Higher-Order Symb Comput (2008) 21: 239–282
DOI 10.1007/s10990-008-9032-6

# Types and trace effects for object orientation

**Christian Skalka**

Published online: 30 September 2008
© Springer Science+Business Media, LLC 2008

**Abstract** *Trace effects* are statically generated program abstractions, that can be model checked for verification of assertions in a temporal program logic. In this paper we develop a type and effect analysis for obtaining trace effects of Object Oriented programs in Featherweight Java. We observe that the analysis is significantly complicated by the interaction of trace behavior with inheritance and other Object Oriented features, particularly overridden methods, dynamic dispatch, and downcasting. We propose an expressive type and effect inference algorithm combining polymorphism and subtyping/subeffecting constraints to obtain a flexible trace effect analysis in this setting, and show how these techniques are applicable to Object Oriented features. We also extend the basic language model with exceptions and stack-based event contexts, and show how trace effects scale to these extensions by structural transformations.

**Keywords** Static type analysis · Programming language-based security · Temporal program logic · Object oriented programming

## 1 Introduction

Program type analysis and model checking have a shared goal: to statically enforce properties of programs. A variety of analyses have been proposed to enforce specific properties, while certain frameworks provide flexibility to enforce a general class of properties. A number of authors [23, 27, 38] have observed that these two approaches can play complementary roles in the verification of general *trace-based* program properties, which are properties of program *event traces* expressible in temporal logic. Type systems can be used to compute program abstractions, which can in turn be used as inputs to model checking [40]. In other words, type analysis can serve as a technique for model extraction [22], for the subsequent verification of a general class of program properties. This paper establishes a foundational theory of type and trace effects for Object Oriented programs in a language model adapted

C. Skalka (✉)
University of Vermont, Burlington, USA
e-mail: skalka@cs.uvm.edu

from Featherweight Java [24], defines a type inference system for automatically reconstructing sound type and trace effects of programs, and shows how trace effect representations can be manipulated to reflect control flow operations such as exceptions.

Trace-based program properties are properties of event traces, where events are records of program actions, explicitly inserted into program code either manually (by the programmer) or automatically (by the compiler). Events are intended to be sufficiently abstract to represent a variety of program actions—e.g. opening a file, access control privilege activation, or entry to or exit from critical regions. Event traces maintain the ordered sequences of events that occur during program execution. Assertions enforce properties of event traces—e.g. certain privileges should be activated before a file can be opened. Results in [7, 27, 37] have demonstrated that static approximations of program event traces can be generated by type and effect analyses [3, 41], in a form amenable to existing model-checking techniques for verification. We call these approximations *trace effects*.

Trace-based analyses have been shown capable of statically enforcing flow-sensitive security properties such as safe locking behavior [19] and resource usage policies such as file usage protocols and memory management [23, 27]. In [5], a trace effect analysis is used to enforce secure service composition. The history-based access control model of [1] can be implemented with event traces and checks [37], as can the policies realizable in that model, e.g. sophisticated Chinese Wall policies [1]. Stack-based security policies are also amenable to this form of analysis, as shown in [37, 38] and this paper. In short, the combination of a primitive notion of program events with a temporal program logic for asserting properties of event traces yields a powerful and general tool for enforcing program properties.

The analyses cited above have been developed in functional language settings, but practical use of these tools require adaptation to realistic languages. In this paper we address technical considerations for application of trace effects to Object Oriented languages, particularly Java. As discussed more thoroughly in Sect. 2, inheritance, dynamic dispatch, and downcasts present significant challenges to trace effect analysis. The effect is that scaling the analysis to Object Oriented programs is a foundational problem, not simply an engineering one. To study these issues in isolation, we extend Featherweight Java (FJ) [24] with events, traces, and checks, and a polymorphic type and effect inference analysis for static enforcement of checks, yielding the language $\mathrm{FJ}_{\mathrm{trace}}$. Technical results in this paper extend and enhance material presented in [35] and [38].

### 1.1 A flexible type analysis

Type theory provides a variety of useful tools in this setting. As will be discussed in detail in later sections, a combination of parametric polymorphism and subtyping can be used to provide the right abstractions and flexibility for addressing issues associated with inheritance and dynamic dispatch, and type constraint representation allows a precise treatment of object downcasting. We extend an Object Oriented type language with *trace effects* that approximate trace behavior of programs. Trace effects are a form of label transition system (LTS), which are amenable to model checking. Therefore, the type analysis serves as a technique for extracting a verifiable abstraction of program trace behavior, with type inference automating the process. We also show that trace effect representations are amenable to transformations that reflect the impact of control flow modifications on trace behavior, particularly exceptions.

The metatheory of types also provides an appealing language for characterizing the analysis and proving its correctness. A type safety result guarantees that programs satisfying the analysis will not have run-time errors, in particular all specified properties of

program traces are guaranteed to hold. This result is established via subject reduction and progress arguments. The type and effect system is shown to be a conservative extension of the underlying Featherweight Java type system, ensuring backwards compatibility with existing programs. We show that type inference is sound, and so-called *trace approximation* is demonstrated, formalizing the idea that trace effects conservatively approximate program trace behavior. We also develop an extended language model with exceptions, and show that a post-processing transformation of inferred trace effects is sufficient to capture the effect of exceptions on control flow, via another type safety result.

### 1.2 Outline of the paper

The remainder of the paper is organized as follows. In Sect. 2, the central issues of our type and effect analysis in relation to Object Oriented programming are described and discussed, clarifying the contribution of this paper. In Sect. 3, the $\mathrm{FJ}_{\mathrm{trace}}$ language is formally defined, which is FJ extended with primitives for a security logic of program traces. In Sect. 4, we formalize the language and meaning of trace effects. In Sect. 5 a logical type system for $\mathrm{FJ}_{\mathrm{trace}}$ is presented, with features and examples discussed in Sect. 5.2.1 and Sect. 5.3.1, and properties including type safety proved in Sect. 5.4. A type inference algorithm is defined in Sect. 6, that is shown to be sound with respect to the logical type system in Sect. 6.4, implying type safety in the implementation. In Sect. 7, we study variations on the basic language model, including exceptions in Sect. 7.1 and stack based trace contexts in Sect. 7.2. We conclude with more discussion of related work and a final summary in Sect. 8.

## 2 Trace effects and object orientation

Subtyping is a common discipline for relating behavior of objects in an inheritance hierarchy. However, as we illustrate below, imposing a subsumption relation on the trace behavior of methods in an inheritance hierarchy is overly restrictive for applications such as access control. It is possible and useful to extend the definition of subtyping to trace effects, as we do in Sect. 5, but a realistic analysis requires that we develop some mechanism for allowing independence of inheritance and effects, and accommodate this independence in the presence of dynamic dispatch. We propose the use of *parametric polymorphism* for this purpose (though we note that the general idea of effect polymorphism is not new, see for example [42]). We also propose a *type constraint* representation; along with known benefits of this approach in application to Object Oriented programming [10, 17], we show how type constraints can be used for a novel soft-typing of downcasts. In this section we discuss and illustrate these issues, before providing formal details in Sect. 5.

### 2.1 Effects and inheritance

The manner in which inheritance and dynamic dispatch complicates trace effect analysis is best illustrated by example. Consider the application of event traces to enforce a history-based access control mechanism, as in [1, 37]. In this model, code is statically signed by its *owner*, an entity identified by their signature who lays claim to the code, and a local access control list $\mathcal{A}$ associates owners with their authorizations. As code executes, its owner's identity is recorded as an event in an execution trace, and a `demand` predicate ensures that the intersection of all authorizations of owners encountered up to the point of the check contains a specified privilege. Let `r` denote the specified privilege, and let `P` range over

owners. Thus, given an authorization trace $\mathtt{P}_1; \ldots; \mathtt{P}_n$, we require that $\mathtt{r} \in \mathcal{A}(\mathtt{P}_1) \cap \cdots \cap \mathcal{A}(\mathtt{P}_n)$ in order for the trace to satisfy demand(r).

In an Object Oriented setting such as Java, static code ownership is usually assigned on a class basis. In our encoding, class-based ownership is simulated by prepending an event at the beginning of every method m defined in the class, that will record the owner's identity on the execution trace when any such m is invoked. An accurate analysis therefore requires independence of trace effects of different method versions in the inheritance hierarchy, to accurately reflect the authorizations associated with these different versions.

Dynamic dispatch complicates the analysis in this respect. Imagine a class Writer that implements a safewrite method signed with a System authorization event, where safewrite takes a Formatter and a File as arguments, and requires that the FileWrite privilege be active before writing the formatter output to the file, via an access control check demand(FileWrite). Note especially that the specification of the check requires that FileWrite must be among the authorizations of the x.format method, since these will affect the flow of control and therefore appear in the safewrite event trace:

```
class Writer extends Object {
   void safewrite(Formatter x, File f){
      System;
      String s = x.format()
      demand(FileWrite);
      write(s, f);
   }
}
```

Thus, we can statically approximate the trace generated by the method safewrite as:

$$\mathtt{System}; \mathtt{H}; \mathtt{demand(FileWrite)}$$

where H represents the trace effect approximation of x.format(). The central issue is, what is H? Note that in a language with dynamic dispatch such as Java, the trace generated by x.format() could be generated by any version of format among the subclasses of Formatter, so it is unsound to imagine H as just the approximation of Formatters version.

In the FJ subtyping system, the type Formatter subsumes its subclass types via the subtyping relation. So as a first approximation, we can imagine extending subtyping to trace effects, meaning that H should subsume the effect of the format method in the Formatter class, *as well as the effects of every* format *method in* Formatter *subclasses*. This approach is taken in [21] as part of a related analysis for Java stack inspection, and a standard definition of effect subtyping [3, 27, 37] approximates the effect of x.format() as the nondeterministic choice of trace effects of the format method in every Formatter subclass. For example, suppose that there exist only two such classes, one which is owned by System, and the other which is owned by an Applet. This would be implemented by prepending the former's format method with a System event, and the latter's with an Applet event. For simplicity, we assume that these methods are otherwise event-free. In this case, we would have $\mathtt{H} = \mathtt{System}|\mathtt{Applet}$, where | is a choice constructor. But since it is natural to assume that Applets are not FileWrite authorized, verification of:

$$\mathtt{System}; (\mathtt{System}|\mathtt{Applet}); \mathtt{demand(FileWrite)}$$

will fail. This means that *any* invocation of `safewrite` would be statically rejected, even if invoked with a `System` formatter. Further, the scheme requires the entire `Formatter` class hierarchy be known in advance for static analysis, since any addition would require recomputation of its effects. This would disallow modularity, and scalability to large codebases where a given class may have many subclasses.

We address this problem by using polymorphism, rather than subtyping, to approximate the effects of method parameters; to wit, the effect `H` in question is represented by a universally quantified type variable. This is accomplished via the object type form $[\mathtt{T}\ \mathtt{C}]$, where `T` contains the inferred type and effects of a given object's methods, and `C` is the declared object class—so that the nominal type language of $\mathrm{FJ}_{\mathrm{trace}}$ is "superimposed" over the type language of FJ, as a conservative extension of the latter (as is discussed more extensively in Sect. 5). Let `StringT`, and `FileT` be the types of `String`, and `File` objects respectively, the details of which are unimportant to the example. Then, when typing the `safewrite` method in the `Writer` class, the $\mathrm{FJ}_{\mathrm{trace}}$ type system will assign an abstract effect `h` to its `x` parameter, as in the following type we abbreviate as `AbsFormatterT`:

$$\mathtt{AbsFormatterT} \triangleq [\mathtt{format} : () \xrightarrow{\mathtt{h}} \mathtt{StringT}\ \ \mathtt{Formatter}]$$

and `safewrite` may be assigned the type we abbreviate as `T`:

$$\mathtt{T} \triangleq (\mathtt{AbsFormatterT}, \mathtt{FileT}) \xrightarrow{\mathtt{System};\mathtt{h};\mathtt{demand(FileWrite)}} \mathtt{void}$$

and the typing $\mathtt{Writer} : \forall \mathtt{h}.[\mathtt{safewrite} : \mathtt{T}\ \ \mathtt{Writer}]$ may be assigned, where the abstract effect `h` of `x.format()` is quantified. At specific application points, `h` can then be instantiated with the accurate trace effect of the substituant of `x`. This example is extended and discussed in Sect. 5.3.1 following formal development of the type system.

## 2.2 Constraint subtyping and casting

To maintain decidability in the type system, we propose only first-order parametric polymorphism. This means that if `x` is a formal parameter of some method `m`, any method $\mathtt{x.m}'$ cannot be invoked within `m` in a polymorphic fashion. To obtain the flexibility necessary to statically allow application of abstracted methods to objects of multiple types, we propose a subtyping relation, similar to that discussed above, that can be used where parametric polymorphism cannot due to first-orderly restrictions.

A number of considerations motivate subtyping in our type and effect system, beyond the fact that it integrates neatly with FJ subtyping analysis. Firstly, while a top-level effect weakening rule, as in [37], is sufficient for a flexible type and effect analysis, a subtyping rule that incorporates weakening of latent effects on function types is more precise and complete, as observed in [3]. Also, we implement subtyping via a recursive constraint representation, which has been shown to allow precise typing of common object-oriented idioms, such as binary methods [10, 17].

A constraint type representation also supports a soft typing analysis of downcasts, which combines static and dynamic checks to ensure soundness [12]. For example, suppose some expression `e` has a type `T`, where `T` is constrained to be a supertype of both `Triangle` and `Polygon` objects, where `Triangle` is a subclass of `Polygon`, and where `R` represents the types of the fields and methods in `Triangles` and `S` represents those of `Polygons`:

$$[\mathtt{R}\ \mathtt{Triangle}] <: \mathtt{T} \qquad\qquad [\mathtt{S}\ \mathtt{Polygon}] <: \mathtt{T}$$

| | |
|---|---|
| $\mathtt{L} ::= \mathtt{class\ C\ extends\ C}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\}$ | *class definitions* |
| $\mathtt{K} ::= \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\mathtt{super}(\bar{\mathtt{f}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\}$ | *constructors* |
| $\mathtt{M} ::= \mathtt{C\ m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return\ e};\}$ | *methods* |
| $\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\bar{\mathtt{e}}) \mid \mathtt{new\ C}(\bar{\mathtt{e}}) \mid (\mathtt{C})\mathtt{e} \mid \mathtt{ev}[\mathtt{i}] \mid \mathtt{chk}[\mathtt{i}]$ | *expressions* |
| $\eta ::= \epsilon \mid \eta;\eta \mid \mathtt{ev}[\mathtt{i}] \mid \mathtt{chk}[\mathtt{i}]$ | *traces* |

**Fig. 1** $\text{FJ}_{\text{trace}}$ language syntax

Then, given the cast (Triangle)e, we first observe that an FJ dynamic cast check will ensure that any run-time scenario in which e evaluates to an object strictly in a superclass of Triangle will be stuck [24]. Guided by the intuition that constraints represent data flow paths, we further observe that any constraint representing flow of an object strictly in a superclass of Triangle to the program point represented by T can be ignored when analyzing the cast (Triangle)e, without compromising type safety, since this unsafe flow will be caught at run time by a dynamic cast check. In our type analysis, we implement this idea by positing a type $\mathtt{T}'$ such that:

$$(\mathtt{Triangle})\mathtt{e} : [\mathtt{T}'\ \mathtt{Triangle}]$$

with the condition that T be a *soft subtype* of $\mathtt{T}'$, written:

$$\mathtt{T} \lessdot [\mathtt{T}'\ \mathtt{Triangle}]$$

This means that if some object type $[\mathtt{U}\ \mathtt{C}]$ is a subtype of T, then $[\mathtt{U}\ \mathtt{C}] <: [\mathtt{T}'\ \mathtt{Triangle}]$ only if $\mathtt{C} <: \mathtt{Triangle}$; see Definition 3 for a formalization of the idea. This implies:

$$[\mathtt{R}\ \mathtt{Triangle}] <: [\mathtt{T}'\ \mathtt{Triangle}]$$

and *not* $[\mathtt{S}\ \mathtt{Polygon}] <: [\mathtt{T}'\ \mathtt{Triangle}]$. Note that requiring the latter would yield an inconsistent constraint set in any case. We believe that a constraint representation yields a distinctly precise type analysis; it is hard to see how the precision obtainable by selective pruning of the constraint graph used in our implementation of soft subtyping (Sect. 6) can be recreated with e.g. a unification-based approach. Subtyping and soft subtyping is further discussed in Sect. 5.2.1 following formal development of the interpretation of subtyping constraints.

## 3 The language $\text{FJ}_{\text{trace}}$

In this section we define the syntax and semantics of $\text{FJ}_{\text{trace}}$, by extending the syntax of FJ with primitives for specifying events and local checks, and the semantics of FJ with event traces as configuration components.

### 3.1 Syntax

The syntax of $\text{FJ}_{\text{trace}}$ is defined in Fig. 1. We let A, B, C, D range over class names, f range over field names, m range over method names, x, y range over both field and method names,

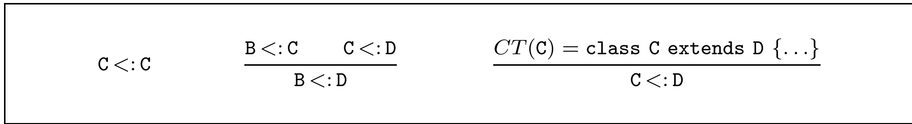

**Fig. 2** Nominal subtyping for FJ

and d, e range over expressions. *Values*, denoted v or u, are objects, i.e. expressions of the form new $\mathtt{C}(\mathtt{v}_1, \ldots, \mathtt{v}_n)$. A *class table* $CT$ is a mapping from class names C to definitions L, and a *program* is a pair $(CT, \mathtt{e})$; for brevity in the following, we assume a fixed class table $CT$. As in [24] we assume Object values that have no fields or methods; we let () denote the value new Object(). Since constructor definitions K are hard-coded by the language grammar, for brevity we do not explicitly include them in example class definitions in this paper.

The essential distinguishing features of $\mathrm{FJ}_{\mathrm{trace}}$ are *events* ev[i] and *local checks* chk[i]. Both events and checks are distinguished by labels i assigned automatically or by the programmer. In this presentation we will sometimes abbreviate events and checks by their labels, e.g. System and demand(FileWrite) abbreviate ev[System] and chk[demand(FileWrite)] as in Sect. 2. Events and checks encountered during execution are accrued in linear order in *traces* $\eta$, with execution blocking if unsuccessful checks are encountered. Events and checks are therefore side-effecting instructions; the value () is the direct evaluation result of checks and events, as specified in the next section.

In this presentation we leave the logic of checks abstract, specifying only that checks are predicates on traces, and write $\eta \vdash \mathtt{chk[i]}$ to denote that $\eta$ satisfies chk[i]. We could for example instantiate the language of checks with the linear mu-calculus, as in [37], but in this presentation we are mainly concerned with typing.

#### *3.1.1 Vector notations*

For brevity in numerous instances, we adopt the vector notations of [24]. We write $\bar{\mathtt{f}}$ to denote the sequence $\mathtt{f}_1, \ldots, \mathtt{f}_n$, similarly for $\bar{\mathtt{C}}$, $\bar{\mathtt{m}}$, $\bar{\mathtt{x}}$, $\bar{\mathtt{e}}$, etc., and we write $\bar{\mathtt{M}}$ as shorthand for $\mathtt{M}_1 \cdots \mathtt{M}_n$. We write the empty sequence as $\varnothing$, and we write $|\bar{\mathtt{x}}|$ to denote the length of $\bar{\mathtt{x}}$. If and only if m is one of the names in $\bar{\mathtt{m}}$, we write $\mathtt{m} \in \bar{\mathtt{m}}$, similarly for $\mathtt{f} \in \bar{\mathtt{f}}$. Given some $\bar{\mathtt{f}}$, we write $\mathtt{f}_i$ to denote the $i$th element of $\bar{\mathtt{f}}$. Vector notation is also used to abbreviate sequences of declarations; we let $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$ and $\bar{\mathtt{C}}\ \bar{\mathtt{f}};$ denote $\mathtt{C}_1\ \mathtt{f}_1, \ldots, \mathtt{C}_n\ \mathtt{f}_n$ and $\mathtt{C}_1\ \mathtt{f}_1; \ldots; \mathtt{C}_n\ \mathtt{f}_n;$ respectively, and we write $\mathtt{C}\ \mathtt{f} \in \bar{\mathtt{C}}\ \bar{\mathtt{f}}$ iff C f is one of the declarations in $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$. The notation $\mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};$ abbreviates the sequence of initializations $\mathtt{this}.\mathtt{f}_1 = \mathtt{f}_1; \ldots; \mathtt{this}.\mathtt{f}_n = \mathtt{f}_n;$. Sequences of names and declarations are assumed to contain no duplicate names.

