt{=}\ \vec{\mathtt{f}}\mathtt{;}\} \\
\mathtt{H} &::= \mathtt{<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>\ T\ m\ (}\vec{\mathtt{T}}\ \vec{\mathtt{x}}\mathtt{);} \\
\mathtt{M} &::= \mathtt{<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>\ T\ m\ (}\vec{\mathtt{T}}\ \vec{\mathtt{x}}\mathtt{)}\ \{\uparrow\mathtt{e;}\}
\end{aligned}$$

$$\begin{aligned}
\mathtt{e} &::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m<}\vec{\mathtt{T}}\mathtt{>(}\vec{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ N(}\vec{\mathtt{e}}\mathtt{)} \\
\mathtt{T} &::= \alpha \mid \tau \mid \mathtt{N} \mid \mathtt{Q} \\
\mathtt{N} &::= \mathtt{C<}\vec{\mathtt{T}}\mathtt{>} \\
\mathtt{Q} &::= \mathtt{I<}\vec{\mathtt{T}}\mathtt{>}
\end{aligned}$$

Subtyping

$$\frac{}{\Delta\vdash\mathtt{T}\ \mathtt{<:}\ \mathtt{T}}\ \textsc{S-Refl} \qquad \frac{}{\Delta\vdash\alpha\ \mathtt{<:}\ \Delta(\alpha)}\ \textsc{S-Poly}$$

$$\frac{\Delta\vdash\mathtt{S}\ \mathtt{<:}\ \mathtt{T}\qquad\Delta\vdash\mathtt{T}\ \mathtt{<:}\ \mathtt{U}}{\Delta\vdash\mathtt{S}\ \mathtt{<:}\ \mathtt{U}}\ \textsc{S-Trans} \qquad \frac{\mathit{IT}(\mathtt{I}) = \mathtt{interface\ I<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>}\lhd\mathtt{Q}\ \{\ldots\}}{\Delta\vdash\mathtt{I<}\vec{\mathtt{T}}\mathtt{>}\ \mathtt{<:}\ [\vec{\alpha}\mapsto\vec{\mathtt{T}}]\,\mathtt{Q}}\ \textsc{S-Sub}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\diamond\mathtt{Q}\ \{\ldots\}}{\Delta\vdash\mathtt{C<}\vec{\mathtt{T}}\mathtt{>}\ \mathtt{<:}\ [\vec{\alpha}\mapsto\vec{\mathtt{T}}]\,\mathtt{N}}\ \textsc{S-Extend}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\diamond\mathtt{Q}\ \{\ldots\}}{\Delta\vdash\mathtt{C<}\vec{\mathtt{T}}\mathtt{>}\ \mathtt{<:}\ [\vec{\alpha}\mapsto\vec{\mathtt{T}}]\,\mathtt{Q}}\ \textsc{S-Impl}$$

Well-formed types

$$\frac{}{\Delta\vdash\mathtt{Object}\ \text{ok}}\ \textsc{WF-Obj} \qquad \frac{}{\Delta\vdash\varepsilon\ \text{ok}}\ \textsc{WF-Empty}$$

$$\frac{}{\Delta\vdash\mathtt{Int}\ \text{ok}}\ \textsc{WF-Int} \qquad \frac{\alpha\in\mathit{dom}(\Delta)}{\Delta\vdash\alpha\ \text{ok}}\ \textsc{WF-Poly}$$

$$\frac{\begin{array}{c}\mathit{IT}(\mathtt{I}) = \mathtt{interface\ I<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>}\lhd\mathtt{Q}\ \{\ldots\}\\ \Delta\vdash\vec{\mathtt{T}}\ \text{ok}\qquad\Delta\vdash\vec{\mathtt{T}}\ \mathtt{<:}\ [\vec{\alpha}\mapsto\vec{\mathtt{T}}]\,\vec{\mathtt{N}}\end{array}}{\Delta\vdash\mathtt{I<}\vec{\mathtt{T}}\mathtt{>}\ \text{ok}}\ \textsc{WF-Interface}$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<}\vec{\alpha}\lhd\vec{\mathtt{N}}\mathtt{>}\lhd\mathtt{N}\diamond\mathtt{Q}\ \{\ldots\}\\ \Delta\vdash\vec{\mathtt{T}}\ \text{ok}\qquad\Delta\vdash\vec{\mathtt{T}}\ \mathtt{<:}\ [\vec{\alpha}\mapsto\vec{\mathtt{T}}]\,\vec{\mathtt{N}}\end{array}}{\Delta\vdash\mathtt{C<}\vec{\mathtt{T}}\mathtt{>}\ \text{ok}}\ \textsc{WF-Class}$$

