Trace Effects and Object Orientation

Christian Skalka
Department of Computer Science
University of Vermont
skalka@cs.uvm.edu

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 an Object Oriented setting.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Language Classifications—*Object oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*

General Terms

Security, Languages, Theory, Verification.

Keywords

Type and effect, type constraints, temporal program logic.

1. INTRODUCTION

Program type analysis and model checking have a shared goal: to statically enforce properties of programs. A number of authors have recently observed [24, 20, 17] that these two approaches can play complementary roles in the verification of general *trace based* program properties, expressed in temporal logics: type systems can be used to compute program abstractions, which can in turn be used as inputs to model checking [26]. In other words, type analysis can serve as a technique for model extraction [16] in this setting.

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 [24, 20, 5] have demonstrated that static approximations of program event traces can be generated by type and and effect analyses [27, 3], in a form amenable to existing model-checking techniques for verification. We call these approximations *trace effects*.

Related trace based analyses have been shown capable of statically enforcing flow-sensitive security properties such as safe locking behavior [15], and resource usage policies such as file usage protocols and memory management [20, 17]. The history-based access control model of [1] can be implemented with event traces and checks [24], as can be the policies realizable in that model, e.g. sophisticated Chinese Wall policies [1]. Also, in previous work we have shown how trace effects can be post-processed to statically verify stack-based checks [24, 25] such as stack inspection. In short, the combination of a primitive notion of program events, and a temporal program logic for asserting properties of event traces, comprises a powerful and general tool for enforcing program properties. Type and effect inference provides an expressive and adaptable technique for trace effect model extraction. The automated character and generality of the overall approach provides programmers with a flexible tool for specifying 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 foundational technical considerations for application of trace effects to Object Oriented languages, particularly Java. As discussed in Sect. 2, inheritance, dynamic dispatch, and downcasts present significant challenges to trace effect analysis. To isolate these issues, we extend Featherweight Java (FJ)[18] with events, traces, and checks, and a polymorphic type and effect inference analysis for static enforcement of checks, yielding the language FJ_{sec}. We demonstrate that the combination of parametric polymorphism, subtyping, and type constraints can be used to obtain a flexible and sound trace effect analysis in an Object Oriented setting.

1.1 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. In Sect. 3, the FJ_{sec} 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 FJ_{sec} is presented and discussed, along with a type safety result. A type inference algorithm is defined in Sect. 6,

PPDP'05, July 11–13, 2005, Lisbon, Portugal.

that is shown to be sound with respect to the logical type system, implying type safety in the implementation. We conclude with a discussion of related work and a final summary in Sect. 7.

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. We also propose a *type constraint* representation; along with known benefits of this approach in application to Object Oriented programming [14, 8], 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 [24, 1], where code is statically signed by its owner (an authorization event), and a local access control list $\mathcal{A}$ associates owners with their authorizations. A `demand` predicate ensures that the intersection of all authorizations 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 $\mathtt{demand}(\mathtt{r})$. Note that standard practice allows class owners to extend classes owned by others, i.e. ownership need not be consistent throughout an inheritance hierarchy. In our encoding, distinct ownership implies method override, at least with regard to authorization events; an accurate analysis therefore requires independence of trace effects of different method versions in the inheritance hierarchy.

The issue is complicated further by dynamic dispatch. 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 `safewrite` method as:

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

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 `Formatter`s 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*. A standard approach [24, 3, 20] is to approximate 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 `Applet`s are not `FileWrite` authorized, verification of:

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

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 re-computation of its effects. This would disallow modularity.

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 type language of FJ_{sec} 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 FJ_{sec} 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`:

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

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

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

Figure 1: FJ$_{\text{sec}}$ language syntax

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 [24], 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 [14, 8].

A constraint type representation also supports a soft typing analysis of downcasts, which combines static and dynamic checks to ensure soundness [10]. 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` and `S` are the field and method types of the `Triangle` and `Polygon` objects, respectively:

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

Then, given the cast $(\mathtt{Triangle})\mathtt{e}$, 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 [18]. 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 $(\mathtt{Triangle})\mathtt{e}$, 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 requires only that if $[\mathtt{U\,C}] <: \mathtt{T}$, then $[\mathtt{U\,C}] <: [\mathtt{T}'\ \mathtt{Triangle}]$ if $\mathtt{C} <: \mathtt{Triangle}$; see Definition 3 for a formalization of the idea. This implies:

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

and *not* $[\mathtt{S\ Polygon}] <: [\mathtt{T}'\ \mathtt{Triangle}]$. Note that requiring the latter would yield an inconsistent constraint set in any case, so a benefit of this approach is completeness in the presence of dynamically checked downcasts. 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.

$$\mathtt{C} <: \mathtt{C} \qquad\qquad \frac{\mathtt{B} <: \mathtt{C} \qquad \mathtt{C} <: \mathtt{D}}{\mathtt{B} <: \mathtt{D}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}$$

Figure 2: Nominal subtyping for FJ

3. THE LANGUAGE FJ$_{\text{sec}}$

In this section we define the syntax and semantics of FJ$_{\text{sec}}$, which comprises FJ extended with primitive features for specifying events and local checks, as well as the notion of a run-time event trace.