### 3.2 Operational semantics

The operational semantics of $\mathrm{FJ}_{\mathrm{trace}}$ are defined in Fig. 3. The small-step reduction relation $\rightarrow$ is defined on closed *configurations*, which are pairs of traces and expressions $\eta, \mathtt{e}$. As in [24], we divide the operational rules into *computation* and *congruence* rules; the former (resp. latter) are those whose names are prefixed by R- (resp. RC-). Computation rules specify how redices reduce, while congruence rules specify how to evaluate within the context of a larger program. The semantics are defined in terms of a number of auxiliary functions and a nominal subtyping relation <: taken from [24] and recalled in Figs. 2 and 5; in particular,

$$\text{R-FIELD}\quad \frac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}}{\eta, (\mathtt{new\ C}(\bar{\mathtt{v}})).\mathtt{f}_i \to \eta, \mathtt{v}_i}$$

$$\text{R-INVK}\quad \frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}}{\eta, (\mathtt{new\ C}(\bar{\mathtt{v}})).\mathtt{m}(\bar{\mathtt{u}}) \to \eta, [\bar{\mathtt{u}}/\bar{\mathtt{x}}, \mathtt{new\ C}(\bar{\mathtt{v}})/\mathtt{this}]\mathtt{e}}$$

$$\text{R-CAST}\quad \frac{\mathtt{C} <: \mathtt{D}}{\eta, (\mathtt{D})(\mathtt{new\ C}(\bar{\mathtt{v}})) \to \eta, \mathtt{new\ C}(\bar{\mathtt{v}})}$$

$$\text{R-EVENT}\quad \eta, \mathtt{ev[i]} \to \eta; \mathtt{ev[i]}, ()$$

$$\text{R-CHECK}\quad \frac{\eta \vdash \mathtt{chk[i]}}{\eta, \mathtt{chk[i]} \to \eta; \mathtt{chk[i]}, ()}$$

$$\text{RC-FIELD}\quad \frac{\eta, \mathtt{e} \to \eta', \mathtt{e}'}{\eta, \mathtt{e.f} \to \eta', \mathtt{e}'.\mathtt{f}}$$

$$\text{RC-INVK-RECV}\quad \frac{\eta, \mathtt{e} \to \eta', \mathtt{e}'}{\eta, \mathtt{e.m}(\bar{\mathtt{e}}) \to \eta', \mathtt{e}'.\mathtt{m}(\bar{\mathtt{e}})}$$

$$\text{RC-INVK-ARG}\quad \frac{\eta, \mathtt{e}_i \to \eta', \mathtt{e}_i'}{\eta, \mathtt{v.m}(\bar{\mathtt{v}}, \mathtt{e}_i, \bar{\mathtt{e}}) \to \eta', \mathtt{v.m}(\bar{\mathtt{v}}, \mathtt{e}_i', \bar{\mathtt{e}})}$$

$$\text{RC-NEW-ARG}\quad \frac{\eta, \mathtt{e}_i \to \eta', \mathtt{e}_i'}{\eta, \mathtt{new\ C}(\bar{\mathtt{v}}, \mathtt{e}_i, \bar{\mathtt{e}}) \to \eta', \mathtt{new\ C}(\bar{\mathtt{v}}, \mathtt{e}_i', \bar{\mathtt{e}})}$$

$$\text{RC-CAST}\quad \frac{\eta, \mathtt{e} \to \eta', \mathtt{e}'}{\eta, (\mathtt{C})\mathtt{e} \to \eta', (\mathtt{C})\mathtt{e}'}$$

**Fig. 3** $\mathrm{FJ}_{\mathrm{trace}}$ operational semantics

*mtype*(m, C) and *mbody*(m, C) are functions that look up the type and body of a method m in a given class C, and implement the Java rules of inheritance. We let $\to^\star$ denote the reflexive, transitive closure of $\to$.

The operational semantics are largely the same as FJ, with the addition of run-time traces and the treatment of events and checks. The command new C($\bar{\mathtt{e}}$) constructs a new C object given field parameters $\bar{\mathtt{e}}$. Method selection is denoted e.f, and e.m($\bar{\mathtt{e}}$) is an invocation of method m. Casting is written (C)e. The rules that directly affect the trace include R-EVENT, which appends an event ev[i] encountered during execution to the end of the trace. The R-CHECK rule is defined similarly, except the check chk[i] is required to be satisfied by the current trace, otherwise computation becomes stuck. Each of the congruence rules propagates changes to the trace effected by the reduction of subterms.

## 4 Semantics of trace effects

The aim of our analysis is to statically guarantee the satisfaction of run-time checks in programs. To this end, our analysis infers an approximation of the trace that will be generated during program execution, by reconstructing the *trace effect* of programs. In essence, trace effects H conservatively approximate traces $\eta$ that may develop during execution, by representing a set of traces containing at least $\eta$. The grammar of trace effects is given in Fig. 6. A trace effect may be an event ev[i] or check chk[i], or a sequencing of trace effects $\mathtt{H}_1; \mathtt{H}_2$, a nondeterministic choice of trace effects $\mathtt{H}_1|\mathtt{H}_2$, or a $\mu$-bound trace effect $\mu\mathtt{h}.\mathtt{H}$ which finitely represents the set of traces that may be generated by a recursive function. Noting that the syntax of traces $\eta$ is the same as linear, variable-free trace effects, we abuse syntax and let $\eta$ also range over linear, variable-free trace effects.

We define a Labeled Transition System (LTS) interpretation of trace effects as sets of *abstract traces*, which are strings of events and checks that may be delimited by a $\downarrow$ symbol to denote termination. The interpretation is defined via strings denoted $\theta$, characterized by the following grammar, where $\epsilon$ denotes the empty string in the usual manner and the alphabet

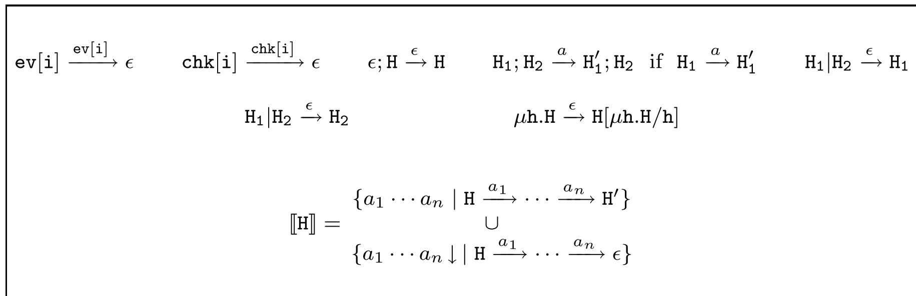

**Fig. 4** Interpretation of trace effects

consists of events and checks:

$$a ::= \texttt{ev[i]} \mid \texttt{chk[i]} \mid \epsilon$$
$$s ::= a \mid s\,s$$
$$\theta ::= s \mid s\downarrow$$

The interpretation of an effect $\texttt{H}$, denoted $[\![\texttt{H}]\!]$ and specified in Fig. 4, is then taken to be the prefix-closed, finite approximation of the trace sets that may be generated by $\texttt{H}$, when viewed as a program in a transition semantics defined by relations $\xrightarrow{a}$ on closed effects. Since programs may not terminate, their traces may be infinite, but this possibility is captured via finite approximation—all finite prefixes of an infinite trace are contained in the interpretation. Trace effect equivalence is defined via the interpretation, i.e. $\texttt{H}_1 = \texttt{H}_2$ iff $[\![\texttt{H}_1]\!] = [\![\texttt{H}_2]\!]$. This relation is in fact undecidable: trace effects are equivalent to BPA's (basic process algebras) [37], and their trace equivalence is known to be undecidable [11]. Note that prefix closure does not cause any loss of information in the interpretation, since the postpending of $\downarrow$ to terminating traces allows them to be distinguished from their prefixes. In particular, it is not necessarily true that $[\![\texttt{H}]\!] \subseteq [\![\texttt{H};\texttt{H}']\!]$ for arbitrary $\texttt{H}$ and $\texttt{H}'$.

Trace effects predict the history of both events and checks. In order to ensure that program checks will succeed, it is not sufficient for trace effects to approximate run time traces, but also success of these checks must be guaranteed. We base *validity* of trace effects on validity of checks that occur in traces in its interpretation. In particular, for any given check in a trace, that check must hold for its prefix:

**Definition 1** $\texttt{H}$ is *valid* iff for all $(a_1 \cdots a_n \texttt{chk[i]}) \in [\![\texttt{H}]\!]$ it is the case that $a_1; \ldots; a_n \vdash \texttt{chk[i]}$ holds.

This definition is logically sufficient to obtain our desired type safety result. It is not algorithmic, but trace effects are equivalent to basic process algebras (BPAs) as observed in [39], for which known model-checking techniques exist [11], allowing automated verification of effect validity [37]. In this paper, we focus on type and effect inference, rather than the effect verification component of our analysis. For details on the latter, the reader is referred to [37, 39].

### 4.1 Properties

Various properties of trace effect equivalence are enumerated as follows. The equivalences will be exploited for brevity and clarity in examples throughout the text, as well as for later proofs:

**Lemma 1** *We note the following properties of trace effect equivalence for closed* $\mathtt{H}, \mathtt{H}_1, \mathtt{H}_2$, *and* $\mathtt{H}_3$:

1. $\mathtt{H}|\mathtt{H} = \mathtt{H}$
2. $\epsilon; \mathtt{H} = \mathtt{H} = \mathtt{H}; \epsilon$
3. $\mu\mathtt{h}.\mathtt{H} = \mathtt{H}$
4. $\mathtt{H}_1|\mathtt{H}_2 = \mathtt{H}_2|\mathtt{H}_1$
5. $\mathtt{H}_1; (\mathtt{H}_2; \mathtt{H}_3) = (\mathtt{H}_1; \mathtt{H}_2); \mathtt{H}_3$
6. $\mathtt{H}_1|(\mathtt{H}_2|\mathtt{H}_3) = (\mathtt{H}_1|\mathtt{H}_2)|\mathtt{H}_3$
7. $\mathtt{H}_1; (\mathtt{H}_2|\mathtt{H}_3) = (\mathtt{H}_1; \mathtt{H}_2)|(\mathtt{H}_1; \mathtt{H}_3)$
8. $(\mathtt{H}_1|\mathtt{H}_2); \mathtt{H}_3 = (\mathtt{H}_1; \mathtt{H}_3)|(\mathtt{H}_2; \mathtt{H}_3)$
9. *Trace effect equivalence is preserved by all constructors.*

We also note some properties related to trace effect interpretation containment; these properties are important, since our type analyses will allow *weakening* of trace effects for flexibility. That is, if a trace effect $\mathtt{H}$ approximates the traces generated by a program, and $[\![\mathtt{H}]\!] \subseteq [\![\mathtt{H}']\!]$, then $\mathtt{H}'$ is also a sound approximation. Like equality, containment is known to be undecidable.

**Lemma 2** *Writing* $\mathtt{H} <: \mathtt{H}'$ *iff* $[\![\mathtt{H}]\!] \subseteq [\![\mathtt{H}']\!]$, *the following properties hold for arbitrary closed* $\mathtt{H}, \mathtt{H}_1$, *and* $\mathtt{H}_2$:

1. $\mathtt{H} <: \mathtt{H}_1$ *is undecidable.*
2. $\mathtt{H} <: \mathtt{H}|\mathtt{H}_1$.
3. *For closed* $\mu\mathtt{h}.\mathtt{H}_0$, *we have that* $\mathtt{H}_0[\mu\mathtt{h}.\mathtt{H}_0/\mathtt{h}] <: \mu\mathtt{h}.\mathtt{H}_0$.
4. *If* $\mathtt{H} <: \mathtt{H}_1$ *then* $\mathtt{H}|\mathtt{H}_2 <: \mathtt{H}_1|\mathtt{H}_2$ *and* $\mathtt{H}_2; \mathtt{H} <: \mathtt{H}_2; \mathtt{H}_1$ *and* $\mathtt{H}; \mathtt{H}_2 <: \mathtt{H}_1; \mathtt{H}_2$.
5. *If* $\mathtt{H}_1 <: \mathtt{H}_2$ *then validity of* $\mathtt{H}_2$ *implies validity of* $\mathtt{H}_1$.

## 5 Types for $\mathbf{FJ}_{\mathbf{trace}}$

Featherweight Java is equipped with a declarative, nominal type system; the type language is based on class names, which annotate function return and argument types, casts, and object creation points. Method and field types are not explicit in the FJ type of objects, which are just class names, but rather can be looked up given the object class name and its definition in the class table. The lookup functions *mtype* and *fields* are defined in Fig. 5. The FJ type system is algorithmically checkable, and enjoys a type safety result [24]. Our intent is to not to redo the type system of FJ, but to "superimpose" a type and effect analysis on it, thereby subsuming type safety for the FJ subset of $\mathrm{FJ}_{\mathrm{trace}}$. This superimposition should be conservative and transparent to the programmer, both for ease of use, and for backwards compatibility with Java. Thus, we reuse the declared, nominal type system of FJ, but add machinery to infer trace effects, for static verification of checks—although our type language will explicitly represent the field and method types of objects, rather than implicitly via lookup functions.

$$\mathit{fields}(\mathtt{Object}) = \varnothing \qquad \frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathit{fields}(\mathtt{D}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}, \bar{\mathtt{C}}\ \bar{\mathtt{f}}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{B\ m}(\bar{\mathtt{B}}\ \bar{\mathtt{x}})\{\mathtt{return\ e};\} \in \bar{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{B}} \to \mathtt{B}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{m} \notin \bar{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathit{mtype}(\mathtt{m}, \mathtt{D})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{B\ m}(\bar{\mathtt{B}}\ \bar{\mathtt{x}})\{\mathtt{return\ e};\} \in \bar{\mathtt{M}}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \qquad \mathtt{m} \notin \bar{\mathtt{M}}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \mathit{mbody}(\mathtt{m}, \mathtt{D})}$$

**Fig. 5** Auxiliary functions

| | |
|---|---|
| $\mathtt{H} ::= \epsilon \mid \mathtt{ev[i]} \mid \mathtt{chk[i]} \mid \mathtt{h} \mid \mathtt{H};\mathtt{H} \mid \mathtt{H}\vert\mathtt{H} \mid \mu\mathtt{h}.\mathtt{H}$ | *trace effects* |
| $\mathtt{T} ::= \mathtt{H} \mid \mathtt{X} \mid [\mathtt{T}\,\mathtt{C}] \mid (\bar{\mathtt{x}} : \bar{\mathtt{T}}) \mid \bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T}$ | *types* |
| $\mathtt{X} ::= \mathtt{h} \mid \mathtt{t}$ | *type variables* |
| $C ::= \mathtt{T} <: \mathtt{T} \mid \mathtt{T} \lessdot \mathtt{T} \mid C \wedge C \mid \mathbf{true}$ | *constraints* |

**Fig. 6** $\mathrm{FJ}_{\mathrm{trace}}$ type and constraint syntax

We define our type and effect analysis via subtyping constraints interpreted in a regular tree model. This representation promotes type reconstruction for common Object Oriented idioms such as object self-reference and binary methods [10, 17], since it possesses precisely the expressiveness of recursive types [32]. A constraint type representation also yields an elegant definition of the *soft subtyping* relation for static analysis of casts, discussed in Sect. 2 and formalized below, which supports a simple form of soft typing [12]. Furthermore, as has been observed frequently in previous related type and effect analyses [23, 27, 37], some flavor of subeffecting is necessary to conservatively extend underlying type structure, with a subtyping approach being particularly flexible [3]. A constraint representation is an effective implementation of subtyping, providing the expressiveness of intersection and union types [17]. While a recursive constraint representation is not the most human-readable type abstraction, our goal here is a transparent program analysis, and automatic extraction of trace effects for verification.

As discussed in Sect. 2, we also incorporate *effect polymorphism* [37], in a manner that allows flexibility and modularity of trace effect analysis in the presence of method override and dynamic dispatch.

## 5.1 Type and constraint language

The type and constraint grammar of $\mathrm{FJ}_{\mathrm{trace}}$ is given in Fig. 6. The type language includes method types $\bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T}$ where $\mathtt{H}$ is the latent effect of the method. Object types are denoted

$$\top^{\varnothing}, \bot^{\varnothing} : k \qquad\qquad \frac{\mathrm{fv}(\mathtt{H}) = \varnothing}{\mathtt{H}^{\varnothing} : \mathit{Eff}}$$

$$\frac{\begin{array}{c} \mathit{mtype}(\mathtt{m},\mathtt{C}) = \bar{\mathtt{D}} \to \mathtt{D} \qquad \varsigma = \mathit{Body}_{\bar{\mathtt{D}}}, \mathit{Eff}, \mathit{Body}_{\mathtt{D}} \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ B}\ \{\ldots\} \qquad \text{if } \mathit{mtype}(\mathtt{m},\mathtt{B}) = \bar{\mathtt{E}} \to \mathtt{E} \text{ then } \bar{\mathtt{E}} = \bar{\mathtt{D}} \text{ and } \mathtt{E} = \mathtt{D} \end{array}}{[\overline{\cdot\,\mathtt{D}}] \overset{\cdot}{\longrightarrow} [\cdot\,\mathtt{D}]^{\varsigma} : \mathit{Meth}_{\mathtt{m},\mathtt{C}}}$$

$$\frac{\bar{\mathtt{x}} \text{ distinct} \qquad |\varsigma| = |\bar{\mathtt{x}}| \qquad k \in \varsigma \Rightarrow k \in \{\mathit{Type}, \mathit{Meth}_{\mathtt{m},\mathtt{C}}\}}{(\bar{\mathtt{x}} : \cdot)^{\varsigma} : \mathit{Type}}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{f}} \qquad \varsigma = \mathit{Body}_{\bar{\mathtt{D}}}, \mathit{Meth}_{\bar{\mathtt{m}},\mathtt{C}}}{(\bar{\mathtt{f}} : [\overline{\cdot\,\mathtt{D}}]\ \bar{\mathtt{m}} : \cdot)^{\varsigma} : \mathit{Body}_{\mathtt{C}}} \qquad\qquad [\cdot\,\mathtt{C}]^{\mathit{Body}_{\mathtt{C}}} : \mathit{Type}$$

**Fig. 7** Regular tree ranked alphabet kinding rules

$[\mathtt{T}\,\mathtt{C}]$, where $\mathtt{C}$ is the class name of the object and $\mathtt{T}$ is either a type variable $\mathtt{t}$ or a vector of bindings $\bar{\mathtt{x}} : \bar{\mathtt{T}}$ for the field and method types of the object. The language of constraints is mostly standard, though in addition to subtyping constraints $\mathtt{T} <: \mathtt{T}$, we include weaker *soft subtyping* constraints $\mathtt{T} \lessdot \mathtt{T}$, to address downcasting in the type analysis.

The type language specified in the grammar is more liberal than what is actually allowed in type judgements. Formally, we will define well-formedness of types via interpretation in a regular tree model, defined and discussed in Sect. 5.2. The model is endowed with a primitive subtyping relation, which also supplies meaning for constraints via the interpretation. We will require that constraints and types $\mathtt{C}, \mathtt{T}$ assigned to expressions have a sensible (*well-kinded* and *realizable*, in our terminology) interpretation in the model. Informally, in any object type $[\mathtt{T}\,\mathtt{C}]$, if $\mathtt{T}$ is a vector of type bindings $\bar{\mathtt{x}} : \bar{\mathtt{T}}$, the set of bound names $\bar{\mathtt{x}}$ are restricted to be exactly the field and method names of $\mathtt{C}$, providing an "inlined" width constraint on the form of the type.

### *5.1.1 Vector notations*

For brevity we extend vector notation to the language of types and constraints. The type $\bar{\mathtt{T}}$ is a vector $\mathtt{T}_1, \ldots, \mathtt{T}_n$, with $\varnothing$ denoting the empty vector. We also write $[\overline{\mathtt{T}\,\mathtt{C}}]$ to denote a vector of class types $[\mathtt{T}_1\,\mathtt{C}_1], \ldots, [\mathtt{T}_n\mathtt{C}_n]$, while $\bar{\mathtt{x}} : \bar{\mathtt{T}}$ denotes a vector of bindings $\mathtt{x}_1 : \mathtt{T}_1 \cdots \mathtt{x}_n : \mathtt{T}_n$. We abbreviate constraints using vector notation, writing $\bar{\mathtt{S}} <: \bar{\mathtt{T}}$ for $\mathtt{S}_1 <: \mathtt{T}_1 \wedge \cdots \wedge \mathtt{S}_\mathtt{n} <: \mathtt{T}_\mathtt{n}$.

Looking ahead to the next section, vector notations are also used to abbreviate the kinding rules in Figs. 7 and 8. We write $\mathit{Body}_{\bar{\mathtt{C}}}$ for $\mathit{Body}_{\mathtt{C}_1}, \ldots, \mathit{Body}_{\mathtt{C}_n}$, and $\mathit{Meth}_{\bar{\mathtt{m}},\mathtt{C}}$ for $\mathit{Meth}_{\mathtt{m}_1,\mathtt{C}}, \ldots, \mathit{Meth}_{\mathtt{m}_n,\mathtt{C}}$. We write $\bar{\varphi} \preccurlyeq_{\mathrm{fin}} \bar{\varphi}'$ to denote $\varphi_1 \preccurlyeq_{\mathrm{fin}} \varphi_1', \ldots, \varphi_n \preccurlyeq_{\mathrm{fin}} \varphi_n'$.

## 5.2 Regular tree model and subtyping

To accommodate recursive constraints, we define subtyping in $\mathrm{FJ}_{\mathrm{trace}}$ via primitive subtyping in a regular tree model, using techniques adapted from [44]. In our model, function type nodes in regular trees are labeled with trace effects, and rather than being constructed from ranked alphabets as in [44] and elsewhere, our regular trees are constructed from kinded alphabets, imposing well-formedness of trees. Trace effect labelings of regular trees require

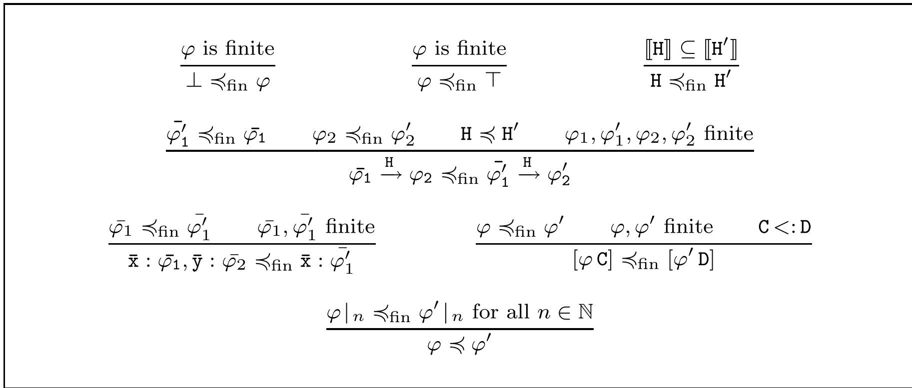

**Fig. 8** Primitive subtyping for regular trees

an extension of the primitive subtyping relation to trace effects, which is based on set containment of effect interpretations. Otherwise, the interpretation is essentially standard.