Expression typing

$$\frac{}{\Delta;\Gamma\vdash\mathtt{x}\in\Gamma(\mathtt{x})}\ \textsc{T-Var}$$

$$\frac{\Delta;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0\qquad\mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \vec{\mathtt{T}}\ \vec{\mathtt{f}}}{\Delta;\Gamma\vdash\mathtt{e}_0\mathtt{.f}_\mathtt{i}\in\mathtt{T}_\mathtt{i}}\ \textsc{T-Field}$$

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0\\ \mathit{mtype}(\mathtt{m},\mathit{bound}_\Delta(\mathtt{T}_0)) = \mathtt{<}\vec{\beta}\lhd\vec{\mathtt{O}}\mathtt{>}\vec{\mathtt{U}}\to\mathtt{U}\\ \Delta\vdash\vec{\mathtt{V}}\ \text{ok}\qquad\Delta\vdash\vec{\mathtt{V}}\ \mathtt{<:}\ [\vec{\beta}\mapsto\vec{\mathtt{V}}]\,\vec{\mathtt{O}}\\ \Delta;\Gamma\vdash\vec{\mathtt{e}}\in\vec{\mathtt{S}}\qquad\Delta\vdash\vec{\mathtt{S}}\ \mathtt{<:}\ [\vec{\beta}\mapsto\vec{\mathtt{V}}]\,\vec{\mathtt{U}}\end{array}}{\Delta;\Gamma\vdash\mathtt{e}_0\mathtt{.m<}\vec{\mathtt{V}}\mathtt{>(}\vec{\mathtt{e}}\mathtt{)}\in[\vec{\beta}\mapsto\vec{\mathtt{V}}]\,\mathtt{U}}\ \textsc{T-Invk} \qquad \frac{\begin{array}{c}\Delta\vdash\mathtt{N}\ \text{ok}\qquad\mathit{fields}(\mathtt{N}) = \vec{\mathtt{T}}\ \vec{\mathtt{f}}\\ \Delta;\Gamma\vdash\vec{\mathtt{e}}\in\vec{\mathtt{S}}\qquad\Delta\vdash\vec{\mathtt{S}}\ \mathtt{<:}\ \vec{\mathtt{T}}\end{array}}{\Delta;\Gamma\vdash\mathtt{new\ N(}\vec{\mathtt{e}}\mathtt{)}\in\mathtt{N}}\ \textsc{T-New}$$

Fig. A.1. FGJ sans casting extended with interfaces: Main definitions

Method typing

$$\dfrac{\begin{array}{c}\Delta = \vec{\alpha}\ \texttt{<:}\ \vec{\texttt{N}}, \vec{\beta}\ \texttt{<:}\ \vec{\texttt{O}} \\ \Delta \vdash \vec{\texttt{T}}\ \text{ok} \qquad \Delta \vdash \texttt{T}\ \text{ok} \qquad \Delta \vdash \vec{\texttt{O}}\ \text{ok} \\ \Delta; \vec{\texttt{x}} : \vec{\texttt{T}}, \texttt{this} : \texttt{C<}\vec{\alpha}\texttt{>} \vdash \texttt{e}_0 \in \texttt{S} \qquad \Delta \vdash \texttt{S}\ \texttt{<:}\ \texttt{T} \\ CT(\texttt{C}) = \texttt{class C<}\vec{\alpha} \triangleleft \vec{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \diamond \texttt{Q}\ \{\ldots\} \\ \mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\vec{\gamma} \triangleleft \vec{\texttt{P}}\texttt{>}\vec{\texttt{U}} \to \texttt{U})\end{array}}{\texttt{<}\vec{\beta} \triangleleft \vec{\texttt{O}}\texttt{> T m (}\vec{\texttt{T}}\ \vec{\texttt{x}}\texttt{)}\ \{\uparrow \texttt{e}_0\texttt{;}\}\ \texttt{OK IN C<}\vec{\alpha} \triangleleft \vec{\texttt{N}}\texttt{>}}\ \textsc{T-Method}$$