3.1 Syntax

The syntax of FJ$_{\text{sec}}$ is defined in Fig. 1. It is the same as the syntax of FJ, where $\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}$ ranges over class names, `x` ranges over variables, `f` ranges over field names, and `m` ranges over method names. *Values*, denoted `v` or `u`, are objects, i.e. expressions of the form $\mathtt{new\ C}(\mathtt{v}_1, \ldots, \mathtt{v}_n)$. 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 [18] we assume `Object` values that have no fields or methods; we let $()$ denote the value $\mathtt{new\ Object}()$. The essential distinguishing features of FJ$_{\text{sec}}$ are *events* $\mathtt{ev}[\mathtt{i}]$ and *local checks* $\mathtt{chk}[\mathtt{i}]$. Both events and checks are distinguished by labels `i`. Events and checks encountered during execution are accrued in linear order in *traces* η, 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 $\eta \vdash \mathtt{chk}[\mathtt{i}]$ to denote that η satisfies $\mathtt{chk}[\mathtt{i}]$. We could for example instantiate the language of checks with the linear mu-calculus, as in [24], but presently we are mainly concerned with the typing aspect of the analysis.

3.1.1 Vector Notations

For brevity in numerous instances, we adopt the vector notations of [18]. 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 $\varnothing$, we use a comma as a sequence concatenation operator, 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 $\mathtt{m} \in \bar{\mathtt{m}}$. 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. The notation $\mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};$ abbreviates the sequence of

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

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

$$\text{R-CAST}\quad \frac{\mathtt{C} <: \mathtt{D}}{\eta, (\mathtt{D})(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})) \to \eta, \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})} \qquad \text{R-EVENT}\quad \eta, \mathtt{ev}[\mathtt{i}] \to \eta; \mathtt{ev}[\mathtt{i}], ()$$

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

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

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

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

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

Figure 3: FJ$_{\text{sec}}$ operational semantics

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 FJ$_{\text{sec}}$ are defined in Fig. 3. The small-step reduction relation $\to$ is defined on closed *configurations*, which are pairs of traces and expressions $\eta, \mathtt{e}$. As in [18], we divide the operational rules into *computation* and *congruence* rules; the former (resp. latter) are those whose names are prefixed by R- (resp. RC-). The semantics are defined in terms of a number of auxiliary functions and a nominal subtyping relation $<:$ taken from [18] and recalled in Fig. 2 and Fig. 5. 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 rules that directly affect the trace include R-EVENT, which appends an event $\mathtt{ev}[\mathtt{i}]$ encountered during execution to the end of the trace. The R-CHECK rule is defined similarly, except the check $\mathtt{chk}[\mathtt{i}]$ 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 $\mathtt{H}$ conservatively approximate traces η that may develop during execution, by representing a set of traces containing at least η. The grammar of trace effects is given in Fig. 6. A trace effect may be an event $\mathtt{ev}[\mathtt{i}]$ or check $\mathtt{chk}[\mathtt{i}]$, 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 μ-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 η is the same as linear, variable-free trace effects, we abuse syntax and let η 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 include a $\downarrow$ symbol to denote termination. Abstract traces may be infinite, because programs may not terminate. The interpretation is defined via strings denoted θ, over the following alphabet:

$$a ::= \mathtt{ev}[\mathtt{i}] \mid \mathtt{chk}[\mathtt{i}] \mid \epsilon \mid \downarrow$$

The interpretation of an effect $\mathtt{H}$, denoted $[\![\mathtt{H}]\!]$, is then taken to be the prefix-closed, finite approximation of the trace sets that may be generated by $\mathtt{H}$, when viewed as a program in a transition semantics defined by relations $\xrightarrow{a}$; this interpretation is specified in Fig. 4. Trace effect equivalence is defined via this interpretation, i.e. $\mathtt{H}_1 = \mathtt{H}_2$ iff $[\![\mathtt{H}_1]\!] = [\![\mathtt{H}_2]\!]$. This relation is in fact undecidable: trace effects are equivalent to BPA's (basic process algebras) [24], and their trace equivalence is known to be undecidable [9].

We then base the notion of validity of trace effects on the 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. $\mathtt{H}$ *is* valid *iff for all* $(a_1 \cdots a_n \mathtt{chk}[\mathtt{i}]) \in [\![\mathtt{H}]\!]$ *it is the case that* $a_1; \ldots; a_n \vdash \mathtt{chk}[\mathtt{i}]$ *holds.*

$$\mathtt{ev}[\mathtt{i}] \xrightarrow{\mathtt{ev}[\mathtt{i}]} \epsilon \qquad \mathtt{chk}[\mathtt{i}] \xrightarrow{\mathtt{chk}[\mathtt{i}]} \epsilon \qquad \epsilon; \mathtt{H} \xrightarrow{\epsilon} \mathtt{H}$$