**Definition 2** (Regular Tree Model) Let the *tree constructor kinds* be defined as:

$$k ::= \mathit{Type} \mid \mathit{Eff} \mid \mathit{Meth}_{\mathtt{m},\mathtt{C}} \mid \mathit{Body}_C$$

and let *signatures* $\varsigma$ range over ordered sequences of kinds, where $\varnothing$ denotes the empty sequence and $\varsigma(n)$ denotes the 0-indexed $n$th kind in $\varsigma$. The alphabet $L$ of *tree constructors* is built from the following grammar:

$$c ::= \top \mid \bot \mid \mathtt{H} \mid (\bar{\mathtt{x}} : \cdot) \mid [\cdot\, \mathtt{C}] \mid [\cdot\, \bar{\mathtt{C}}] \xrightarrow{\cdot} [\cdot\, \mathtt{C}]$$

where each element of the alphabet is indexed by a signature, written $c^\varsigma$, and must be *well-kinded* according to the rules given in Fig. 7.

A *tree* $\varphi$ is a partial function from finite sequences (*paths*) $\pi$ of natural numbers $\mathbb{N}^\star$ to $L$ such that $dom(\varphi)$ is prefix-closed. Furthermore, for all $\pi n \in dom(\varphi)$, with $c^\varsigma = \varphi(\pi)$, it is the case that $\varphi(\pi n) : \varsigma(n)$. Trees of the form $(\bar{\mathtt{x}} : \bar{\varphi})$ are equated up to reordering of labels. The *subtree at* $\pi \in dom(\varphi)$ is the function $\lambda\pi'.\varphi(\pi\pi')$, while $|\pi|$ is the *level* of that subtree. A tree is *regular* iff the set of its subtrees is finite, and we define $\mathbb{T}$ as the set of regular trees over $L$.

A partial order over $\mathbb{T}$ is then defined via an approximate relation over finite $\varphi \in \mathbb{T}$. First, define a *level-n cut* $\varphi|_n$ for $\varphi \in \mathbb{T}$ as the finite tree obtained by replacing all subtrees at level $n$ of $\varphi$ with $\top$. Then, $\preccurlyeq_{\mathrm{fin}}$ is the partial order over finite $\varphi \in \mathbb{T}$ axiomatized in Fig. 8, and $\preccurlyeq$ is the partial order over $\mathbb{T}$ approximated by $\preccurlyeq_{\mathrm{fin}}$ axiomatized in Fig. 8.

The meaning of subtyping constraints is then defined via interpretation in the regular tree model. The principal novelties here are the extension of subtyping to trace effects, and the interpretation of the soft subtyping relation.

**Definition 3** (Interpretation of Constraints) *Interpretations* $\rho$ are total mappings from type variables $\mathtt{X}$ to $\mathbb{T}$. Interpretations are extended to types in Fig. 9, and also to effects by an

$$
\begin{aligned}
\rho(\mathtt{h}, hs) &= \rho(\mathtt{h}) && \mathtt{h} \notin hs\\
\rho(\mathtt{h}, hs) &= \mathtt{h} && \mathtt{h} \in hs\\
\rho(\mathtt{ev[i]}, hs) &= \mathtt{ev[i]}\\
\rho(\mathtt{chk[i]}, hs) &= \mathtt{chk[i]}\\
\rho(\epsilon, hs) &= \epsilon\\
\rho(\mathtt{H_1};\mathtt{H_2}, hs) &= \rho(\mathtt{H_1}, hs); \rho(\mathtt{H_2}, hs)\\
\rho(\mathtt{H_1}|\mathtt{H_2}, hs) &= \rho(\mathtt{H_1}, hs)|\rho(\mathtt{H_2}, hs)\\
\rho(\mu\mathtt{h}.\mathtt{H}, hs) &= \mu\mathtt{h}.\rho(\mathtt{H}, hs \cup \{h\})\\[1ex]
\rho([\mathtt{T}\,\mathtt{C}]) &= [\rho(\mathtt{T})\,\mathtt{C}]\\
\rho(\mathtt{x}:\mathtt{T}) &= \mathtt{x}:\rho(\mathtt{T})\\
\rho(\bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T}) &= \rho(\bar{\mathtt{T}}) \xrightarrow{\rho(\mathtt{H})} \rho(\mathtt{T})\\
\rho(\bar{\mathtt{T}}, \mathtt{T}) &= \rho(\bar{\mathtt{T}}), \rho(\mathtt{T})\\
\rho(\varnothing) &= \varnothing\\
\rho(\mathtt{H}) &= \rho(\mathtt{H}, \varnothing)
\end{aligned}
$$

**Fig. 9** Interpretations extended to types and effects

abuse of notation allowing parameterization by sets *hs* of effect variables, to prevent substitution of $\mu$-bound variables. The relation $\rho \vdash C$, pronounced $\rho$ *satisfies* or *solves* $C$, is axiomatized as follows:

$$
\rho \vdash \mathbf{true} \qquad \frac{\rho(\mathtt{S}) \preccurlyeq \rho(\mathtt{T})}{\rho \vdash \mathtt{S} <: \mathtt{T}} \qquad \frac{\rho \vdash C \qquad \rho \vdash D}{\rho \vdash C \wedge D}
$$

$$
\frac{\forall \mathtt{BR}.(\mathtt{B} <: \mathtt{D} \wedge \rho \vdash [\mathtt{R}\,\mathtt{B}] <: \mathtt{S}) \Rightarrow \rho \vdash [\mathtt{R}\,\mathtt{B}] <: [\mathtt{T}\,\mathtt{D}]}{\rho \vdash \mathtt{S} \lessdot [\mathtt{T}\,\mathtt{D}]}
$$

The relation $C \Vdash D$ holds iff $\rho \vdash C$ implies $\rho \vdash D$ for all interpretations $\rho$. Constraints $C$ and $D$ are *equivalent*, written $C = D$, iff $C \Vdash D$ and $D \Vdash C$.

We immediately observe that transitivity of primitive subtyping is reflected in the type and constraint representation.

**Lemma 3** (Transitivity of Entailment and Subtyping) *$C \Vdash D$ and $D \Vdash E$ implies $C \Vdash E$, and $C \Vdash \mathtt{R} <: \mathtt{S} \wedge \mathtt{S} <: \mathtt{T}$ implies $C \Vdash \mathtt{R} <: \mathtt{T}$.*

We also note the relevance of soft subtyping for the type analysis, that a soft subtyping relation entails an analogous subtyping relation just in case the latter could be sound.

**Lemma 4** *If $C \Vdash [\mathtt{T}\,\mathtt{C}] \lessdot [\mathtt{S}\,\mathtt{D}]$, then $C \Vdash [\mathtt{T}\,\mathtt{C}] <: [\mathtt{S}\,\mathtt{D}]$ iff $\mathtt{C} <: \mathtt{D}$.*

### *5.2.1 Discussion*

The kinding rules in the interpretation of types impose certain well-formedness qualities on types, in particular the class name component $\mathtt{C}$ of a type $[\mathtt{T}\,\mathtt{C}]$ fixes any fields and methods appearing in the type to be exactly those that are listed in the class definition. This is because

the underlying FJ type system allows any object to be treated as an object in its statically known class– thus, types for all known fields and methods for that class must be available, and no others. For example, imagining a class `Foo` with two fields `a` and `b` of `Object` type:

$$\texttt{class Foo extends Object} \{ \texttt{Object a;}\ \texttt{Object b;} \}$$

The type of `Foo` objects are labeled with the class name, and types for each of the fields:

$$[(\texttt{a} : [\texttt{Object}]; \texttt{b} : [\texttt{Object}])\ \texttt{Foo}]$$

While the type system will allow any `Foo` object to be viewed as an `Object` by subsumption, any object type term labeled `Foo` with anything other than both `a` and `b` fields has no interpretation due to the kinding rules. For example, the following type term has no interpretation:

$$[(\texttt{a} : [\texttt{Object}])\ \texttt{Foo}]$$

The kinding rule for method typings also deserves attention for its contribution to the independence of inheritance and effects in the analysis. Recalling that the type system of [24] requires that method overrides in a subclass have the same type signature as the overridden methods in their superclass, we observe that the kinding rule for constructors of kind $\mathit{Meth}_{\texttt{m},\texttt{C}}$ imposes this restriction, but only on the declared class name component of the type. The inferred effect component $\texttt{H}$, on the other hand, is not restricted to relate to superclass method effects in any way. This issue is revisited with examples in Sect. 5.3.1.

The interpretation of type constraints defines a system of object width and depth subtyping, with method subtyping predicated on subsumption of trace effects. This extension, defined in Fig. 8, is based on containment of trace effect interpretations, reflecting a type soundness requirement that if $\texttt{T}$ subsumes $\texttt{S}$, it must also subsume the trace effects of $\texttt{S}$. Since constraints are defined on the basis of interpretations, constraints on types with abstract components are meaningful; for example, we could assert:

$$\texttt{S} <: \texttt{T} \wedge \bar{\texttt{T}} <: \bar{\texttt{S}} \Vdash \bar{\texttt{S}} \xrightarrow{\texttt{h}} \texttt{S} <: \bar{\texttt{T}} \xrightarrow{\texttt{h}|(\texttt{ev[1]};\texttt{ev[2]})} \texttt{T}$$

since any interpretation of $\texttt{h}$ must be contained in the same interpretation $\texttt{h}|(\texttt{ev[1]};\ \texttt{ev[2]})$.

The soft subtyping relation is useful in application to downcasts, allowing constraints to be relaxed for downcasting. For example, assuming $\texttt{C} <: \texttt{D}$, we could meaningfully assert:

$$[\texttt{R}\,\texttt{C}] <: \texttt{T} \wedge [\texttt{S}\,\texttt{D}] <: \texttt{T} \wedge \texttt{T} \lessdot [\texttt{U}\,\texttt{C}] \Vdash [\texttt{R}\,\texttt{C}] <: [\texttt{U}\,\texttt{C}]$$

but not:

$$[\texttt{R}\,\texttt{C}] <: \texttt{T} \wedge [\texttt{S}\,\texttt{D}] <: \texttt{T} \wedge \texttt{T} \lessdot [\texttt{U}\,\texttt{C}] \Vdash [\texttt{S}\,\texttt{D}] <: [\texttt{U}\,\texttt{C}]$$

which formalizes the idea discussed in Sect. 2. This relaxation is allowable at casting points, since dynamic checks will "pick up the slack", preserving type safety as discussed in the next section.

This definition of subtyping may raise questions about the decidability of typing, since subtyping is predicated on trace effect containment, but equivalence of trace effects was shown to be undecidable in Sect. 4. However, while trace effect containment is undecidable in the general case, we show in Sect. 6 that constraints generated by type inference are in a normal form that is amenable to algorithmic solution.

T-Var
$$\Gamma, C, \epsilon \vdash \mathtt{x} : \Gamma(\mathtt{x})$$

T-Field
$$\frac{\Gamma, C, \mathtt{H} \vdash \mathtt{e} : [\mathtt{S}\,\mathtt{C}] \qquad C \Vdash \mathtt{S} <: (\mathtt{f} : [\mathtt{T}\,\mathtt{D}]) \qquad \mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C})}{\Gamma, C, \mathtt{H} \vdash \mathtt{e.f} : [\mathtt{T}\,\mathtt{D}]}$$

T-SeqNil
$$\frac{}{\Gamma, C, \epsilon \vdash \varnothing : \varnothing}$$

T-SeqCons
$$\frac{\Gamma, C, \mathtt{H}_1 \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}} \qquad \Gamma, C, \mathtt{H}_2 \vdash \mathtt{e} : \mathtt{T}}{\Gamma, C, \mathtt{H}_1; \mathtt{H}_2 \vdash \bar{\mathtt{e}}, \mathtt{e} : \bar{\mathtt{T}}, \mathtt{T}}$$

T-Invk
$$\frac{\Gamma, C, \mathtt{H}_1 \vdash \mathtt{e} : [\mathtt{T}\,\mathtt{C}] \qquad \Gamma, C, \mathtt{H}_2 \vdash \bar{\mathtt{e}} : \bar{\mathtt{S}} \qquad C \Vdash \mathtt{T} <: (\mathtt{m} : \bar{\mathtt{S}} \xrightarrow{\mathtt{H}_3} [\mathtt{S}\,\mathtt{D}]) \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{D}} \to \mathtt{D}}{\Gamma, C, \mathtt{H}_1; \mathtt{H}_2; \mathtt{H}_3 \vdash \mathtt{e.m}(\bar{\mathtt{e}}) : [\mathtt{S}\,\mathtt{D}]}$$

T-New
$$\frac{\begin{array}{c}\Gamma(\mathtt{C}) = \forall \bar{\mathtt{X}}[D].[\mathtt{T}\,\mathtt{C}] \\ C \Vdash D[\bar{\mathtt{S}}/\bar{\mathtt{X}}] \qquad \mathit{fieldsig}[\mathtt{T}\,\mathtt{C}] = \bar{\mathtt{T}} \qquad \Gamma, C, \mathtt{H} \vdash \bar{\mathtt{e}} : \bar{\mathtt{R}} \qquad C \Vdash \bar{\mathtt{R}} <: \bar{\mathtt{T}}[\bar{\mathtt{S}}/\bar{\mathtt{X}}]\end{array}}{\Gamma, C, \mathtt{H} \vdash \mathtt{new}\ \mathtt{C}(\bar{\mathtt{e}}) : [\mathtt{T}[\bar{\mathtt{S}}/\bar{\mathtt{X}}]\,\mathtt{C}]}$$

T-Event
$$\Gamma, C, \mathtt{ev}[\mathtt{i}] \vdash \mathtt{ev}[\mathtt{i}] : \mathtt{Unit}$$

T-Check
$$\Gamma, C, \mathtt{chk}[\mathtt{i}] \vdash \mathtt{chk}[\mathtt{i}] : \mathtt{Unit}$$

T-Cast
$$\frac{\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T} \qquad C \Vdash \mathtt{T} \lessdot [\mathtt{S}\,\mathtt{D}]}{\Gamma, C, \mathtt{H} \vdash (\mathtt{D})\mathtt{e} : [\mathtt{S}\,\mathtt{D}]}$$

T-Weaken
$$\frac{\Gamma, C, \mathtt{H}' \vdash \mathtt{e} : \mathtt{T} \qquad C \Vdash \mathtt{H}' <: \mathtt{H}}{\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}}$$

T-Meth
$$\frac{\Gamma; \bar{\mathtt{x}} : \bar{\mathtt{S}}, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{S} \qquad C \Vdash \mathtt{S} <: \mathtt{T} \wedge \bar{\mathtt{T}} <: \bar{\mathtt{S}} \qquad \Gamma(\mathtt{this}).\mathtt{m} = \bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}}{\Gamma, C \vdash \mathtt{m}, \mathtt{C} : \bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T}}$$

T-Class
$$\frac{\Gamma; \mathtt{C} : [\mathtt{T}\,\mathtt{C}]; \mathtt{this} : [\mathtt{T}\,\mathtt{C}], C \vdash \mathtt{m}_i, \mathtt{C} : \mathtt{T}_i \text{ for all } \mathtt{m}_i \in \mathit{meths}(\mathtt{C}) \qquad \bar{\mathtt{X}} \cap \mathrm{fv}(\Gamma) = \varnothing}{\Gamma \vdash \mathtt{C} : \forall \bar{\mathtt{X}}[C].[\mathtt{T}\,\mathtt{C}]}$$

**Fig. 10** $\mathrm{FJ}_{\mathrm{trace}}$ logical typing rules

### 5.3 Logical type judgements and properties

To define the logical type system, we introduce constrained type schemes $\forall \bar{\mathtt{X}}[C].T$ and type environments $\Gamma$ binding class names and variables to type schemes. If $\bar{\mathtt{X}} \cap \mathrm{fv}(\mathtt{T}) = \varnothing$ we abbreviate $\forall \bar{\mathtt{X}}[\mathbf{true}].T$ as $\mathtt{T}$. We also introduce three forms of type judgements: $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ for expressions, $\Gamma, C \vdash \mathtt{m}, \mathtt{C} : \bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T}$ for method $\mathtt{m}$ in class $\mathtt{C}$, and $\Gamma \vdash \mathtt{C} : \forall \bar{\mathtt{X}}[C].T$ for classes. Sensibility of types and constraints in judgements is imposed by realizability conditions, as follows.

**Definition 4** (Realizability of Types) We say that a constraint, type pair $C, \mathtt{T}$ or type scheme $\forall \bar{\mathtt{X}}[C].T$ is *realizable* iff there exists a solution $\rho$ of $C$ such that $\rho(\mathtt{T})$ is well-kinded, and we impose the following sanity conditions on judgements: for any $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ or $\Gamma, C \vdash \mathtt{m}, \mathtt{C} : \mathtt{T}$ we require that $C, \mathtt{T}$ is realizable, and every type scheme in $\Gamma$ must be realizable, in which case we say that $\Gamma$ is realizable.

$$\frac{\{\mathtt{m}_1, \ldots, \mathtt{m}_n\} = \{\mathtt{m} \mid \mathit{mbody}(\mathtt{m}, \mathtt{C}) \text{ is defined}\}}{\mathit{meths}(\mathtt{C}) = \mathtt{m}_1, \ldots, \mathtt{m}_n}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \mathtt{D}_1\ \mathtt{f}_1, \ldots, \mathtt{D}_n\ \mathtt{f}_n \qquad \mathtt{f}_1 : \mathtt{T}_1 \in \mathtt{T}, \ldots, \mathtt{f}_n : \mathtt{T}_n \in \mathtt{T}}{\mathit{fieldsig}[\mathtt{T}\,\mathtt{C}] = \mathtt{T}_1, \ldots, \mathtt{T}_n}$$

**Fig. 11** Auxiliary functions

Type derivation rules are given in Fig. 10, with some auxiliary functions defined in Fig. 11. For these rules and later examples, we posit a `Unit` type, which is the type of objects in the `Object` class, that possess no fields or methods; i.e.:

$$\mathtt{Unit} \triangleq [\mathtt{Object}]$$

In these and later rules we also write $[\bar{\mathtt{x}} : \bar{\mathtt{T}}\mathtt{C}].\mathtt{x}_i$ to denote $\mathtt{T}_i$. We note that ∀-intro and -elim are located at class definition and object construction points, respectively. Subeffecting of top level effects is allowable at any point in a derivation via T-WEAKEN. The soft subtyping relation is used for casts $(\mathtt{C})\mathtt{e}$; due to the interpretation of soft subtyping defined in the previous section, this rule will only track the flow of objects to the casting point that are in a subclass of `C`. Any other objects will cause a dynamic cast check exception, and can therefore be ignored statically without compromising type safety, as in [24]. Ignoring "junk" in this manner gives a more precise analysis, and allows typing of downcasts. We omit the distinction of "stupid casts" entailing "stupid warnings" as in [24], noting that we essentially follow their approach, and can easily adopt this distinction. We make the following definitions:

**Definition 5** The judgement $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ is *valid* iff it is derivable and there exists a solution $\rho$ of $C$ such that $\rho(\mathtt{H})$ is valid. An environment $\Gamma$ is *well-formed* iff $\Gamma \vdash \mathtt{C} : \forall X[C].T$ is derivable for all $(\mathtt{C} : \forall X[C].T) \in \Gamma$.

It is demonstrable that for closed, event- and check-free expressions `e`, there exists a derivable typing for `e` in $\mathrm{FJ}_{\mathrm{trace}}$ iff `e` is well-typed in the type system of [24], as we show in Sect. 5.4, Lemma 10. This means that our system absorbs properties of FJ, including type safety for the FJ subset of $\mathrm{FJ}_{\mathrm{trace}}$, implying that the only case we really need to consider, to extend type safety to $\mathrm{FJ}_{\mathrm{trace}}$, is progress for trace checks. That is, we must show that checks can be statically enforced by our type analysis. We establish this by a subject reduction argument, showing that trace effect approximations are preserved by computation. Note that the statement of subject reduction must apply to configurations; the proof, given in Sect. 5.4, follows by induction on derivations:

**Lemma 5** (Subject Reduction) *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : [\mathtt{T}\,\mathtt{C}]$ *is derivable for closed e and well formed* $\Gamma$, *and* $\eta, \mathtt{e} \to \eta', \mathtt{e}'$, *then* $\Gamma, C, \mathtt{H}' \vdash \mathtt{e}' : [\mathtt{T}'\,\mathtt{C}']$ *is derivable with* $C \Vdash [\mathtt{T}'\,\mathtt{C}'] <: [\mathtt{T}\,\mathtt{C}]$ *and* $C \Vdash \eta'; \mathtt{H}' <: \eta; \mathtt{H}$.

A corollary of this result formalizes the intuition that trace effects approximate event traces, insofar as any event trace generated during evaluation of an expression must be contained in the trace effect assigned to that expression by the type analysis:

**Corollary 1** *If $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : [\mathtt{T}\,\mathtt{C}]$ is derivable for closed e and well-formed $\Gamma$, and $\epsilon, \mathtt{e} \to \eta, \mathtt{e}'$, then $C \Vdash \eta <: \mathtt{H}$.*

To state our type safety result, we must define what we mean by "run-time checks". In short, these are checks in the hole of an evaluation context encountered during execution:

**Definition 6** *Evaluation contexts* are defined as follows:

$$\mathtt{E} ::= [\,] \mid \mathtt{E.f} \mid \mathtt{E.m}(\bar{\mathtt{e}}) \mid \mathtt{v.m}(\bar{\mathtt{v}}, \mathtt{E}, \bar{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}}, E, \bar{\mathtt{e}}) \mid (\mathtt{C})\mathtt{E}$$

We can then formally assert that run-time checks in well-typed programs are guaranteed to succeed:

**Lemma 6** (Static Enforcement of Checks) *Given closed e and well-formed $\Gamma$, if the judgement $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ is valid , and $\epsilon, \mathtt{e} \to^\star \eta, \mathtt{E}[\mathtt{chk}[\mathtt{x}]]$, then $\eta \vdash \mathtt{chk}[\mathtt{x}]$.*

Proofs are demonstrated in Sect. 5.4.