Method header typing

$$\dfrac{\begin{array}{c}\Delta = \vec{\alpha}\ \texttt{<:}\ \vec{\texttt{N}}, \vec{\beta}\ \texttt{<:}\ \vec{\texttt{O}} \\ \Delta \vdash \vec{\texttt{T}}\ \text{ok} \qquad \Delta \vdash \texttt{T}\ \text{ok} \qquad \Delta \vdash \vec{\texttt{O}}\ \text{ok} \\ IT(\texttt{I}) = \texttt{interface I<}\vec{\alpha} \triangleleft \vec{\texttt{N}}\texttt{>} \triangleleft \texttt{Q}\ \{\ldots\} \\ \mathit{override}(\texttt{m}, \texttt{Q}, \texttt{<}\vec{\gamma} \triangleleft \vec{\texttt{P}}\texttt{>}\vec{\texttt{U}} \to \texttt{U})\end{array}}{\texttt{<}\vec{\beta} \triangleleft \vec{\texttt{O}}\texttt{> T m (}\vec{\texttt{T}}\ \vec{\texttt{x}}\texttt{); OK IN I<}\vec{\alpha} \triangleleft \vec{\texttt{N}}\texttt{>}}\ \textsc{T-Mhead}$$

Class typing

$$\dfrac{\begin{array}{c}\vec{\alpha}\ \texttt{<:}\ \vec{\texttt{N}} \vdash \vec{\texttt{N}}\ \text{ok} \qquad \vec{\alpha}\ \texttt{<:}\ \vec{\texttt{N}} \vdash \texttt{N}\ \text{ok} \qquad \vec{\alpha}\ \texttt{<:}\ \vec{\texttt{N}} \vdash \texttt{Q}\ \text{ok} \\ \vec{\alpha}\ \texttt{<:}\ \vec{\texttt{N}} \vdash \vec{\texttt{T}}\ \text{ok} \qquad \mathit{implement}((\vec{\texttt{M}}_0, \vec{\texttt{M}}_1), \vec{\texttt{H}}) \\ \mathit{methods}(\texttt{N}) = \vec{\texttt{M}}_1 \qquad \mathit{mheaders}(\texttt{Q}) = \vec{\texttt{H}} \\ \mathit{fields}(\texttt{N}) = \vec{\texttt{U}}\ \vec{\texttt{g}} \qquad \vec{\texttt{M}}_0, \vec{\texttt{M}}_1\ \texttt{OK IN C<}\vec{\alpha} \triangleleft \vec{\texttt{N}}\texttt{>} \\ \texttt{K} = \texttt{C(}\vec{\texttt{U}}\ \vec{\texttt{g}}\texttt{, }\vec{\texttt{T}}\ \vec{\texttt{f}}\texttt{)}\ \{\texttt{super(}\vec{\texttt{g}}\texttt{); this.}\vec{\texttt{f}}\texttt{ = }\vec{\texttt{f}}\texttt{;}\}\end{array}}{\texttt{class C<}\vec{\alpha} \triangleleft \vec{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \diamond \texttt{Q}\ \{\vec{\texttt{T}}\ \vec{\texttt{f}}\texttt{; K }\vec{\texttt{M}}_0\}\ \texttt{OK}}\ \textsc{T-Class}$$

Interface typing

$$\dfrac{\begin{array}{c}\vec{\alpha}\ \texttt{<:}\ \vec{\texttt{N}} \vdash \vec{\texttt{N}}\ \text{ok} \qquad \vec{\alpha}\ \texttt{<:}\ \vec{\texttt{N}} \vdash \texttt{Q}\ \text{ok} \\ \vec{\texttt{H}}\ \texttt{OK IN I<}\vec{\alpha} \triangleleft \vec{\texttt{N}}\texttt{>}\end{array}}{\texttt{interface I<}\vec{\alpha} \triangleleft \vec{\texttt{N}}\texttt{>} \triangleleft \texttt{Q}\ \{\vec{\texttt{H}}\}\ \texttt{OK}}\ \textsc{T-Interface}$$

Fig. A.2. FGJ sans casting extended with interfaces: Main definitions continued

Computation

$$\dfrac{\mathit{fields}(\texttt{N}) = \vec{\texttt{T}}\ \vec{\texttt{f}}}{\texttt{(new N(}\vec{\texttt{e}}\texttt{)).f}_i \rightsquigarrow \texttt{e}_i}\ \textsc{Comp-Field}$$

$$\dfrac{\mathit{mbody}(\texttt{m<}\vec{\texttt{V}}\texttt{>}, \texttt{N}) = (\vec{\texttt{x}}, \texttt{e}_0)}{\texttt{(new N(}\vec{\texttt{e}}\texttt{)).m<}\vec{\texttt{V}}\texttt{>(}\vec{\texttt{d}}\texttt{)} \rightsquigarrow [\vec{\texttt{x}} \mapsto \vec{\texttt{d}}, \texttt{this} \mapsto \texttt{new N(}\vec{\texttt{e}}\texttt{)}]\, \texttt{e}_0}\ \textsc{Comp-Invk}$$

Fig. A.3. FGJ sans casting extended with interfaces: Computation rules