*5.3.1 Discussion*

In this section we discuss some examples that illustrate properties of the type system. We assume the definitions and types given in Sect. 2. We also assume the trivial extension of the language with a sequencing construct `e; e` and a lexically scoped name-to-value binding construct `C x = new C()`. To provide intuitions more easily, we present types and effects in a unified form. We posit the definition of a class `PasswdTmpl`, which is owned by `System`; in class `PasswdTmpl` the `format` method is overridden, defined to format a password file template string:

```
class PasswdTmpl extends Formatter {
   String format() { System; ...}
}
```

Assuming that the body of this version of `format` is effect free other than the initial `System` event, the type of `PasswdTmpl` can be given as follows, where `StringT` is the type of a `String` object, the details of which are unimportant to the example:

$$\mathtt{PTmplT} \triangleq [\mathtt{format} : () \xrightarrow{\mathtt{System}} \mathtt{StringT}\ \ \mathtt{PasswdTmpl}]$$

We also posit the definition of a class `Backdoor`, devised as an untrusted `Applet` extending `PasswdTmpl`, that formats a password file template containing a uname/passwd combination known to the attacker:

```
class Backdoor extends PasswdTmpl {
   String format() { Applet; ...}
}
```

Assuming that the body of this version of `format` is also effect free other than the initial `Applet` event, the type of `Backdoor` can be given as follows, which we abbreviate as `BdoorT`:

$$\mathtt{BdoorT} \triangleq [\mathtt{format} : () \xrightarrow{\mathtt{Applet}} \mathtt{StringT}\ \ \mathtt{Backdoor}]$$

Notice that the effects assigned to the `Backdoor` and `PasswdTmpl` versions of `format` are incomparable, despite their inheritance relation.

For the purposes of the example, we further imagine that there exists an open `PasswdFile` object in the current namespace. Now, since `Writer.safewrite` can be assigned a type which is polymorphic in the effects of its arguments, the code:

```
Writer w = new Writer();
PasswdTmpl p = new PasswdTmpl();
w.safewrite(p, PasswdFile);
```

could be assigned the effect:

```
System; System; demand(FileWrite);
```

which we assume is verifiable, whereas the following application of `safewrite` could be treated independently:

```
Writer w = new Writer();
Backdoor b = new Backdoor();
w.safewrite(b, PasswdFile);
```

and assigned the following effect, which we assume is not verifiable:

```
System; Applet; demand(FileWrite)
```

Since polymorphism is restricted to be first orderly, method parameters themselves cannot be polymorphic. This means that in code such as the following, the domain effects of functionally abstracted objects will be merged by subtyping:

```
class C extends Object {
   void m(Writer w, File f){
      Backdoor b = new Backdoor();
      PasswdTmpl p = new PasswdTmpl();
      w.safewrite(b, f);
      w.safewrite(p, f);
   }
}
```

This implies that the effects at each calling site of `w.safewrite` generated by the following code cannot be distinguished:

```
Writer w = new Writer();
C c = new C();
c.m(w, PasswdFile);
```

so that the following effect would be assigned:

```
System; (System|Applet); demand(FileWrite);

System; (System|Applet); demand(FileWrite)
```

Soft subtyping can be used at downcasts to ignore possibly unsound flow that will definitely be caught by dynamic cast checks. Suppose that type analysis predicts that some expression `e` may evaluate to either an `Object` object or a `PasswdTmpl` object; i.e. $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ with:

$$C \Vdash \mathtt{PTmplT} <: \mathtt{T} \wedge \mathtt{Unit} <: \mathtt{T}$$

where `Unit` and `PTmplT` are defined as above. By the T-CAST typing rule and properties of soft subtyping discussed above, we may assert:

$$\Gamma, C, \epsilon \vdash (\mathtt{PTmplT})\mathtt{e} : \mathtt{PTmplT}$$

meaning that the following is also a valid typing, regardless of the precise nature of `e`:

$$\Gamma, C, \mathtt{System} \vdash ((\mathtt{PTmplT})\mathtt{e}).\mathtt{format}() : \mathtt{StringT}$$

Note that if `e` turns out to be a `PTmplT` object at run-time, the invocation of `format` is safe, whereas if `e` turns out to be an `Object`, this will be caught by a dynamic cast check, so safety is guaranteed in any case.

### 5.4 Properties

In this section we demonstrate our main type safety result, based on a subject reduction argument. We also show that $\mathrm{FJ}_{\mathrm{trace}}$ typings conserve FJ typings, i.e. any program that is typable in the FJ type system is also typable in the $\mathrm{FJ}_{\mathrm{trace}}$ type system. In the course of proving these results, we demonstrate a standard suite of results (weakening, substitution, canonical forms, etc.) for the $\mathrm{FJ}_{\mathrm{trace}}$ type system.

We begin by demonstrating conservation of FJ typing in $\mathrm{FJ}_{\mathrm{trace}}$. FJ typing is nominal, and object types are represented "implicitly" via the *mtype* and *fields* functions. In $\mathrm{FJ}_{\mathrm{trace}}$, types are represented "explicitly" via type terms and constraints. Thus, we define a translation, or "expansion", from types of the former to the latter. The translation expands every class name `C` into a representation $[\mathtt{T}\,\mathtt{C}]$, where `T` is a listing of field and method types. Each method is assigned an $\epsilon$ effect, since FJ is effect free. Expansion of any class name `C` is only one level deep—that is, class names `D` in `C`'s field and method types are not recursively expanded, but assigned a special type variable that is constrained to be the lower bounds of the expansion of `D`. This ensures termination of expansion in the presence of recursive types. Since constraints can be recursive, a corollary of this definition is the representability of recursive class types in the expansion—i.e., the constraint is solvable.

**Definition 7** Assume given a fixed type variable $\mathtt{t}_{\mathtt{C}}$ for each $\mathtt{C} \in dom(CT)$. Then the expansion of method types, class names, and vectors of class names is defined as follows:

$$\frac{mtype(\mathtt{m}, \mathtt{C}) = (\mathtt{D}_1, \ldots, \mathtt{D}_n) \to \mathtt{D} \qquad [\overline{\mathtt{S}\,\mathtt{D}}] = [\mathtt{t}_{\mathtt{D}_1}\,\mathtt{D}_1], \ldots, [\mathtt{t}_{\mathtt{D}_n}\mathtt{D}_n]}{expand(\mathtt{m}, \mathtt{C}) = [\overline{\mathtt{S}\,\mathtt{D}}] \xrightarrow{\epsilon} [\mathtt{t}_{\mathtt{D}}\,\mathtt{D}]}$$

$$\frac{\begin{array}{c} fields(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{f}} \qquad \bar{\mathtt{D}} = \mathtt{D}_1, \ldots, \mathtt{D}_n \qquad [\overline{\mathtt{S}\mathtt{D}}] = [\mathtt{t}_{\mathtt{D}_1}\,\mathtt{D}_1], \ldots, [\mathtt{t}_{\mathtt{D}_n}\mathtt{D}_n] \\ meths(\mathtt{C}) = \mathtt{m}_1, \ldots, \mathtt{m}_m \qquad \bar{\mathtt{T}} = expand(\mathtt{m}_1, \mathtt{C}), \ldots, expand(\mathtt{m}_m, \mathtt{C}) \end{array}}{expand(\mathtt{C}) = [\bar{\mathtt{f}} : [\overline{\mathtt{S}\mathtt{D}}]\bar{\mathtt{m}} : \bar{\mathtt{T}}\mathtt{C}]}$$

$$expand(\bar{\mathtt{C}}) = expand(\mathtt{C}_1), \ldots, expand(\mathtt{C}_n)$$

We also construct a possibly recursive system of lower bounds on type variables $\mathtt{t}_{\mathtt{C}}$, as follows. Given $CT$ such that $dom(CT) = \mathtt{C}_1, \ldots, \mathtt{C}_n$. Then:

$$C_{CT} \triangleq expand(\mathtt{C}_1) <: [\mathtt{t}_{\mathtt{C}_1}\,\mathtt{C}_1] \wedge \cdots \wedge expand(\mathtt{C}_{\mathrm{n}}) <: [\mathtt{t}_{\mathtt{C}_n}\,\mathtt{C}_n]$$

**Lemma 7** *$C_{CT}$, expand*$(\mathtt{C})$ *is realizable for all* $\mathtt{C} \in dom(CT)$.

We observe that expansion preserves subtyping, as desired:

**Lemma 8** *If* $\mathtt{C} <: \mathtt{D}$ *then* $C_{CT} \Vdash expand(\mathtt{C}) <: expand(\mathtt{D})$.

Now, we show that the expansion allows a simulation of FJ field and method typing.

**Lemma 9** *Given* $expand(\mathtt{D}) = [\mathtt{T}\,\mathtt{D}]$. *Then the following properties hold*:

1. $C_{CT} \Vdash \mathtt{T} <: (\mathtt{m} : expand(\bar{\mathtt{B}}) \xrightarrow{\epsilon} expand(\mathtt{B}))$ *if* $mtype(\mathtt{m}, \mathtt{D}) = \bar{\mathtt{B}} \to \mathtt{B}$.
2. $C_{CT} \Vdash \mathtt{T} <: (\mathtt{f} : expand(\mathtt{B}))$ *if* $\mathtt{B}\ \mathtt{f} \in fields(\mathtt{D})$.

Now we can prove the conservativity result by induction on FJ type derivations. To properly frame the result, we need to extend expansion to environments, both to correctly state the induction, and to gather class type representations into the $\mathrm{FJ}_{\mathrm{trace}}$ type environment in the appropriate manner.

**Lemma 10** (Conservation of FJ Typing) *Define*:

$$\frac{expand(\Gamma) = \Gamma' \qquad expand(\mathtt{C}) = \mathtt{T}}{expand(\Gamma; x : \mathtt{C}) = \Gamma'; x : \mathtt{T}}$$

$$\frac{dom(()CT) = \mathtt{C}_1, \ldots, \mathtt{C}_n}{expand(\varnothing) = \mathtt{C}_1 : \forall\varnothing[C_{CT}].expand(\mathtt{C}_1); \ldots; \mathtt{C}_n : \forall\varnothing[C_{CT}].expand(\mathtt{C}_n)}$$

*If* $\Gamma \vdash \mathtt{e} : \mathtt{C}$ *is derivable*, *then so is* $expand(\Gamma), C_{CT}, \epsilon \vdash \mathtt{e} : expand(\mathtt{C})$.

*Proof* The result follows by induction on $\Gamma \vdash \mathtt{e} : \mathtt{C}$ and case analysis on the last step in the derivation. Here, we give only the T-INVK case, wherein $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\bar{\mathtt{e}})$, and by inversion of the rule we have:

$$\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad mtype(\mathtt{m}, \mathtt{C}_0) = \bar{\mathtt{D}} \to \mathtt{C} \qquad \Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{C}} \qquad \bar{\mathtt{C}} <: \bar{\mathtt{D}}$$

Hence by the induction hypothesis the judgements:

$$expand(\Gamma), C_{CT}, \epsilon \vdash \mathtt{e}_0 : expand(\mathtt{C}_0) \qquad expand(\Gamma), C_{CT}, \epsilon \vdash \bar{\mathtt{e}} : expand(\bar{\mathtt{C}})$$

are derivable. Let $expand(\mathtt{C}_0) = [\mathtt{T}\,\mathtt{C}_0]$. Then by Lemmas 8 and 9, the following are valid:

$$C_{CT} \Vdash \mathtt{T} <: (\mathtt{m} : expand(\bar{\mathtt{D}}) \xrightarrow{\epsilon} expand(\mathtt{C})) \qquad \mathtt{C}_{\mathtt{CT}} \Vdash expand(\bar{\mathtt{C}}) <: expand(\bar{\mathtt{D}})$$

But then by Definition 3 and Lemma 3:

$$C_{CT} \Vdash \mathtt{T} <: (\mathtt{m} : expand(\bar{\mathtt{C}}) \xrightarrow{\epsilon} expand(\mathtt{C}))$$

so the judgement:

$$expand(\Gamma), C_{CT}, \epsilon; \epsilon; \epsilon \vdash \mathtt{e} : expand(\mathtt{C})$$

is derivable by T-INVK in $\mathrm{FJ}_{\mathrm{trace}}$, hence this case follows by T-WEAKEN. □

Next, we turn to subject reduction and type soundness. We start by showing that valid typing judgements can always be specialized, either by weakening the top-level constraint in the judgement, instantiating type variables, or by adding extraneous bindings to the type environment. Each of these results follows by a straightforward induction on type derivations. The instantiation lemma also follows because instantiation preserves subtyping, as we show in Lemma 12, a result that follows immediately by Definition 3.

**Lemma 11** (Constraint Weakening) *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *and* $D \Vdash C$ *for solvable* $D$ *then* $\Gamma, D, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$.

**Lemma 12** *If* $C \Vdash \mathtt{S} <: \mathtt{T}$ *then* $C[\bar{\mathtt{T}}/\bar{\mathtt{X}}] \Vdash \mathtt{S}[\bar{\mathtt{T}}/\bar{\mathtt{X}}] <: \mathtt{T}[\bar{\mathtt{T}}/\bar{\mathtt{X}}]$.

**Lemma 13** (Instantiation) *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is derivable, then so is the judgement* $\Gamma[\bar{\mathtt{T}}/\bar{\mathtt{X}}], C[\bar{\mathtt{T}}/\bar{\mathtt{X}}], \mathtt{H}[\bar{\mathtt{T}}/\bar{\mathtt{X}}] \vdash \mathtt{e} : \mathtt{T}[\bar{\mathtt{T}}/\bar{\mathtt{X}}]$.

**Lemma 14** (Environment Weakening) *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is derivable and* $\Gamma(\mathtt{C}) = \Gamma'(\mathtt{C})$ *for all* $\mathtt{C} \in dom(()CT)$ *and* $\Gamma(\mathtt{x}) = \Gamma'(\mathtt{x})$ *for all* $\mathtt{x} \in fv(\mathtt{e})$, *then* $\Gamma', C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is derivable.*

Since the T-WEAKEN rule is not syntax directed, it introduces non-determinism in derivations. The following result allows us to treat specified normal forms of derivations in proofs, without loss of generality.

**Lemma 15** (Normalization) *If the judgement* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is derivable, then the judgement follows by one instance of* T-WEAKEN *preceded by an instance of a syntax-directed rule corresponding to the form of* $\mathtt{e}$.

We also show that substitution preserves typings, to underpin the method invocation case of subject reduction. After that, we give a canonical forms lemma, specifying the form of values inhabiting particular types. We also observe that values can always be assigned top-level $\epsilon$ effects– since values do not reduce, they can have no trace history.

**Lemma 16** (Substitution) *If* $\Gamma; \mathtt{x} : \mathtt{S}, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Gamma, C, \epsilon \vdash \mathtt{v} : \mathtt{S}'$ *with* $C \Vdash \mathtt{S}' <: \mathtt{S}$, *then* $\Gamma, C, \mathtt{H}' \vdash [\mathtt{v}/\mathtt{x}]\mathtt{e} : \mathtt{T}'$ *for some* $\mathtt{T}'$ *and* $\mathtt{H}'$ *with* $C \Vdash \mathtt{T}' <: \mathtt{T} \wedge \mathtt{H}' <: \mathtt{H}$.

**Lemma 17** (Canonical Forms) *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{v} : [\mathtt{T}\,\mathtt{C}]$ *is derivable for closed* $\mathtt{v}$ *and well formed* $\Gamma$, *then* $\mathtt{v} = \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})$ *for some* $\bar{\mathtt{v}}$.

**Lemma 18** *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{v} : [\mathtt{T}\,\mathtt{C}]$ *is derivable, then* $C \Vdash \epsilon <: \mathtt{H}$ *and* $\Gamma, C, \epsilon \vdash \mathtt{v} : [\mathtt{T}\,\mathtt{C}]$ *is also derivable.*

The next two utility results allow the typings of method bodies to be retrieved by inversion of method invocation typings. Like Lemma 16, they will be instrumental for the method invocation case of subject reduction.

**Lemma 19** *Given* $\Gamma(\mathtt{C}) = \forall \bar{\mathtt{X}}[C].[\mathtt{T}\,\mathtt{C}]$ *and* $mbody(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}$ *for well formed* $\Gamma$. *Then there exists* $(m : \bar{\mathtt{S}} \xrightarrow{\mathtt{H}} \mathtt{S}) \in \mathtt{T}$, $\bar{\mathtt{R}}$ *and* $\mathtt{R}$ *such that the following are valid*:

$$C \Vdash \mathtt{R} <: \mathtt{S} \wedge \bar{\mathtt{S}} <: \bar{\mathtt{R}} \qquad \Gamma; \mathtt{this} : [\mathtt{T}\,\mathtt{C}]; \bar{\mathtt{x}} : \bar{\mathtt{R}}, \mathtt{C}, \mathtt{H} \vdash \mathtt{e} : \mathtt{R}$$

*Proof* By inversion of the T-CLASS and T-METH rules, and Lemma 14. □

**Lemma 20** *Given well formed $\Gamma$ and*:

$$\Gamma, C, \mathtt{H}_0 \vdash \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}}) : [\mathtt{TC}] \qquad mbody(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e} \qquad C \Vdash \mathtt{T} <: (\mathtt{m} : \bar{\mathtt{S}} \xrightarrow{\mathtt{H}_1} \mathtt{S})$$

*Then*:

$$\Gamma; \mathtt{this} : [\mathtt{T\,C}]; \bar{\mathtt{x}} : \bar{\mathtt{R}}, C, \mathtt{H}_2 \vdash \mathtt{e} : \mathtt{R}$$

*is derivable*, *where*:

$$C \Vdash \mathtt{R} <: \mathtt{S} \wedge \bar{\mathtt{S}} <: \bar{\mathtt{R}} \wedge \mathtt{H}_2 <: \mathtt{H}_1$$

*Proof* By Lemma 15 and inversion of the T-NEW rule, we have that $[\mathtt{T\,C}] = [\mathtt{T}_0\,\mathtt{C}][\bar{\mathtt{S}}'/\bar{\mathtt{X}}]$ where $\Gamma(\mathtt{C}) = \forall \bar{\mathtt{X}}[D].[\mathtt{T}_0\,\mathtt{C}]$ and $C \Vdash D[\bar{\mathtt{S}}'/\bar{\mathtt{X}}]$. Then by Lemma 19, there exists $(\mathtt{m} : \bar{\mathtt{R}}_0 \xrightarrow{\mathtt{H}} \mathtt{R}_0) \in \mathtt{T}_0$, $\bar{\mathtt{S}}_0$ and $\mathtt{S}_0$ such that:

$$C \Vdash \mathtt{S}_0 <: \mathtt{R}_0 \wedge \bar{\mathtt{R}}_0 <: \bar{\mathtt{S}}_0 \qquad \Gamma; \mathtt{this} : [\mathtt{T}_0\mathtt{C}]; \bar{\mathtt{x}} : \bar{\mathtt{S}}_0, D, \mathtt{H} \vdash \mathtt{e} : \mathtt{S}_0$$

are both valid. Letting:

$$\bar{\mathtt{R}} \triangleq \bar{\mathtt{S}}_0[\bar{\mathtt{S}}'/\bar{\mathtt{X}}] \qquad \mathtt{R} \triangleq \mathtt{S}_0[\bar{\mathtt{S}}'/\bar{\mathtt{X}}] \qquad \mathtt{H}_2 \triangleq \mathtt{H}[\bar{\mathtt{S}}'/\bar{\mathtt{X}}]$$

Then by Lemma 12:

$$C \Vdash \mathtt{R} <: \mathtt{R}_0[\bar{\mathtt{S}}'/\bar{\mathtt{X}}] \wedge \bar{\mathtt{R}}_0[\bar{\mathtt{S}}'/\bar{\mathtt{X}}] <: \bar{\mathtt{R}}$$

so by Definition 3 and Lemma 3:

$$C \Vdash \mathtt{R} <: \mathtt{S} \wedge \bar{\mathtt{S}} <: \bar{\mathtt{R}} \wedge \mathtt{H}_2 <: \mathtt{H}_1$$

and by Lemmas 11 and 13:

$$\Gamma; \mathtt{this} : [\mathtt{T\,C}]; \bar{\mathtt{x}} : \bar{\mathtt{R}}, C, \mathtt{H}_2 \vdash \mathtt{e} : \mathtt{R}$$

which was to be demonstrated. □

We can now prove subject reduction. Note that the statement of the proof allows for an increase in precision of type and effect in reduced terms.

**Lemma 5** (Subject Reduction) *If $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : [\mathtt{T\,C}]$ is derivable for closed* e *and well formed $\Gamma$, and $\eta, \mathtt{e} \to \eta', \mathtt{e}'$, then $\Gamma, C, \mathtt{H}' \vdash \mathtt{e}' : [\mathtt{T}'\,\mathtt{C}']$ is derivable with $C \Vdash [\mathtt{T}'\,\mathtt{C}'] <: [\mathtt{T\,C}]$ and $C \Vdash \eta'; \mathtt{H}' <: \eta; \mathtt{H}$.*

*Proof* By induction on $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : [\mathtt{T\,C}]$, case analysis on the last step in the derivation, and associated subcase analysis defined by possible last steps in the derivation of and $\eta, \mathtt{e} \to \eta', \mathtt{e}'$, given the form of e as required by the case. We consider the most interesting and crucial cases, treating method invocation, casting and soft subtyping, events, and effect weakening.

Case T-INVK. In this case, by inversion of the rule we have:

$$\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\bar{\mathtt{e}}) \qquad \mathtt{H} = \mathtt{H}_1;\mathtt{H}_2;\mathtt{H}_3 \qquad \Gamma, \mathcal{C}, \mathtt{H}_1 \vdash \mathtt{e}_0 : [\mathtt{S}\,\mathtt{D}] \qquad \Gamma, \mathcal{C}, \mathtt{H}_2 \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}$$

$$\mathcal{C} \Vdash \mathtt{S} <: (\mathtt{m} : \bar{\mathtt{T}} \xrightarrow{\mathtt{H}_3} [\mathtt{T}\mathtt{C}])$$

A subcase analysis for the form of $\mathtt{e}$ in this case comprises R-INVK, RC-INVK-RECV, and RC-INVK-ARG, as follows.

Subcase R-INVK. In this subcase by rule R-INVK and Lemma 17 we have:

$$\mathtt{e}_0 = \mathtt{new}\;\mathtt{D}(\bar{\mathtt{v}}) \qquad \bar{\mathtt{e}} = \bar{\mathtt{u}} \qquad \mathtt{e}' = [\bar{\mathtt{u}}/\bar{\mathtt{x}}, \mathtt{new}\;\mathtt{D}(\bar{\mathtt{v}})/\mathtt{this}]\mathtt{e}_1 \qquad \mathit{mbody}(\mathtt{m}, \mathtt{D}) = \bar{\mathtt{x}}.\mathtt{e}_1$$

$$\eta' = \eta$$

Thus by Lemma 20:

$$\Gamma; \mathtt{this} : [\mathtt{S}\,\mathtt{D}]; \bar{\mathtt{x}} : \bar{\mathtt{S}}, \mathcal{C}, \mathtt{H}_4 \vdash \mathtt{e}_1 : [\mathtt{T}_1\,\mathtt{C}_1]$$

is derivable, where:

$$\mathcal{C} \Vdash [\mathtt{T}_1\,\mathtt{C}_1] <: [\mathtt{T}\,\mathtt{C}] \wedge \bar{\mathtt{T}} <: \bar{\mathtt{S}} \wedge \mathtt{H}_4 <: \mathtt{H}_3$$

Therefore by Lemma 18 and Lemma 16 the judgement:

$$\Gamma, \mathcal{C}, \mathtt{H}'_4 \vdash [\bar{\mathtt{u}}/\bar{\mathtt{x}}, \mathtt{new}\;\mathtt{D}(\bar{\mathtt{v}})/\mathtt{this}]\mathtt{e}_1 : [\mathtt{T}'_1\mathtt{C}'_1]$$

is derivable, where $\mathcal{C} \Vdash [\mathtt{T}'_1\,\mathtt{C}'_1] <: [\mathtt{T}_1\,\mathtt{C}_1]$ and $\mathcal{C} \Vdash \mathtt{H}'_4 <: \mathtt{H}_4$. Thus $\mathcal{C} \Vdash [\mathtt{T}'_1\,\mathtt{C}'_1] <: [\mathtt{T}\,\mathtt{C}]$ and $\mathcal{C} \Vdash \mathtt{H}'_4 <: \mathtt{H}_3$ by Lemma 3, so it only remains to be shown that $\mathcal{C} \Vdash \eta; \mathtt{H}'_4 <: \eta; \mathtt{H}_3$. But $\mathcal{C} \Vdash \epsilon <: \mathtt{H}_1$ and $\mathcal{C} \Vdash \epsilon <: \mathtt{H}_2$ by Lemma 18, hence $\mathcal{C} \Vdash \mathtt{H}_3 <: \mathtt{H}_1;\mathtt{H}_2;\mathtt{H}_3$ by Lemma 2, so the result follows by Lemmas 2 and 3.

Subcase RC-INVK-RECV. In this subcase $\mathtt{e}' = \mathtt{e}_1.\mathtt{m}(\bar{\mathtt{e}})$ for some $\mathtt{e}_1$ such that $\eta, \mathtt{e}_1 \to \eta', \mathtt{e}_1$. But then by the induction hypothesis we have $\Gamma, \mathcal{C}, \mathtt{H}'_1 \vdash \mathtt{e}_1 : [\mathtt{S}'\,\mathtt{D}']$ with $\mathcal{C} \Vdash \eta'; \mathtt{H}'_1 <: \eta; \mathtt{H}_1 \wedge [\mathtt{S}'\,\mathtt{D}'] <: [\mathtt{S}\,\mathtt{D}]$. These facts, Definition 3, and Lemma 3 imply $\mathcal{C} \Vdash \mathtt{S}' <: (\mathtt{m} : [\bar{\mathtt{R}}\mathtt{B}] \xrightarrow{\mathtt{H}_3} [\mathtt{T}\,\mathtt{C}])$, hence $\Gamma, \mathcal{C}, \mathtt{H}'_1;\mathtt{H}_2;\mathtt{H}_3 \vdash \mathtt{e}_1.\mathtt{m}(\bar{\mathtt{e}}) : [\mathtt{T}\mathtt{C}]$ is derivable by an instance of T-INVK, with $\mathcal{C} \Vdash \eta'; \mathtt{H}'_1;\mathtt{H}_2;\mathtt{H}_3 <: \eta; \mathtt{H}_1;\mathtt{H}_2;\mathtt{H}_3$ by Lemma 2, so this case holds. Subcase RC-INVK-ARG follows in a similar manner.

Case T-CAST. In this case, by inversion of the rule we have:

$$\mathtt{e} = (\mathtt{C})\mathtt{e}_0 \qquad \Gamma, \mathcal{C}, \mathtt{H} \vdash \mathtt{e}_0 : [\mathtt{S}\,\mathtt{D}] \qquad \mathcal{C} \Vdash [\mathtt{S}\,\mathtt{D}] \lessdot [\mathtt{T}\,\mathtt{C}]$$

A subcase analysis for the form of $\mathtt{e}$ in this case comprises R-CAST and RC-CAST, as follows.

Subcase R-CAST. In this subcase we have:

$$\mathtt{e}_0 = \mathtt{new}\;\mathtt{D}(\bar{\mathtt{v}}) \qquad \mathtt{D} <: \mathtt{C} \qquad \mathtt{e}' = \mathtt{e}_0 \qquad \eta = \eta'$$

by inversion of R-CAST and Lemma 17. But then $\mathcal{C} \Vdash [\mathtt{S}\,\mathtt{D}] <: [\mathtt{T}\,\mathtt{C}]$ by Lemma 4; the result follows. Subcase RC-CAST follows in a manner similar to the RC-INVK-RECV subcase of the T-INVK case above.

Case T-EVENT. In this case by inversion of the rule we have:

$$\mathtt{e} = \mathtt{ev}[\mathtt{i}] \qquad \mathtt{H} = \mathtt{ev}[\mathtt{i}] \qquad [\mathtt{T}\,\mathtt{C}] = \mathtt{Unit}$$

The only reduction rule that applies to the form of $\mathtt{e}$ in this case is R-EVENT, inversion of which obtains:

$$\mathtt{e}' = () \qquad\qquad \eta' = \eta; \mathtt{ev[i]}$$

But $\Gamma, C, \epsilon \vdash () : \mathtt{Unit}$ by assumption and Lemma 18, so this case holds.

Case T-WEAKEN. In this case we have $\Gamma, C, \mathtt{H}' \vdash \mathtt{e} : [\mathtt{T}\,\mathtt{C}]$ with $C \Vdash \mathtt{H}' <: \mathtt{H}$ by inversion of rule T-WEAKEN. But then $\Gamma, C, \mathtt{H}'' \vdash \mathtt{e}' : [\mathtt{T}'\,\mathtt{C}']$ with $C \Vdash \eta'; \mathtt{H}'' <: \eta; \mathtt{H}' \wedge [\mathtt{T}'\,\mathtt{C}'] <: [\mathtt{T}\,\mathtt{C}]$ by the induction hypothesis, and $C \Vdash \eta'; \mathtt{H}'' <: \eta; \mathtt{H}$ Lemmas 2 and 3, so this case holds. □

We then prove one auxiliary lemma followed by our main type safety result, demonstrating that run-time checks in well-typed programs are guaranteed to succeed:

**Lemma 21** *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{E[chk[x]]} : \mathtt{T}$ *then* $C \Vdash \mathtt{chk[x]}; \mathtt{H}' <: \mathtt{H}$.

**Lemma 6** (Static Enforcement of Checks) *Given closed e and well-formed* $\Gamma$*, if the judgement* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is valid, and* $\epsilon, \mathtt{e} \to^\star \eta, \mathtt{E[chk[x]]}$*, then* $\eta \vdash \mathtt{chk[x]}$.

*Proof* By Lemmas 5 and 21, $\Gamma, C, \mathtt{H}' \vdash \mathtt{E[chk[x]]} : \mathtt{T}$ is derivable with $C \Vdash \mathtt{chk[x]}; \mathtt{H}'' <: \mathtt{H}'$ and $C \Vdash \eta; \mathtt{H}' <: \mathtt{H}$. Now, by assumption there exists a solution $\rho$ of $C$ such that $\rho(\mathtt{H})$ is valid; but since $[\![\rho(\mathtt{chk[x]}; \mathtt{H}'')]\!] \subseteq [\![\rho(\mathtt{H}')]\!]$ and $[\![\rho(\eta; \mathtt{H}')]\!] \subseteq [\![\rho(\mathtt{H})]\!]$ by previous facts and Definition 3, therefore $[\![\rho(\eta; \mathtt{chk[x]}; \mathtt{H}'')]\!] \subseteq [\![\rho(\mathtt{H})]\!]$. By Lemma 2 it follows that the effect $\rho(\eta; \mathtt{chk[x]}; \mathtt{H}')$ is valid, thus $\eta \vdash \mathtt{chk[x]}$ by Definition 1. □

## 6 Type inference for $\mathbf{FJ_{trace}}$

We now develop an implementation of the $\mathrm{FJ_{trace}}$ type system. This includes a type inference algorithm for reconstructing type judgements. In addition, it is necessary to check satisfiability of constraints, to ensure that these judgements are coherent. For this purpose we adapt standard *closure* techniques for subtyping constraints [31], extended to accommodate effect constraints. Closure distills inferred constraints into their basic component elements, upon which a simple structural consistency check can be used to ensure satisfiability. Inference and closure serve as preliminary phases for statically verifying trace-based program assertions, which can finally be accomplished by model checking trace effects, as in [37]. Model checking is applicable, since we endow trace effects with an LTS semantics (Sect. 4), for which a variety of model checking techniques exist [11]. However, standard techniques expect term, rather than constraint, representation of LTSs. Therefore, it is also necessary to define a means of extracting a unified trace effect representation from inferred typing judgements. For this purpose we define an algorithm, called *hextract*, to obtain a unified representation of trace effects from the inferred constraint representation. The composition of type inference, closure, and hextraction obtains an algorithmic technique for enforcing type safety, as observed in Theorem 1.

### 6.1 The Type Language

The type and constraint language used by inference is the same as that presented in Sect. 5, albeit specialized to integrate neatly with closure. There are two forms of object types used by inference, a form $[\mathtt{t}\,\mathtt{C}]$ that we call *abstract*, and an *hexpanded* form defined in Fig. 12. This form is similar to the expanded form of object types defined in Sect. 5.4, where field

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ B}\ \{\ldots\} \qquad \begin{array}{c} mtype(\mathtt{m},\mathtt{C}) = \bar{\mathtt{D}} \to \mathtt{D} \\ \text{if } mtype(\mathtt{m},\mathtt{B}) = \bar{\mathtt{E}} \to \mathtt{E} \text{ then } \bar{\mathtt{E}} = \bar{\mathtt{D}} \text{ and } \mathtt{E} = \mathtt{D} \end{array}}{hexpand(\mathtt{m},\mathtt{C}) = [\overline{\mathtt{t}\,\mathtt{D}}] \xrightarrow{\mathtt{h}} [\mathtt{t}\,\mathtt{D}]}$$

$$\frac{fields(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{f}} \qquad meths(\mathtt{C}) = \mathtt{m}_1,\ldots,\mathtt{m}_n \qquad \bar{\mathtt{T}} = hexpand(\mathtt{m}_1,\mathtt{C}),\ldots,hexpand(\mathtt{m}_n,\mathtt{C})}{hexpand(\mathtt{C}) = [\bar{\mathtt{f}} : [\overline{\mathtt{t}\,\mathtt{D}}]\ \bar{\mathtt{m}} : \bar{\mathtt{T}}\ \mathtt{C}]}$$

$$hexpand(\bar{\mathtt{C}}) = hexpand(\mathtt{C}_1),\ldots,hexpand(\mathtt{C}_n)$$

**Fig. 12** *hexpand* type construction

and method types of a given class are explicitly listed, except method types are assigned abstract effects, and *canonical* hexpansion will always choose fresh variables with which to construct the type.

Inference always assigns hexpanded type forms to expressions (see Lemma 27 for a precise statement of this), which invariant provides a uniform representation for proofs. More importantly, the invariant preserves soundness when imposing constraints due to method or field selection. That is, the selection of a method or field $\mathtt{x}$ from an object of type $[\mathtt{T}\,\mathtt{C}]$ will impose a constraint of the form $\mathtt{T} <: (\mathtt{x} : \mathtt{S})$. If $[\mathtt{T}\,\mathtt{C}]$ is in hexpanded form, a lower bound constraint on the width of $\mathtt{T}$ is hard-coded by the form of $\mathtt{T}$, whereas if $\mathtt{T}$ was a type variable $\mathtt{t}$, a lower bound width constraint on $\mathtt{t}$ would have to be explicitly imposed to ensure soundness.

Since hexpanded types do have abstract forms as subterms, the latter occur in types and constraints in inference judgements, but only serve as conduits for transitive flow during closure. Furthermore, type variables $\mathtt{t}$ never appear "bare" during inference, but only within an abstract object type form $[\mathtt{t}\,\mathtt{C}]$, so that object type variables always have an implicit width constraint associated with $\mathtt{C}$.

## 6.2 Type inference judgements

Type inference rules are given in Fig. 13; the $W$ subscripting the relation $\vdash_W$ distinguishes type inference from logical typing judgements, and is named after the polymorphic type reconstruction algorithm $W$ [15]. The type inference rules are deterministic except for the choice of type variables; we call *canonical* those derivations that always choose fresh type variables, and hereafter restrict our consideration to canonical derivations without loss of generality.

Type inference rules are mostly analogous to their logical counterparts. The T-CLASS rule is slightly different, and the T-CONSTRAINT rule interposed, to deal with mutually recursive class definitions. If $\bar{\mathtt{C}}$ are all mutually recursive, then the T-CLASS rule allows typings to be assigned to them "in parallel", as a group—a group of one, in case a given class is not mutually recursive with any others. We specify a normal form for environments in inference derivations, and also we define the inference analog of well formed environments, which formalizes the notion of complete typing for a full class table.

**Definition 8** Any $\Gamma$ is in *inference form* iff $hexpand(\mathtt{C}) = \mathtt{T}$ for all $\mathtt{C} : \forall\bar{\mathtt{X}}[C].T \in \Gamma$, and for all $\mathtt{x} : \mathtt{S} \in \Gamma$ there exists $\mathtt{D}$ such that $hexpand(\mathtt{D}) = \mathtt{S}$.

T-VAR
$$\Gamma, \mathbf{true}, \epsilon \vdash_W \mathtt{x} : \Gamma(\mathtt{x})$$

T-FIELD
$$\frac{\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : [\mathtt{T}\,\mathtt{C}] \qquad \mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C}) \qquad \mathit{hexpand}(\mathtt{D}) = [\mathtt{S}\,\mathtt{D}]}{\Gamma, C \wedge \mathtt{T} <: (\mathtt{f} : [\mathtt{S}\,\mathtt{D}]), \mathtt{H} \vdash_W \mathtt{e.f} : [\mathtt{S}\,\mathtt{D}]}$$

T-SEQNIL
$$\Gamma, \mathbf{true}, \epsilon \vdash_W \varnothing : \varnothing$$

T-SEQCONS
$$\frac{\Gamma, C_1, \mathtt{H}_1 \vdash_W \bar{\mathtt{e}} : \bar{\mathtt{T}} \qquad \Gamma, C_2, \mathtt{H}_2 \vdash_W \mathtt{e} : \mathtt{T}}{\Gamma, C_1 \wedge C_2, \mathtt{H}_1; \mathtt{H}_2 \vdash_W \bar{\mathtt{e}}, \mathtt{e} : \bar{\mathtt{T}}, \mathtt{T}}$$

T-INVK
$$\frac{\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : [\mathtt{T}\,\mathtt{C}] \qquad \Gamma, D, \mathtt{H}' \vdash_W \bar{\mathtt{e}} : \bar{\mathtt{S}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{D}} \to \mathtt{D} \qquad \mathit{hexpand}(\mathtt{D}) = [\mathtt{S}\,\mathtt{D}]}{\Gamma, C \wedge D \wedge \mathtt{T} <: (\mathtt{m} : \bar{\mathtt{S}} \xrightarrow{\mathtt{h}} [\mathtt{S}\,\mathtt{D}]), \mathtt{H}; \mathtt{H}'; \mathtt{h} \vdash_W \mathtt{e.m}(\bar{\mathtt{e}}) : [\mathtt{S}\,\mathtt{D}]}$$

T-NEW
$$\frac{\Gamma(\mathtt{C}) = \forall \bar{\mathtt{X}}[D].[\mathtt{T}\,\mathtt{C}] \qquad \mathit{fieldsig}[\mathtt{T}\,\mathtt{C}] = \bar{\mathtt{T}} \qquad \Gamma, C, \mathtt{H} \vdash_W \bar{\mathtt{e}} : \bar{\mathtt{S}}}{\Gamma, C \wedge D[\bar{\mathtt{X}}'/\bar{\mathtt{X}}] \wedge \bar{\mathtt{S}} <: \bar{\mathtt{T}}[\bar{\mathtt{X}}'/\bar{\mathtt{X}}], \mathtt{H} \vdash_W \mathtt{new}\ \mathtt{C}(\bar{\mathtt{e}}) : [\mathtt{T}[\bar{\mathtt{X}}'/\bar{\mathtt{X}}]\,\mathtt{C}]}$$

T-EVENT
$$\Gamma, \mathbf{true}, \mathtt{ev[i]} \vdash_W \mathtt{ev[i]} : \mathtt{Unit}$$

T-CHECK
$$\Gamma, \mathbf{true}, \mathtt{chk[i]} \vdash_W \mathtt{chk[i]} : \mathtt{Unit}$$

T-CAST
$$\frac{\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T} \qquad \mathit{hexpand}(\mathtt{D}) = [\mathtt{S}\,\mathtt{D}]}{\Gamma, C \wedge \mathtt{T} \lessdot [\mathtt{S}\,\mathtt{D}], \mathtt{H} \vdash_W (\mathtt{D})\mathtt{e} : [\mathtt{S}\,\mathtt{D}]}$$

T-METH
$$\frac{\Gamma; \bar{\mathtt{x}} : \bar{\mathtt{S}}, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{S} \qquad \mathit{hexpand}(\bar{\mathtt{D}}) = \bar{\mathtt{S}} \qquad \Gamma(\mathtt{this}).\mathtt{m} = [\overline{\mathtt{T}\,\mathtt{D}}] \xrightarrow{\mathtt{h}} \mathtt{T} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}}{\Gamma, C \wedge \mathtt{S} <: \mathtt{T} \wedge [\overline{\mathtt{T}\,\mathtt{D}}] <: \bar{\mathtt{S}} \wedge \mathtt{H} <: \mathtt{h} \vdash_W \mathtt{m}, \mathtt{C} : [\overline{\mathtt{T}\,\mathtt{D}}] \xrightarrow{\mathtt{h}} \mathtt{T}}$$

T-CLASSCONSTRAINT
$$\frac{\mathit{meths}(\mathtt{C}) = \bar{\mathtt{m}} \qquad \Gamma; \mathtt{this} : \Gamma(\mathtt{C}), D_i \vdash_W \mathtt{m}_i, \mathtt{C} : \mathtt{T}_i \text{ for all } \mathtt{m}_i \in \bar{\mathtt{m}}}{\Gamma \vdash_W \mathtt{C} : D_1 \wedge \cdots \wedge D_n, \Gamma(\mathtt{C})}$$

T-CLASS
$$\frac{\Gamma; \bar{\mathtt{C}} : \mathit{hexpand}(\bar{\mathtt{C}}) \vdash_W \mathtt{C}_i : D_i, \mathtt{T}_i \text{ and } \mathrm{fv}(D, \mathtt{T}_i) = \bar{\mathtt{X}}_i \text{ for all } i \in 1..n \qquad D = D_1 \wedge \cdots \wedge D_n}{\Gamma \vdash_W \mathtt{C}_1 : \forall \bar{\mathtt{X}}_1[D].\mathtt{T}_1, \ldots, \mathtt{C}_n : \forall \bar{\mathtt{X}}_n[D].\mathtt{T}_n}$$

**Fig. 13** $\mathrm{FJ}_{\mathrm{trace}}$ type inference rules

Hereafter, we restrict inference judgements to the use of environments in inference form, and observe that derivations preserve this property.

We obtain soundness for type inference via the following result for expression inference; generalization to method and class inference are obtained on this basis. The result follows by induction on inference derivations. The Lemma states that inference derivations can be reconstructed as logical derivations of less general typings; this formulation is necessary to allow the induction to go through, since logical judgements are given complete constraints a priori, whereas they are reconstructed from the leaves towards the root in inference derivations. The proof is given in Sect. 6.4.

**Lemma 22** (Soundness of Inference) *If* $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ *is derivable with* $\Gamma$ *and* $C \wedge D, \mathtt{T}$ *realizable, then* $\Gamma, C \wedge D, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is derivable.*

### 6.3 Closure and extraction of effects

To automatically check satisfiability of constraints, hence realizability of types, the type implementation comprises a constraint closure algorithm and consistency check. We say

C-FN
$(\bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T} <: \bar{\mathtt{S}} \xrightarrow{\mathtt{H}'} \mathtt{S}) \rightsquigarrow_{close} (\bar{\mathtt{S}} <: \bar{\mathtt{T}} \wedge \mathtt{T} <: \mathtt{S} \wedge \mathtt{H} <: \mathtt{H}')$

C-TRANS
$(\mathtt{R} <: \mathtt{S} \wedge \mathtt{S} <: \mathtt{T}) \rightsquigarrow_{close} \mathtt{R} <: \mathtt{T}$

C-STRANS
$(\mathtt{R} <: \mathtt{S} \wedge \mathtt{S} \lessdot \mathtt{T}) \rightsquigarrow_{close} \mathtt{R} \lessdot \mathtt{T}$

C-OBJ
$[(\bar{\mathtt{x}} : \bar{\mathtt{T}})\, \mathtt{C}] <: [(\bar{\mathtt{y}} : \bar{\mathtt{S}})\, \mathtt{D}] \rightsquigarrow_{close} (\bar{\mathtt{x}} : \bar{\mathtt{T}}) <: (\bar{\mathtt{y}} : \bar{\mathtt{S}})$

C-ROW
$(\bar{\mathtt{x}} : \bar{\mathtt{R}}, \bar{\mathtt{y}} : \bar{\mathtt{S}}) <: (\bar{\mathtt{x}} : \bar{\mathtt{T}}) \rightsquigarrow_{close} \bar{\mathtt{R}} <: \bar{\mathtt{T}}$

C-CAST
$$\frac{\mathtt{C} <: \mathtt{D}}{[\mathtt{T}\,\mathtt{C}] \lessdot [\mathtt{S}\,\mathtt{D}] \rightsquigarrow_{close} [\mathtt{T}\,\mathtt{C}] <: [\mathtt{S}\,\mathtt{D}]}$$

C-CONTEXT
$$\frac{C' \subseteq C \qquad C' \rightsquigarrow_{close} D \qquad D \not\subseteq C}{C \rightarrow_{close} C \wedge D}$$

**Fig. 14** Constraint closure rules

$$\vdash \mathbf{true} : ok \qquad \frac{\vdash C : ok \qquad \vdash D : ok}{\vdash C \wedge D : ok} \qquad \vdash \mathtt{H} <: \mathtt{H}' : ok \qquad \frac{\mathtt{C} <: \mathtt{D}}{\vdash [\mathtt{T}\,\mathtt{C}] <: [\mathtt{S}\,\mathtt{D}] : ok}$$

$$\vdash [\mathtt{T}\,\mathtt{C}] \lessdot [\mathtt{S}\,\mathtt{D}] : ok \qquad \vdash (\bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T} <: \bar{\mathtt{S}} \xrightarrow{\mathtt{H}'} \mathtt{S}) : ok \qquad \vdash (\bar{\mathtt{x}} : \bar{\mathtt{R}}, \bar{\mathtt{y}} : \bar{\mathtt{S}}) <: (\bar{\mathtt{x}} : \bar{\mathtt{T}}) : ok$$

**Fig. 15** Constraint consistency rules

that a constraint $C$ is *consistent* iff $\vdash C : ok$ is derivable given the deterministic rules in Fig. 15. For brevity in the definition of closure, we introduce the following notation:

**Definition 9** Let $\hat{C}$ range over *atomic* constraints, i.e.:

$$\hat{C} \ ::= \ \mathbf{true} \mid \mathtt{T} <: \mathtt{T} \mid \mathtt{T} \lessdot \mathtt{T}$$

and for all $C = \hat{C}_1 \wedge \cdots \wedge \hat{C}_n$, let $set(C) = \{C_1, \ldots, C_n\}$. Then define:

$$\hat{C} \in C \iff \hat{C} \in set(C) \qquad\qquad D \subseteq C \iff set(D) \subseteq set(C)$$

Constraint closure is then defined via the rewrite rules given in Fig. 14 and Definition 10. The closure rules are mostly standard, except for those that treat soft subtyping constraints, C-CAST and C-STRANS. These rules implement selective pruning of the constraint graph described previously; note that they effectively discard unsound flow along soft subtyping edges.

**Definition 10** (Closure) The rewrite relations $\rightsquigarrow_{close}$ and $\rightarrow_{close}$ are defined in Fig. 14. $C$ is *closed* iff there does not exist $D$ such that $C \rightarrow_{close} D$. The relation $\rightarrow^{\star}_{close}$ is the reflexive, transitive closure of $\rightarrow_{close}$. We define $close(C)$ as a closed constraint such that $C \rightarrow^{\star}_{close} close(C)$.

Correctness of closure for inferred types and constraints is stated as follows. While our interpretation has slight modifications for specialized object type forms, our approach to type

$$
\begin{aligned}
hextract_C(\epsilon, hs) &= \epsilon \\
hextract_C(\mathtt{ev[i]}, hs) &= \mathtt{ev[i]} \\
hextract_C(\mathtt{chk[i]}, hs) &= \mathtt{chk[i]} \\
hextract_C(\mathtt{H}_1;\mathtt{H}_2, hs) &= (hextract_C(\mathtt{H}_1, hs));(hextract_C(\mathtt{H}_2, hs)) \\
hextract_C(\mathtt{H}_1|\mathtt{H}_2, hs) &= (hextract_C(\mathtt{H}_1, hs))|(hextract_C(\mathtt{H}_2, hs)) \\
hextract_C(\mathtt{h}, \{\mathtt{h}\} \cup hs) &= \mathtt{h} \\
hextract_C(\mathtt{h}, hs) &= \mu\mathtt{h}.hextract_C(bounds_C(\mathtt{h}), hs \cup \{\mathtt{h}\})
\end{aligned}
$$

$$bounds_C(\mathtt{h}) = \mathtt{H}_1|\cdots|\mathtt{H}_n \quad \text{where } \{\mathtt{H}_1, \ldots, \mathtt{H}_n\} = \{\mathtt{H} \mid \mathtt{H} <: \mathtt{h} \in C\}$$

**Fig. 16** *hextract* and *bounds* functions

constraint closure is fundamentally standard, and correctness of standard closure techniques has been thoroughly treated in previous work, notably [31]. Solvability of closed, consistent effect constraints is established constructively in Lemma 24, and solutions of type and effect constraints can be composed to obtain a solution for whole constraints generated by inference. This argument is developed in more detail in [39].

**Lemma 23** *If $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ is derivable, then $C$ is satisfiable iff $close(C)$ is consistent.*

A key property of effect constraints that ensures their satisfiability, is that they define a system of lower bounds on effect variables. That is, if $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ is derivable and $\mathtt{H} <: \mathtt{H}' \in close(\mathtt{C})$, then $\mathtt{H}'$ is an effect variable $\mathtt{h}$. It is easy to demonstrate this property by observing that it is established by inference, and preserved by closure. This form of effect constraint implies that, given $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$, a solution for $\mathtt{H}$ can be obtained, more or less, by recursively joining the lower bounds of type variable components of $\mathtt{H}$ in $close(C)$. The algorithm for extracting term representation of effects from a constraint representation, called *hextract*, is defined in Fig. 16. Since extraction does not alter the given constraint in any way, the fixed parameter $C$ is written as a subscript. Extraction returns a closed trace effect that is a sound term representation of the top-level effect of given closed expressions, in the following sense. The result constructively proves satisfiability of closed, consistent effect constraints.

**Lemma 24** (Extraction Correctness) *If $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ is derivable for closed $e$ and $close(C)$ is consistent, then $hextract_{close(C)}(\mathtt{H})$ is defined and there exists a solution $\rho$ of $C$ such that $hextract_{close(C)}(\mathtt{H}) = \rho(\mathtt{H})$.*

Correctness of the *hextract* algorithm is rigorously established in [39] by a fixpoint construction argument. Since the form of constraints treated in that result is the same as that generated by inference in this paper, the result is entirely applicable.

To make the final connection to logical typing judgements, we must define the analogue of well-formed environments, wherein all type schemes bindings are realizable and logically derivable. In the inference system, this requires that environments contain bindings that are logically inferable, and whose constraint closures are consistent.

**Definition 11** Letting $\sigma$ range over constrained type schemes, and given:

$$\Gamma = \mathtt{C}_1 : \sigma_1; \ldots; \mathtt{C}_n : \sigma_n$$

Then $\Gamma$ is *inferable* iff for all $0 < i \leq n$ there exists $0 < k \leq i$ such that:

$$\mathtt{C}_1 : \sigma_1; \ldots; \mathtt{C}_{k-1} : \sigma_{k-1} \vdash_W \mathtt{C}_k : \sigma_k, \ldots, \mathtt{C}_i : \sigma_i$$

and for all $\forall \bar{\mathtt{X}}[C].T \in \Gamma$, $close(C)$ is consistent.

Putting the pieces together, a complete analysis is obtained by composition of inference, closure, consistency, and *hextract*ion. Validity of effects can be implemented by techniques described in [37].

**Theorem 1** (Soundness of Analysis) *Suppose* $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ *is derivable for inferable* $\Gamma$ *and closed* $\mathtt{e}$, $close(C)$ *is consistent, and* $hextract_{close(C)}(\mathtt{H})$ *is valid. Then* $\epsilon, \mathtt{e} \to^\star \eta, \mathtt{E}[\mathtt{chk}[\mathtt{x}]]$ *implies* $\eta \vdash \mathtt{chk}[\mathtt{x}]$.

The proof is given in Sect. 6.4, following by soundness of inference as established by Lemmas 22, 23, 24, and type safety as established by Lemma 6.

## 6.4 Properties

We now investigate formal properties of type inference in more depth. First, we observe a fairly obvious characteristic of constraints, that any solution of a compound constraint is also a solution of its parts.

**Lemma 25** $C \wedge D \Vdash D$.

Now, we characterize the essential property allowing the innards of an hexpanded object type to be unwrapped, while still retaining a width constraint associated with the object class. This is essential for ensuring soundness of inference. The result is a straightforward consequence of the definition of *hexpand*.

**Lemma 26** *Let* $hexpand(\mathtt{C}) = [\mathtt{T}\,\mathtt{C}]$. *Then if* $\rho(\mathtt{T})$ *is well kinded, so is* $\rho([\mathtt{T}\,\mathtt{C}])$.

We also observe the invariant that all top-level types in inference are in hexpanded form, which are well-formed with respect to any inferred top-level constraint. The result follows by induction on type derivations.

**Lemma 27** *If* $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ *is derivable for well formed* $\Gamma$, *then* $\mathtt{T} = hexpand(\mathtt{C})$, *and satisfiability of* $C$ *implies realizability of* $C, \mathtt{T}$.

Soundness of inference follows by induction on inference derivations. In essence, soundness is proved by transforming an inference derivation into a logical derivation, by allowing the root constraint to be pushed towards the leaves in the induction, via appropriate statement of the result (i.e. the logical derivation permits weakening of the top level constraint with some constraint $D$).

**Lemma 22** (Soundness of Inference) *If* $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ *is derivable with* $\Gamma$ *and* $C \wedge D, \mathtt{T}$ *realizable, then* $\Gamma, C \wedge D, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is derivable.*

*Proof* By induction on the derivation of $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : [\mathtt{T}\,\mathtt{C}]$ and case analysis on the last step.

Cases T-VAR, T-SEQNIL, T-EVENT, and T-CHECK are immediate.

Case T-FIELD. In this case, by inversion of the rule we have:

$$\mathtt{e} = \mathtt{e}_0.\mathtt{f} \qquad \mathit{hexpand}(\mathtt{C}) = [\mathtt{T}\,\mathtt{C}] \qquad C = C_0 \wedge \mathtt{S} <: (\mathtt{f} : [\mathtt{T}\,\mathtt{C}]) \qquad \Gamma, \mathtt{C}_0, \mathtt{H} \vdash_W \mathtt{e} : [\mathtt{S}\,\mathtt{D}]$$

$$\mathtt{C}\ \mathtt{f} \in \mathit{fields}(\mathtt{D})$$

Since by assumption $C$ has a solution $\rho$, therefore $\rho(\mathtt{S})$ is well kinded, so $C, [\mathtt{S}\,\mathtt{D}]$ is realizable by Lemmas 27 and 26. Hence $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : [\mathtt{S}\,\mathtt{D}]$ is derivable by the induction hypothesis, and $C \Vdash \mathtt{S} <: (\mathtt{f} : [\mathtt{T}\,\mathtt{C}])$ by Lemma 25, so the result follows by an instance of T-FIELD.

Case T-INVK. By inversion of the rule in this case we have:

$$C = C_1 \wedge C_2 \wedge \mathtt{S} <: (\mathtt{m} : [\bar{\mathtt{S}}\,\mathtt{B}] \xrightarrow{\mathtt{h}} [\mathtt{T}\,\mathtt{C}]) \qquad \mathtt{e} = \mathtt{e}_0.\mathtt{m}(\bar{\mathtt{e}}) \qquad \mathtt{H} = \mathtt{H}_1; \mathtt{H}_2; \mathtt{h}$$

$$\Gamma, \mathtt{C}_1, \mathtt{H}_1 \vdash_W \mathtt{e}_0 : [\mathtt{S}\,\mathtt{D}] \qquad \Gamma, \mathtt{C}_2, \mathtt{H}_2 \vdash_W \bar{\mathtt{e}} : [\bar{\mathtt{S}}\,\mathtt{B}] \qquad \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \bar{\mathtt{C}} \rightarrow \mathtt{C}$$

$$\mathit{hexpand}(\mathtt{C}) = [\mathtt{T}\,\mathtt{C}]$$

Since $C$ has a solution $\rho$ by assumption, therefore $\rho(\mathtt{S})$ and $\rho([\bar{\mathtt{S}}\,\mathtt{B}])$ are well kinded, so $C, [\bar{\mathtt{S}}\,\mathtt{B}]$ is realizable, and $C, [\mathtt{S}\,\mathtt{D}]$ is realizable by Lemmas 27 and 26. Thus, $\Gamma, C, \mathtt{H}_1 \vdash \mathtt{e}_0 : [\mathtt{S}\,\mathtt{D}]$ and $\Gamma, C, \mathtt{H}_2 \vdash \bar{\mathtt{e}} : [\bar{\mathtt{S}}\,\mathtt{B}]$ by the induction hypothesis, and $C \Vdash \mathtt{S} <: (\mathtt{m} : [\bar{\mathtt{S}}\,\mathtt{B}] \xrightarrow{\mathtt{h}} [\mathtt{T}\,\mathtt{C}])$ by Lemma 25, so the result follows by an instance of T-INVK.

Case T-NEW. In this case, by inversion of the rule we have:

$$\mathtt{e} = \mathtt{new}\ \mathtt{C}(\bar{\mathtt{e}}) \qquad \mathtt{T} = [\mathtt{S}[\bar{\mathtt{X}}'/\bar{\mathtt{X}}]\,\mathtt{C}] \qquad C = C_0 \wedge D_0[\bar{\mathtt{X}}'/\bar{\mathtt{X}}] \wedge \bar{\mathtt{S}} <: \bar{\mathtt{T}}[\bar{\mathtt{X}}'/\bar{\mathtt{X}}]$$

$$\Gamma(\mathtt{C}) = \forall \bar{\mathtt{X}}[\mathtt{D}_0].[\mathtt{S}\,\mathtt{C}] \qquad \mathit{fieldsig}\,[\mathtt{S}\,\mathtt{C}] = \bar{\mathtt{T}} \qquad \Gamma, \mathtt{C}_0, \mathtt{H} \vdash_W \bar{\mathtt{e}} : \bar{\mathtt{S}}$$

But $C \wedge D$ is satisfiable by assumption, so $C \wedge D, \bar{\mathtt{S}}$ is realizable, therefore $\Gamma, C \wedge D, \mathtt{H} \vdash \bar{\mathtt{e}} : \bar{\mathtt{S}}$ is derivable by the induction hypothesis. And by Lemma 25:

$$C \wedge D \Vdash D_0[\bar{\mathtt{X}}'/\bar{\mathtt{X}}] \qquad C \wedge D \Vdash \bar{\mathtt{S}} <: \bar{\mathtt{T}}[\bar{\mathtt{X}}'/\bar{\mathtt{X}}]$$

so the result follows in this case by an instance of T-NEW. The T-CAST and T-SEQ cases follow in a straightforward manner by the induction hypothesis. □

A final technical hurdle is to show that inferable environments are well-formed. The result shows how to obtain logical class typing judgements from inferred ones with consistent closures.

**Lemma 28** *If $\Gamma$ is inferable, then it is well-formed.*

*Proof* Assume that the following judgement is derivable in the inference system, with $\mathit{close}(D)$ consistent:

$$\Gamma \vdash_W \mathtt{C}_1 : \forall \bar{\mathtt{X}}_1[D].\mathtt{T}_i, \ldots, \mathtt{C}_n : \forall \bar{\mathtt{X}}_n[D].\mathtt{T}_n$$

Then it suffices to show that for any $i \in [1..n]$ the judgement:

$$\Gamma; \mathtt{C}_1 : \forall \bar{\mathtt{X}}_1[D].\mathtt{T}_1; \ldots; \mathtt{C}_n : \forall \bar{\mathtt{X}}_n[D].\mathtt{T}_n \vdash \mathtt{D} : \forall \bar{\mathtt{X}}_i[D].\mathtt{T}_i$$

is derivable in the logical system. Inverting the T-CLASS rule we can reconstruct:

$$\frac{\Gamma; \bar{\mathtt{C}} : hexpand(\bar{\mathtt{C}}) \vdash_W \mathtt{C}_i : D_i, \mathtt{T}_i \text{ and } \mathrm{fv}(D, \mathtt{T}_i) = \bar{\mathtt{X}}_i \text{ for all } i \in 1..n \quad D = D_1 \wedge \cdots \wedge D_n}{\Gamma \vdash_W \mathtt{C}_1 : \forall \bar{\mathtt{X}}_1[D].\mathtt{T}_1, \ldots, \mathtt{C}_n : \forall \bar{\mathtt{X}}_n[D].\mathtt{T}_n}$$

where for each $i \in 1..n$, the constraint $D_i$ is of the form $C_1 \wedge \cdots \wedge C_j$, and for each $\mathtt{m}_k \in$ $meths(\mathtt{C}_i)$ a judgement:

$$\Gamma; \bar{\mathtt{C}} : hexpand(\bar{\mathtt{C}}); \mathtt{this} : \Gamma(\mathtt{C}), C_k \vdash_W \mathtt{m}_k, \mathtt{C}_i : \mathtt{T}_k$$

is derivable, by inversion of the T-CLASSCONSTRAINT rule. Let:

$$\Gamma' \triangleq (\Gamma; \bar{\mathtt{C}} : hexpand(\bar{\mathtt{C}}); \mathtt{this} : \Gamma(\mathtt{C}))$$

Then by inversion of the T-METH rule we have that $\mathtt{T}_k$ is of the form $[\mathtt{T}\bar{\mathtt{D}}] \xrightarrow{\mathtt{h}} \mathtt{T}$, and $C_k$ is of the form:

$$C \wedge \mathtt{S} <: \mathtt{T} \wedge [\mathtt{T}\bar{\mathtt{D}}] <: \bar{\mathtt{S}} \wedge \mathtt{H} <: \mathtt{h}$$

and we can reconstruct:

$$\frac{hexpand(\bar{\mathtt{D}}) = \bar{\mathtt{S}} \qquad \begin{array}{c}\Gamma'; \bar{\mathtt{x}} : \bar{\mathtt{S}}, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{S} \\ \Gamma'(\mathtt{this}).\mathtt{m}_k = [\mathtt{T}\bar{\mathtt{D}}] \xrightarrow{\mathtt{h}} \mathtt{T}\end{array} \qquad mbody(\mathtt{m}_k, \mathtt{C}_i) = \bar{\mathtt{x}}.\mathtt{e}}{\Gamma', C \wedge \mathtt{S} <: \mathtt{T} \wedge [\mathtt{T}\bar{\mathtt{D}}] <: \bar{\mathtt{S}} \wedge \mathtt{H} <: \mathtt{h} \vdash_{\mathtt{W}} \mathtt{m}_k, \mathtt{C}_i : [\mathtt{T}\bar{\mathtt{D}}] \xrightarrow{\mathtt{h}} \mathtt{T}}$$

Since $close(D)$ is consistent by assumption, therefore $D$ has a solution by Lemma 23. And since $\mathtt{S}$ occurs in $D$, therefore $D, \mathtt{S}$ is realizable, hence by Lemma 22 the judgement $\Gamma'; \bar{\mathtt{x}} : \bar{\mathtt{S}}, D, \mathtt{H} \vdash \mathtt{e} : \mathtt{S}$ is derivable in the logical system, from which can be derived $\Gamma'; \bar{\mathtt{x}} : \bar{\mathtt{S}}, D, \mathtt{h} \vdash \mathtt{e} : \mathtt{S}$ by Lemma 25 and an instance of T-WEAKEN. But then the form of $C_k$ as noted above, Lemma 25, and an instance of T-METH in the logical system allows us to derive $\Gamma', D \vdash \mathtt{m}_k, \mathtt{C}_i : [\mathtt{T}\bar{\mathtt{D}}] \xrightarrow{\mathtt{h}} \mathtt{T}$. But since $[\mathtt{T}\,\mathtt{D}]$ and $\mathtt{h}$ and $\mathtt{T}$ all occur in $D$, therefore $D, [\mathtt{T}\bar{\mathtt{D}}] \xrightarrow{\mathtt{h}} \mathtt{T}$ is realizable. The result follows by an instance of T-CLASS in the logical system. □

Now we can demonstrate our main result, showing how the different components of inference can be combined to generate a sound automated type analysis. The result follows by soundness of inference and closure, correctness of *hextract*ion, and type safety.

**Theorem 1** (Soundness of Analysis) *Suppose* $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ *is derivable for inferable* $\Gamma$ *and closed* $\mathtt{e}$, $close(C)$ *is consistent, and* $hextract_{close(C)}(\mathtt{H})$ *is valid. Then* $\epsilon, \mathtt{e} \to^\star \eta, \mathtt{E}[\mathtt{chk}[\mathtt{x}]]$ *implies* $\eta \vdash \mathtt{chk}[\mathtt{x}]$.

*Proof* If $\Gamma$ is inferable, then it is well-formed by Lemma 28. If $close(C)$ is consistent, then $C$ has a solution by Lemma 23. Therefore $C, \mathtt{T}$ is realizable by Lemma 27, and $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ is derivable in the logical system by Lemma 22. By Lemma 24, there exists a solution $\rho$ of $C$ such that $hextract_{close(C)}(\mathtt{H}) = \rho(H)$, the validity of which establishes validity of $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$. The result follows by Lemma 6. □

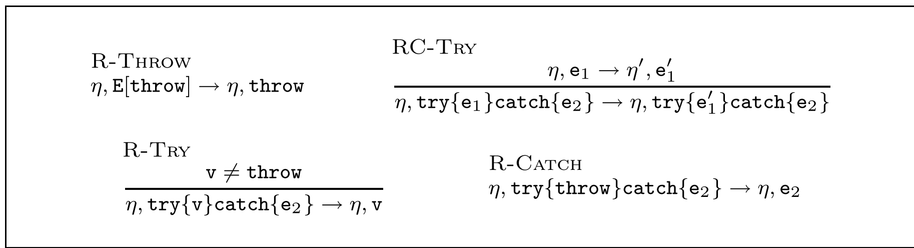

**Fig. 17** Semantics of exceptions

## 7 Control flow and effect transformations

In this section we show that a benefit of our approach is that trace effect representations are amenable to transformational techniques, for flexibility of analysis. These transformations can be used to post-process inferred effects, without requiring any reworking of the inference component of analysis. This means that certain extensions of the language can be treated statically in a *modular* fashion, including extensions that have a fundamental effect on program semantics and control flow.

We consider *exnization* and *stackification* transformations. Exnization implements the impact of exceptions on effect representations. Stackification is useful in a stack-based safety context– that is, where events associated with function activations are "forgotten" when that activation returns, as in e.g. Java stack inspection. Thus, security contexts are established by the current calling context.

### 7.1 A Transformation for exceptions

Exceptions exist in Java, so a realistic application of our approach must account for them. In this section we consider a first approximation of the full exception feature set, where we assume there exists only one anonymous exception in the language. The $\mathrm{FJ}_{\mathrm{trace}}$ language of expressions is extended to include the form $\mathtt{throw}$ for throwing this anonymous exception, and to include the form $\mathtt{try}\{\mathtt{e}_1\}\mathtt{catch}\{\mathtt{e}_2\}$ for handling thrown exceptions, yielding the language $\mathrm{FJ}_{\mathrm{exn}}$. The semantics of $\mathrm{FJ}_{\mathrm{exn}}$ are the semantics of $\mathrm{FJ}_{\mathrm{trace}}$ extended with the rules in Fig. 17, where evaluation contexts $\mathtt{E}$ are as specified in Definition 6.

#### *7.1.1 Logical type system*

To treat these new language forms in type analysis, we introduce two new forms to the language of effects: $\mathtt{throw}$ to identify control flow points where an exception is thrown, and $\mathtt{H}_1 \Rsh \mathtt{H}_2$ to represent the effect of handlers, where $\mathtt{H}_1$ represents the effect of the $\mathtt{try}$ clause, and $\mathtt{H}_2$ represents the effect of the $\mathtt{catch}$ clause. We endow these forms with an LTS semantics appropriate to the behavior of exceptions, as follows:

$$\mathtt{throw}; \mathtt{H} \xrightarrow{\epsilon} \mathtt{throw} \qquad \mathtt{throw} \Rsh \mathtt{H} \xrightarrow{\epsilon} \mathtt{H} \qquad \epsilon \Rsh \mathtt{H} \xrightarrow{\epsilon} \epsilon$$

$$\mathtt{H}_1 \Rsh \mathtt{H}_2 \xrightarrow{a} \mathtt{H}_1' \Rsh \mathtt{H}_2 \quad \text{if } \mathtt{H}_1 \xrightarrow{a} \mathtt{H}_1'$$

The transition rules ensure that for any sequence of events $\mathtt{H}_1; \mathtt{H}_2$, if $\mathtt{H}_1$ encounters a $\mathtt{throw}$, then none of the events in $\mathtt{H}_2$ are reached. Also, if $\mathtt{H}_1$ in the effect $\mathtt{H}_1 \Rsh \mathtt{H}_2$ of some handler

T-THROW
$$\Gamma, \mathbf{true}, \mathtt{throw} \vdash \mathtt{throw} : [\mathtt{T}\ \mathtt{C}]$$

T-TRYCATCH
$$\frac{\Gamma, C, \mathtt{H}_1 \vdash \mathtt{e}_1 : [\mathtt{T}\ \mathtt{C}] \qquad \Gamma, C, \mathtt{H}_2 \vdash \mathtt{e}_2 : [\mathtt{T}\ \mathtt{C}]}{\Gamma, C, \mathtt{H}_1 \Rsh \mathtt{H}_2 \vdash \mathtt{try}\{\mathtt{e}_1\}\mathtt{catch}\{\mathtt{e}_2\} : [\mathtt{T}\ \mathtt{C}]}$$

**Fig. 18** Logical typing rules for exceptions

encounters a throw event, then the transition rules ensure that the effect $\mathtt{H}_2$ of the handling clause is encountered. Otherwise, if $\mathtt{H}_1$ transitions safely to $\epsilon$ without encountering a throw then the events in $\mathtt{H}_2$ are never encountered.

We also redefine the interpretation of trace effects as follows, to account for the fact that effect computation may be terminated by an uncaught exception:

$$[\![\mathtt{H}]\!]_{exn} = \begin{array}{l} \{a_1 \cdots a_n \quad \mid \mathtt{H} \xrightarrow{a_1} \cdots \xrightarrow{a_n} \mathtt{H}'\} \\ \qquad\qquad \cup \\ \{a_1 \cdots a_n \downarrow \mid \mathtt{H} \xrightarrow{a_1} \cdots \xrightarrow{a_n} \mathtt{H}' \text{ where } \mathtt{H}' \in \{\epsilon, \mathtt{throw}\}\} \end{array}$$

We refer to this interpretation as the *exn interpretation* of effects, as opposed to the previous *exn-free interpretation* of effects. The definition of primitive subtyping is modified to accommodate this interpretation of effects:

$$\frac{[\![\mathtt{H}]\!]_{exn} \subseteq [\![\mathtt{H}']\!]_{exn}}{\mathtt{H} \preccurlyeq_{\text{fin}} \mathtt{H}'}$$

Finally, we extend the logical type system to account for exceptions and handlers, as in Fig. 18.

While the exn interpretation of effects is a sufficient and appealing approach to obtaining type safety, it has the drawback that throw and $\mathtt{H}_1 \Rsh \mathtt{H}_2$ forms are non-standard and not treated by existing model-checking techniques. Our solution to this is to soundly transform $\mathrm{FJ}_{\mathrm{exn}}$ effects into $\mathrm{FJ}_{\mathrm{trace}}$ effects. In particular, we are interested in a class of functions we call *exn transformers*, that take an effect $\mathtt{H}$ and return a pair of sets $s, t$ of exn-free trace effects, where $s$ generates the *safe* traces in $[\![\mathtt{H}]\!]$ that are not terminated by a raised exception, and $t$ generates the remaining *throw* traces in $[\![\mathtt{H}]\!]$. To manipulate these sets in subsequent development, we define the following notation:

$$s_1; s_2 = \{\mathtt{H}_1; \mathtt{H}_2 \mid \mathtt{H}_1 \in s_1 \text{ and } \mathtt{H}_2 \in s_2\} \qquad\qquad s[\mathtt{H}/\mathtt{h}] = \{\mathtt{H}'[\mathtt{H}/\mathtt{h}] \mid \mathtt{H}' \in s\}$$

$$\begin{aligned} join(\{\mathtt{H}\}) &= \mathtt{H} \\ join(\{\mathtt{H}\} \cup s) &= \mathtt{H}|join(s) \end{aligned}$$

We equate sets of effects up to their join interpretation, i.e. $s = t$ iff $join(s) = join(t)$. Formally, we define exn transformers as follows:

**Definition 12** An *exn transformer* is a total function $f$ on trace effects such that for all $\mathtt{H}$, $f(\mathtt{H})$ is a pair of trace effect sets $(s, t)$ such that $[\![\mathtt{H}]\!]_{exn} \subseteq [\![join(s \cup t)]\!]$.

We precisely characterize the behavior of exn transformers via the combinator *XZ* defined in Fig. 19, clearly illustrating the relation between input and output of transformation of

$$
\begin{aligned}
XZ\ f\ \epsilon &= \{\epsilon\}, \varnothing \\
XZ\ f\ \mathtt{ev[i]} &= \{\mathtt{ev[i]}\}, \varnothing \\
XZ\ f\ \mathtt{throw} &= \varnothing, \{\epsilon\} \\
XZ\ f\ \mathtt{H}_1|\mathtt{H}_2 &= \text{let } s_1, t_1 = f\ \mathtt{H}_1 \text{ in} \\
&\quad\ \text{let } s_2, t_2 = f\ \mathtt{H}_2 \text{ in} \\
&\quad\ s_1 \cup s_2, t_1 \cup t_2 \\
XZ\ f\ \mathtt{H}_1;\mathtt{H}_2 &= \text{let } s_1, t_1 = f\ \mathtt{H}_1 \text{ in} \\
&\quad\ \text{let } s_2, t_2 = f\ \mathtt{H}_2 \text{ in} \\
&\quad\ s_1; s_2, t_1 \cup (s_1; t_2) \\
XZ\ f\ \mathtt{H}_1 \Rsh \mathtt{H}_2 &= \text{let } s_1, t_1 = f\ \mathtt{H}_1 \text{ in} \\
&\quad\ \text{let } s_2, t_2 = f\ \mathtt{H}_2 \text{ in} \\
&\quad\ s_1 \cup (t_1; s_2), t_1; t_2 \\
XZ\ f\ \mu\mathtt{h.H} &= f\ \mathtt{H}[\mu\mathtt{h.H/h}]
\end{aligned}
$$

**Fig. 19** The *XZ* combinator

T-THROW
$$\Gamma, \mathbf{true}, \mathtt{throw} \vdash_W \mathtt{throw} : \mathtt{t}$$

T-TRYCATCH
$$\frac{\Gamma, C, \mathtt{H}_1 \vdash_W \mathtt{e}_1 : [\mathtt{T}\ C] \qquad \Gamma, D, \mathtt{H}_2 \vdash_W \mathtt{e}_2 : [\mathtt{S}\ C]}{\Gamma, C \wedge D \wedge \mathtt{S} <: \mathtt{t} \wedge \mathtt{T} <: \mathtt{t}, \mathtt{H}_1 \Rsh \mathtt{H}_2 \vdash_W \mathtt{try}\{\mathtt{e}_1\}\mathtt{catch}\{\mathtt{e}_2\} : [\mathtt{t}\ C]}$$

**Fig. 20** Type inference rules for exceptions

closed effects. For example, all the throw paths of $\mathtt{H}_1$ are prefixes of the safe and throw paths of $\mathtt{H}_2$ in the transform interpretation of $\mathtt{H}_1 \Rsh \mathtt{H}_1$, reflecting its exn interpretation. An important consequence of the definition of *XZ* and the exn interpretation of effects is:

**Lemma 29** *Any fixpoint of XZ is an exn transformer.*

It would be possible to show that *XZ* is monotonic, guaranteeing the existence of an exn transformer. However, we will instead define an algorithm that is shown to be a fixpoint of *XZ*, providing a technique for applying standard model checking techniques to $\mathrm{FJ}_{\mathrm{exn}}$ type safety enforcement.

### 7.1.2 Type inference system

To obtain type inference for $\mathrm{FJ}_{\mathrm{exn}}$, we extend the $\mathrm{FJ}_{\mathrm{trace}}$ type inference system with the rules specified in Fig. 20 for language forms related to exceptions. Also necessary is an extension of *hextract* to accommodate new effect forms:

$$
\begin{aligned}
&\mathit{hextract}(\mathtt{throw}, \mathit{hs}) = \mathtt{throw} \\
&\mathit{hextract}(\mathtt{H}_1 \Rsh \mathtt{H}_2, \mathit{hs}) = \mathit{hextract}(\mathtt{H}_1, \mathtt{hs}) \Rsh \mathit{hextract}(\mathtt{H}_2, \mathtt{hs})
\end{aligned}
$$

This completes the logical specification of the system, for which we can demonstrate a subject reduction result as in Lemma 31 below. But to apply standard model checking techniques to $\mathrm{FJ}_{\mathrm{exn}}$ as discussed above, we now define an exn transformer. The core of it

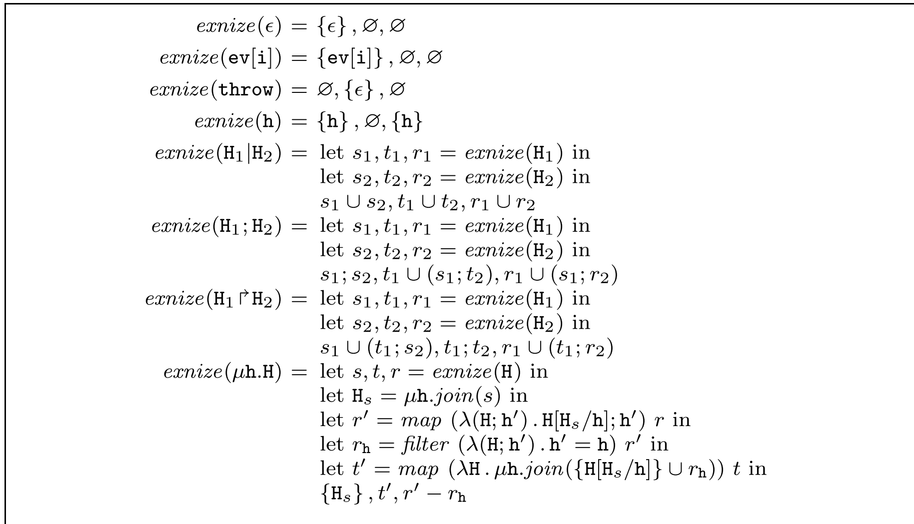

**Fig. 21** The exception transformation function *exnize*

is a function called *exnize*, defined in Fig. 21. Since termination in the algorithm precludes unrolling of $\mu$-bound effects, it must deal with free variables. Thus, in addition to returning safe and throw paths, *exnize* returns a set of effects called *precursors*. Precursors are safe paths that end in a variable (tail recursive paths); they may be "pruned back" from other paths (safe, throw, or recursive), since if a recursive call causes a throw, everything after the call will be short-circuited until the first enclosing handler. The idea is that precursors should be joined with throw paths within a $\mu$-binding, since any number of recursive calls can be made before a throw is encountered; this yields $\mu$-bound throw paths. For example, given:

$$\mathtt{H} \triangleq \mu\mathtt{h}.\mathtt{ev[1]}|(\mathtt{ev[2]};\mathtt{h};\mathtt{ev[3]})|(\mathtt{ev[4]};\mathtt{throw};\mathtt{ev[5]})$$

we yield the following safe and throw paths obtained from H:

$$\textit{safe:}\ \ \mu\mathtt{h}.\mathtt{ev[1]}|(\mathtt{ev[2]};\mathtt{h};\mathtt{ev[3]}) \qquad \textit{throw:}\ \ \mu\mathtt{h}.\mathtt{ev[4]}|(\mathtt{ev[2]};\mathtt{h})$$

An added complication is that recursive calls h may precede other tail recursions or throws in recursive or throw paths; the analysis replaces these "inner" recursions with the safe recursive call paths of h, to obtain safe preceding ground paths.

In more detail, the algorithm *exnize*, defined in Fig. 21, returns a triple $s, t, r$, where $s$ are the safe paths, $t$ are the throw paths, and $r$ are the precursors, each represented as history effect sets. The exception transformation is defined via some auxiliary functions, including *map* and *filter* defined as usual (where the latter accepts only values that match the given predicate). We also adapt a pattern-matching syntax, so that functions $(\lambda(\mathtt{H};\mathtt{h})\ldots)$ match effect arguments of the form $\mathtt{H};\mathtt{h}$; we observe that precursors are guaranteed to be of the this form, by definition of *exnize* and type inference.

At the top-level, the ultimate transformation is the join of the deduced safe and throw paths; note also that at the top-level, the set of precursors should be empty. Thus, the excep-

tion transformation of an effect $\mathtt{H}$ is implemented as:

$$\text{let } s, t, \varnothing = \mathit{exnize}(\mathtt{H}) \text{ in } \mathit{join}(s \cup t)$$

For brevity, we have excluded a special subcase of $\mathit{exnize}(\mu\mathtt{h}.\mathtt{H})$, where the recursive call $\mathit{exnize}(\mathtt{H})$ returns $s, t, r$ such that $s = \varnothing$. However, this is the case where every program control path through a function throws an exception, which we believe will be rare, and can be easily dealt with by modifying the case where $s$ is not empty.

*7.1.3 Properties*

We now establish relevant results for the theory developed in this section, in particular we demonstrate type soundness, and show that the *exnize* transformation obtains a sound approximation of the *exn* interpretation in the *exn*-free interpretation of effects.

Type soundness is proved by extending Lemma 5, in light of the definitions above. In order to prove the R-THROW case, we observe that the effect of an expression $\mathtt{E[throw]}$ will reflect that $\mathtt{throw}$ is the "first thing that happens".

**Lemma 30** *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{E[throw]} : \mathtt{T}$ *then* $C \Vdash \mathtt{throw}; \mathtt{H}' <: \mathtt{H}$.

This lemma is proved in the identical manner as Lemma 21, since evaluation context forms are conserved. Now, we can easily prove subject reduction for $\mathrm{FJ}_{\mathrm{exn}}$.

**Lemma 31** ($\mathrm{FJ}_{\mathrm{exn}}$ Subject Reduction) *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : [\mathtt{TC}]$ *is derivable for closed* $e$ *and well-formed* $\Gamma$, *and* $\eta, \mathtt{e} \to \eta', \mathtt{e}'$, *then* $\Gamma, C, \mathtt{H}' \vdash \mathtt{e}' : [\mathtt{TC}]$ *is derivable with* $C \Vdash \eta'; \mathtt{H}' <: \eta; \mathtt{H}$.

*Proof* Since the semantics of $\mathrm{FJ}_{\mathrm{exn}}$ are conservative with respect to the semantics of $\mathrm{FJ}_{\mathrm{trace}}$, Lemma 5 implies the result for the exn-free subset of $\mathrm{FJ}_{\mathrm{exn}}$. What remains is to prove the RC-TRY, R-TRY, and R-CATCH, and R-THROW cases. The first three follow in a straightforward manner by definition of logical typing for exception handlers, the interpretation of effects $\mathtt{H}_1 \upharpoonleft \mathtt{H}_2$, and the induction hypothesis. The interesting case is R-THROW, which goes as follows. We have $\mathtt{e} = \mathtt{E[throw]}$, $\mathtt{e}' = \mathtt{throw}$, and $\eta = \eta'$, and by Lemma 30 it is the case that $C \Vdash \mathtt{throw}; \mathtt{H}'' <: \mathtt{H}$ for some $\mathtt{H}''$. But $\Gamma, C, \mathtt{throw} \vdash \mathtt{throw} : [\mathtt{TC}]$ by T-THROW, and it is easy to show that $C \Vdash \mathtt{throw} <: \mathtt{throw}; \mathtt{H}''$, given the LTS semantics of $\mathtt{throw}$, therefore $C \Vdash \eta; \mathtt{throw} <: \eta; \mathtt{throw}; \mathtt{H}''$ by Lemma 1. □

Finally, we establish that *exnize* provides an exn transformer, via the following lemma.

**Lemma 32** ($\lambda x.\mathit{let}\ s, t, \varnothing = \mathit{exnize}(x)\ \mathit{in}\ s, t$) *is an exn transformer.*

We just need to show that the function is a fixpoint of *XZ*. Clearly, the main issue is to prove that *exnize* deals with variables and the $\mu\mathtt{h}.\mathtt{H}$ case correctly. This is essentially established by the next auxiliary lemma.

**Lemma 33** *Let all identifiers be as defined in the* $\mu\mathtt{h}.\mathtt{H}$ *case of exnize, and define* $s_{\circlearrowleft}, r_{\circlearrowleft}, t_{\circlearrowleft} = \mathit{exnize}(H[\mu\mathtt{h}.\mathtt{H}/h])$. *Then*:

1. $s_{\circlearrowleft} = s[\mathtt{H}_s/\mathtt{h}]$

2. $r_{\circlearrowleft} = r' - r_{\mathtt{h}}$
3. $t_{\circlearrowleft} = (map\ (\lambda \mathtt{H}.\mathtt{H}[\mathtt{H}_s/\mathtt{h}])\ t) \cup (\bigcup (map\ (\lambda(\mathtt{H};\mathtt{h}).\{\mathtt{H}\}; t')\ r_{\mathtt{h}}))$
4. $join(s_{\circlearrowleft}) = \mathtt{H}_s$
5. $join(t_{\circlearrowleft}) = join(t')$

*Proof* (Sketch) Each property follows by a straightforward induction on H, and (intuitively) since the computation of *exnize*(H[$\mu$h.H/h]) will encounter $\mu$h.H in recursive calls, rather than h, hence the latter will be replaced with appropriate parts of *exnize*($\mu$h.H) in the transformation. In essence, property (1) follows since the safe paths of *exnize*(H[$\mu$h.H/h]) will be the same as the safe paths of *exnize*(H), except with every instance of h replaced with the safe paths of *exnize*($\mu$h.H)—that is, $\mathtt{H}_s$. Property (2) follows since the only precursors in $r'$ not also returned by *exnize*(H[$\mu$h.H/h]) will be those terminated by h—that is, $r_{\mathtt{h}}$—since h is no longer free in H[$\mu$h.H/h]. Property (3) follows by definition of the $\mathtt{H}_1; \mathtt{H}_2$ case of *exnize*, and since *exnize*(H[$\mu$h.H/h]) will encounter the same throw terminated paths as *exnize*(H), except with the safe paths of *exnize*($\mu$h.H) substituted for internal occurrences of h, since free h encountered by *exnize*(H) will instead be occurrences of $\mu$h.H. For the same reason, the h-terminated precursors of *exnize*(H) will be treated as safe prefixes of throw paths of *exnize*($\mu$h.H) in *exnize*(H[$\mu$h.H/h]). Properties (4) and (5) follow immediately by (1) and (3) and properties of effect equivalence as described in Lemma 1. □

Now we can establish the precondition of the main result for *exnize*.

**Lemma 34** *($\lambda x.let\ s, t, \varnothing = exnize(x)\ in\ s, t$) is a fixpoint of XZ.*

*Proof* All cases are trivial except the $\mu$h.H case, where it remains to be shown that *exnize*($\mu$h.H) = *exnize*(H[$\mu$h.H]) by definition of *XZ*—but this case follows immediately by Lemma 33 (2), (4) and (5). □

### 7.2 A transformation for stack-based policies

Rather than consistently accruing events in a trace, a stack-based model can be defined where events generated by method activations are associated with the activation call-stack frame; when the activation is popped, so are the associated events. This allows security decisions to be made with respect to events in the current calling context. The Java stack inspection [20] access control mechanism, for example, is based on sequences of events on the call stack. A stack-based access control model has additionally been combined with a history-based security mechanism in [1], and a static analysis for enforcing general stack-based properties via temporal logic is presented in [8]. Direct type inference for a stack-based security model has been studied previously, e.g. in [36]; however, since security mechanisms such as that proposed in [1] require both a history- and stack-based perspective, we believe that our uniform approach is simpler than e.g. combining direct stack- and history-based inference in such a context.

In this section, we observe that a stack-based security model can be statically enforced, not by redefining the inference system discussed in Sect. 6, but by an effect post-processing technique called *stackification*. Stackification takes as input an effect that predicts the trace generated by a program, and returns the stack contexts generated by a program. The stack contexts generated by a program are formalized by refiguring the $\mathrm{FJ}_{\mathrm{trace}}$ operational semantics with regard to stacks, rather than histories, yielding the language model we call $\mathrm{FJ}_{\mathrm{stack}}$.

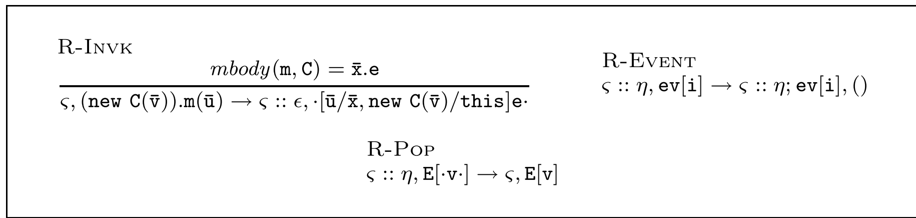

Fig. 22 The stack-based semantics of $FJ_{stack}$

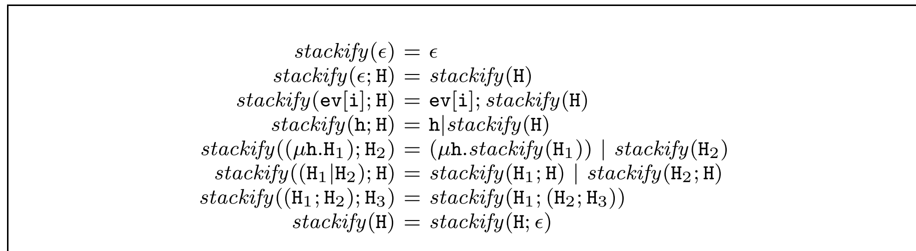

Fig. 23 The *stackify* algorithm

Stack contexts maintain a notion of ordering; hence stacks, which we denote $\varsigma$, are LIFO sequences of histories, and are either **nil** or constructed with a cons operator (::):

$$\varsigma ::= \mathbf{nil} \mid \varsigma :: \eta \qquad \textit{trace stacks}$$

The $FJ_{stack}$ source language is identical to $FJ_{trace}$, except that so-called framed expressions $\cdot\mathtt{e}\cdot$ are included, to delimit regions of code associated with a stack frame. Stack frames are associated with activations in keeping with the standard stack-based model. We also extend evaluation contexts (Definition 6) with the form $\cdot\mathtt{E}\cdot$. Thus, in the operational semantics defined in Fig. 22, the rule governing method invocation, R-INVK, will push a new frame on the stack, and delimit the code region associated with the activation. Frames are popped, as in R-POP, when activations return the result of evaluation. Events are accrued in order within an activation frame, as in R-EVENT.

Given this model, a static analysis in $FJ_{stack}$ will approximate the stack contexts that can be generated during program execution. As mentioned above, we accomplish this by a *stackify* transformation, which takes as input trace effects output by $FJ_{trace}$ type inference, which is also applicable to $FJ_{stack}$ source programs (framed expressions are only generated at run-time). A phenomenon exploited by our transformation is that the scope of inferred method effects is always delimited by a $\mu$-binding. This is because the *hextract* algorithm will resolve any trace effect variable $h$ as a $\mu$-bound effect, and every method is assigned a variable $h$ as its effect during inference. In other words, stack "pushes" and "pops" are implicitly recorded during inference as the beginning and end of $\mu$-scope.

This means that *stackify*, defined in Fig. 23, can use the syntax of effects to recognize corresponding pushes and pops. Note that in the transformation of $\mu$-bound effects, any effects $\mathtt{H}_2$ following a $\mu$-bound effect $\mathtt{H}_1$ will be considered as part of a different stack context, since $\mathtt{H}_1$ is associated with an activation that will be pushed and popped before any events predicted by $\mathtt{H}_2$ can occur.

The *stackify* algorithm generally exploits a normal form representation of effects as a sequence $\mathtt{H}_1; \mathtt{H}_2$. The last three clauses use trace effect equalities to massage trace effects into this normal form. Observe that the range of *stackify* consists of trace effects that are all tail-recursive; stacks are therefore finite-state transition systems and more efficient model-checking algorithms are possible for stacks than for general histories [18].

*Example 1* With a, b and c representing arbitrary events, and results of stackification simplified via effect equivalences to increase readability:

$$\mathit{stackify}(\mathtt{a};\mathtt{b}) = \mathtt{a};\mathtt{b} \qquad \mathit{stackify}(\mathtt{a};(\mu\mathtt{h}.\mathtt{b})) = \mathtt{a};\mathtt{b} \qquad \mathit{stackify}((\mu\mathtt{h}.\mathtt{a});\mathtt{b}) = \mathtt{a}|\mathtt{b}$$

$$\mathit{stackify}(\mu\mathtt{h}.\mathtt{a}|(\mathtt{b};\mathtt{h})) = (\mu\mathtt{h}.\mathtt{a}|(\mathtt{b};\mathtt{h}))|\epsilon \qquad \mathit{stackify}(\mu\mathtt{h}.\mathtt{a}|(\mathtt{h};\mathtt{b})) = (\mu\mathtt{h}.\mathtt{a}|\mathtt{b}|\mathtt{h})|\epsilon$$

In the second example, since $\mu\mathtt{h}.\mathtt{a}$ precedes b, but $\mu\mathtt{h}.\mathtt{a}$ denotes the effect of a function call, stackification specifies that no events precede b since a will be popped before encountering b. In the last example, since b is preceded by h, which represents recursive $\mu$-scope and hence a recursive call, any events preceding b will be popped before b is encountered, hence stackification specifies that no events precede b. The $\epsilon$ in the last two examples is an artifact of the transformation, that could be cleaned up with some minor alterations to *stackify*.

The astute reader may notice that the form of stackification presented here exhibits the unappealing property of not necessarily preserving equivalence of interpretations, for example $\mathit{stackify}(\mu\mathtt{h}.\mathtt{ev}[1]) \neq \mathit{stackify}(\mathtt{ev}[1])$. However, this form of stackification is an abbreviation that suppresses details of a full encoding presented in [39]. In the full encoding, explicit “push” and “pop” events delimit function scope, and stackification does preserve equivalence of interpretations. In the abbreviation presented here, push and pop events are implicitly represented by function scope at the term level, and $\mu$-scope at the type level.

## 8 Conclusion

In this section we conclude with a discussion of related work and some final remarks.

### 8.1 Related work

Previous work relevant to the application of trace-based security models has been noted in Sect. 1. A number of different systems have been developed to enforce trace-based, or temporal, properties of program execution. Perhaps the principal division between them is run-time [1, 34] vs. compile-time [4, 8, 13, 33] verification. The focus of this paper is on the latter, which have in common the idea of extracting an abstract interpretation of some form from a program and verifying properties of that abstraction. The MOPS system [13] compiles C programs to Push-down Automata (PDAs) reflecting the program control flow, where transitions are program transitions and the automaton stack abstracts the program call stack. [8, 26] assume that some (undefined) algorithm has already converted a program to a control flow graph, expressed as a form of PDA.

These aforementioned abstractions work well for procedural programs, but are not powerful enough to fully address advanced language features such as higher order functions and objects. Our type and effect [3, 41] approach, on the other hand, allows abstract interpretation of higher order [37] and Object Oriented programs. Trace effects yielded by the

analysis provide a conservative approximation of trace behavior via an LTS (Labelled Transition System) interpretation. This allows the expression of program assertions as temporal logical formulae and the automated verification of assertions via model-checking techniques [40]. The flavor of polymorphism we use is essentially ML-style let-polymorphism extended with effects, where polymorphic recursion is disallowed. Other type and effect systems with similar flavors of polymorphism have been sufficiently expressive to allow polymorphic recursion, including systems for safe region-based memory management [42, 43]. Another approach to region based memory management uses static union types for flexibility in the presence of dynamic type checking [30], and is similar to our treatment of downcasting in the presence of dynamic cast checks.

Some of the aforecited systems also automatically verify assertions at compile-time via model-checking, including [4, 8, 13], though none of these define a rigorous process for extracting an LTS from higher order or Object Oriented programs. In these works, the specifications are temporal logics, regular languages, or finite automata, and the abstract control flow is extracted as an LTS in the form of a finite automaton, grammar, or PDA. These particular formats are chosen because these combinations of logics and abstract interpretations can be automatically model-checked.

Security automata [34] use finite automata for the specification and run-time enforcement of language safety properties. Systems have also been developed for statically verifying correctness of security automata using dependent types [45] and in a more general form as refinement types [29]. These systems do not extract any abstract interpretations, so they are in a somewhat different category than the aforementioned (and our) work.

Logical assertions can be local, concerning a particular program point, or global, defining the whole behavior required. However, access control systems [1, 16, 46], use local checks. Since we are interested in the static enforcement of access control mechanisms, the focus in this paper is on local, compile-time checkable assertions, though in principle the verification of global properties is possible in our system. Related work has also modified our basic approach to enforce “policy framings” that support so-called local liveness properties [6].

Perhaps the most closely related work is [27], which proposes a similar type and effect system and type inference algorithm, but their “resource usage” abstraction is of a markedly different character, based on grammars rather than LTSs. Their system lacks parametric polymorphism, which restricts expressiveness in practice, and verifies global, rather than local, assertions. Furthermore, their system analyzes only history-based properties, not stack-based properties as in our system. The system of [23] is based on linear types, not effect types. Their usages $U$ are similar to our history effects $H$, but the usages have a much more complex grammar, and assert legal patterns of resource access, which resources can be generated dynamically, unlike our events which are statically declared. This analysis has more recently been extended to a language model with exceptions [25]. Their specification logic is left abstract, thus they provide no automated mechanism for expressing or deciding assertions. Also, the systems in both of these cited works are developed in functional, not Object Oriented, language models.

The systems in [8, 9, 14, 26] use LTSs extracted from control-flow graph abstractions to model-check program security properties expressed in temporal logic. Their approach is close in several respects, but we are primarily focused on the programming language as opposed to the model-checking side of the problem. Their analyses assume the pre-existence of a control-flow graph abstraction, which is in the format for a first-order program analysis only. Our type-based approach is defined directly at the language level, and type inference provides an explicit, scalable mechanism for extracting an abstract program interpretation, which is applicable to Object Oriented features. Furthermore, polymorphic effects are inferable in our system and events may be parameterized by constants so partial dataflow

information can be included. We believe our results are critical to bringing this general approach to practical fruition for production programming languages such as ML and Java [35, 38].

Another important related work is [21], where a type system for static enforcement of stack inspection is developed for Java bytecode. In particular, they address issues related to dynamic dispatch and linking that are similar to those discussed in Sect. 2, although our system treats a more general class of program properties than stack inspection. Also, their system essentially relies on the "join of all effects" approach discussed in Sect. 2 for dynamically dispatched method typings, whereas we develop a more accurate solution based on parametric polymorphism.

Some recent work [2, 28] has focused on analyzing patterns of method invocations for model checking safety properties of Object Oriented programs, although traces in these works are represented as regular expressions, not LTSs. While there are some similarities in their approaches and applications, we believe that our system is the first to consider the extension of trace effects *per se* to Object Oriented programs, as a technique for statically verifying assertions in a general event trace program logic.

### 8.2 Summary

We have defined the language $\mathrm{FJ}_{\mathrm{trace}}$, a version of Featherweight Java (FJ) extended with event traces and checks, which are local assertions imposing well-formedness properties on traces. This provides a foundation for a general logic of trace-based program properties in Object Oriented languages such as Java. We have defined a static type and effect analysis that automatically generates conservative approximations of $\mathrm{FJ}_{\mathrm{trace}}$ program trace behavior, called trace effects. Trace effects are endowed with a label transition system semantics, and are therefore amenable to model checking for static verification of asserted trace-based properties. The analysis is sound, in that static verification of program trace effects guarantees success of dynamic checks.

The Object Oriented paradigm presents several challenges to trace effect analysis, including complications due to inheritance, method override, and dynamic dispatch. In particular, we have observed that different versions of methods in a given class hierarchy should not be required to agree in their trace effects, since this requirement would be overly restrictive. We have proposed a particular application of parametric polymorphism to promote flexibility in the presence of dynamic dispatch. We have also shown that a novel definition of subtyping constraints in a regular tree model can be used for flexibility in application to Object Oriented program features, including dynamically checked downcasts.

In addition to a basic Object Oriented model based on FJ, we have considered extensions including exceptions and stack-based security contexts. Transformations of inferred trace effects were defined, that were demonstrated to faithfully reflect the behavior of these extensions. This provides additional evidence that trace effects are scalable static representations of program trace behavior, well suited to a general purpose Object Oriented model. Interesting topics for future work, that would extend our analysis to a realistic (vs. idealized) Object Oriented language model, include mutation and state, dynamic linking, and bytecode-level type systems.

**Acknowledgements** The author would like to thank David Van Horn, Scott Smith, and anonymous referees for their helpful comments on drafts of this paper. This research was supported by AFOSR grant number USAF 9550-06-1-0313.

## References


1. Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03) (2003)
2. Alur, R., Cerny, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 98–109. ACM Press (2005)
3. Amtoft, T., Nielson, F., Nielson, H.R.: Type and Effect Systems. Imperial College Press (1999)
4. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: SPIN, pp. 113–130 (2000)
5. Bartoletti, M., Degano, P., Ferrari, G.L.: Enforcing secure service composition. In: CSFW, pp. 211–223. IEEE Computer Society (2005)
6. Bartoletti, M., Degano, P., Ferrari, G.L.: History-based access control with local policies. In: Sassone, V. (ed.) FoSSaCS. Lecture Notes in Computer Science, vol. 3441, pp. 316–332. Springer, Berlin (2005)
7. Bartoletti, M., Degano, P., Ferrari, G.L.: Policy framings for access control. In: WITS '05: Proceedings of the 2005 Workshop on Issues in the Theory of Security, pp. 5–11. ACM Press (2005)
8. Besson, F., Jensen, T., Le Métayer, D., Thorn, T.: Model checking security properties of control flow graphs. J. Comput. Secur. **9**, 217–250 (2001)
9. Besson, F., de Grenier de Latour, T., Jensen, T.: Secure calling contexts for stack inspection. In: Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02), pp. 76–87. ACM Press (2002)
10. Bruce, K.B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S.F., Trifonov, V., Leavens, G.T., Pierce, B.C.: On binary methods. Theory Pract. Object Syst. **1**(3), 221–242 (1995)
11. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures. In: Smolka, S., Bergstra, J., Pons, A. (eds.) Handbook on Process Algebra. North-Holland, Amsterdam (2001)
12. Cartwright, R., Fagan, M.: Soft typing. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, pp. 278–292. ACM Press (1991)
13. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, pp. 235–244, Washington, DC, November 18–22, 2002
14. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 54–66 (2000)
15. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 207–212 (1982)
16. Edjlali, G., Acharya, A., Chaudhary, V.: History-based access control for mobile code. In: ACM Conference on Computer and Communications Security, pp. 38–48 (1998)
17. Eifrig, J., Smith, S., Trifonov, V.: Type inference for recursively constrained types and its application to OOP. In: Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science, vol. 1. Elsevier Science, Amsterdam (1995)
18. Esparza, J., Kucera, A., Schwoon, S.: Model-checking LTL with regular valuations for pushdown systems. In: TACS: 4th International Conference on Theoretical Aspects of Computer Software (2001)
19. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1–12, Berlin, Germany, June 2002
20. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In: USENIX Symposium on Internet Technologies and Systems, pp. 103–112, Monterey, CA, December 1997
21. Higuchi, T., Ohori, A.: A static type system for JVM access control. ACM Trans. Program. Lang. Syst. **29**(1) (2007)
22. Holzmann, G.J., Smith, M.H.: Software model checking: extracting verification models from source code. Softw. Test. Verif. Reliab. **11**(2), 65–79 (2001)
23. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 331–342, Portland, Oregon, January 2002
24. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001)
25. Iwama, F., Igarashi, A., Kobayashi, N.: Resource usage analysis for a functional language with exceptions. In: PEPM '06: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 38–47. ACM Press, New York (2006)
26. Jensen, T., Le Métayer, D., Thorn, T.: Verification of control flow based security properties. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy (1999)

27. Stuckey, P.J., Marriott, K., Sulzmann, M.: Resource usage verification. In: Proc. of First Asian Programming Languages Symposium, APLAS 2003 (2003)
28. Logozzo, F.: Separate compositional analysis of class-based object-oriented languages. In: Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology (AMAST'2004). Lectures Notes in Computer Science, vol. 3116, pp. 332–346. Springer, Berlin (2004)
29. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03), Uppsala, Sweden, August 2003
30. Nagata, A., Kobayashi, N., Yonezawa, A.: Region-based memory management for a dynamically-typed language. In: Asian Programming Languages Symposium. Lecture Notes in Computer Science. Springer, Berlin (2004)
31. Palsberg, J., O'Keefe, P.: A type system equivalent to flow analysis. In: POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 367–378. ACM Press, New York (1995)
32. Palsberg, J., Smith, S.: Constrained types and their expressiveness. ACM Trans. Program. Lang. Syst. **18**(5), 519–527 (1996)
33. Schmidt, D.A.: Trace-based abstract interpretation of operational semantics. Lisp Symb. Comput. **10**(3), 237–271 (1998)
34. Schneider, F.B.: Enforceable security policies. Inf. Syst. Secur. **3**(1), 30–50 (2000)
35. Skalka, C.: Trace effects and object orientation. In: PPDP '05: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pp. 139–150. ACM Press, New York (2005)
36. Skalka, C., Smith, S.: Static enforcement of security with types. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), pp. 34–45, Montréal, Canada, September 2000
37. Skalka, C., Smith, S.: History effects and verification. In: Asian Programming Languages Symposium. Lecture Notes in Computer Science, vol. 3302. Springer, Berlin (2004)
38. Skalka, C., Smith, S., Van Horn, D.: A type and effect system for flexible abstract interpretation of Java. In: Proceedings of the ACM Workshop on Abstract Interpretation of Object Oriented Languages. Electronic Notes in Theoretical Computer Science, January 2005
39. Skalka, C., Smith, S., Van Horn, D.: Types and trace effects of higher order programs. J. Funct. Program. **18**(2), 179–249 (2008)
40. Steffen, B., Burkart, O.: Model checking for context-free processes. In: CONCUR'92, Stony Brook (NY). Lecture Notes in Computer Science, vol. 630, pp. 123–137. Springer, Heidelberg (1992)
41. Talpin, J.-P., Jouvelot, P.: The type and effect discipline. In: Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California, pp. 162–173. IEEE Computer Society Press, Los Alamitos (1992)
42. Tofte, M., Talpin, J.-P.: Region-based memory management. Inf. Comput. **132**(2), 109–176 (1997)
43. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N.: A retrospective on region-based memory management. High. Order Symb. Comput. **17**(3), 245–265 (2004)
44. Trifonov, V., Smith, S.: Subtyping constrained types. In: Proceedings of the Third International Static Analysis Symposium, vol. 1145, pp. 349–365. Springer, Berlin (1996)
45. Walker, D.: A type system for expressive security policies. In: Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 254–267, Boston, Massachusetts, January 2000
46. Wallach, D.S., Felten, E.: Understanding Java stack inspection. In: Proceedings of the 1998 IEEE Symposium on Security and Privacy, May 1998