$$\mathtt{H}_1; \mathtt{H}_2 \xrightarrow{a} \mathtt{H}_1'; \mathtt{H}_2 \ \text{ if } \ \mathtt{H}_1 \xrightarrow{a} \mathtt{H}_1' \qquad \mathtt{H}_1|\mathtt{H}_2 \xrightarrow{\epsilon} \mathtt{H}_1 \qquad \mathtt{H}_1|\mathtt{H}_2 \xrightarrow{\epsilon} \mathtt{H}_2$$

$$\mu\mathtt{h}.\mathtt{H} \xrightarrow{\epsilon} \mathtt{H}[\mu\mathtt{h}.\mathtt{H}/\mathtt{h}]$$

$$[\![\mathtt{H}]\!] = \begin{array}{c} \{a_1 \cdots a_n \mid \mathtt{H} \xrightarrow{a_1} \cdots \xrightarrow{a_n} \mathtt{H}'\} \\ \cup \\ \{a_1 \cdots a_n \downarrow \mid \mathtt{H} \xrightarrow{a_1} \cdots \xrightarrow{a_n} \epsilon\} \end{array}$$

Figure 4: Interpretation of trace effects

5. TYPES FOR FJ$_{\text{sec}}$

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. The system is algorithmically checkable, and enjoys a type safety result [18]. 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 FJ$_{\text{sec}}$. 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.

$$
\begin{array}{rcll}
\mathtt{H} & ::= & \epsilon \mid \mathtt{ev[i]} \mid \mathtt{chk[i]} \mid \mathtt{h} \mid \mathtt{H;H} \mid \mathtt{H|H} \mid \mu\mathtt{h.H} & \textit{trace effects} \\
\mathtt{T} & ::= & \mathtt{H} \mid \mathtt{X} \mid \bar{\mathtt{T}} \mid [\mathtt{T\,C}] \mid \mathtt{x:T} \mid \mathtt{T} \xrightarrow{\mathtt{H}} \mathtt{T} & \textit{types} \\
\mathtt{X} & ::= & \mathtt{h} \mid \mathtt{t} & \textit{type variables} \\
C & ::= & \mathtt{T} <: \mathtt{T} \mid \mathtt{T} \lessdot \mathtt{T} \mid C \wedge C \mid \mathbf{true} & \textit{constraints}
\end{array}
$$

Figure 6: FJ$_{\text{sec}}$ type and constraint syntax

$$
fields(\mathtt{Object}) = \varnothing
$$

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

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

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

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

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

Figure 5: Auxiliary functions

$$
\mathtt{t} : \mathit{Body}_{\mathtt{C}} \qquad \frac{fields(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{f}} \qquad \bar{\mathtt{S}} : \mathit{Body}_{\bar{\mathtt{D}}} \qquad \bar{\mathtt{T}} : \mathit{Meth}_{\bar{\mathtt{m}},\mathtt{C}}}{(\bar{\mathtt{f}} : [\bar{\mathtt{S}}\,\bar{\mathtt{D}}]\ \bar{\mathtt{m}} : \bar{\mathtt{T}}) : \mathit{Body}_{\mathtt{C}}}
$$

$$
\frac{\mathtt{T} : \mathit{Body}_{\mathtt{C}}}{[\mathtt{T\,C}] : \mathit{Type}}
$$

$$
\frac{\begin{array}{c} mtype(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{D}} \to \mathtt{D} \qquad \bar{\mathtt{T}} : \mathit{Body}_{\bar{\mathtt{D}}} \\ \mathtt{T} : \mathit{Body}_{\mathtt{D}} \qquad CT(\mathtt{C}) = \mathtt{class\ C\ extends\ B}\ \{\ldots\} \\ \text{if } mtype(\mathtt{m}, \mathtt{B}) = \bar{\mathtt{E}} \to \mathtt{E} \text{ then } \bar{\mathtt{E}} = \bar{\mathtt{D}} \text{ and } \mathtt{E} = \mathtt{D} \end{array}}{[\bar{\mathtt{T}}\,\bar{\mathtt{D}}] \xrightarrow{\mathtt{H}} [\mathtt{T\,D}] : \mathit{Meth}_{\mathtt{m},\mathtt{C}}}
$$

Figure 7: FJ$_{\text{sec}}$ type kinding rules

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 binary methods [14, 8]. 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 [10]. Furthermore, as has been observed frequently in previous related type and effect analyses [24, 17, 20], 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 [14]. 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* [24], 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 FJ$_{\text{sec}}$ is given in Fig. 6. We immediately endow types with kinding rules; hereafter we implicitly consider only well-kinded types. The kinding rules refer to auxiliary functions borrowed from [18] and defined in Fig. 5. Our type term language incorporates the nominal type language of [18]. Types $[\mathtt{T\,C}]$ of kind *Type* are object types, where C is the declared name of the object's class and T denotes the object's instance and method types. The class name component of the type fixes the instance and methods appearing in the type to conform to those in the declare class. This is enforced by requiring that T be of kind $\mathit{Body}_{\mathtt{C}}$. Note also that in any object type, the type of its methods and fields must match their declared signatures, since the $\mathit{Meth}_{\mathtt{m},\mathtt{C}}$ kinding rule imposes this form on the type of a method m in class C. Abusing notation, we let x range over field and method names, as well as variables, and write $\mathtt{x} : \mathtt{T}$ to associate the type T with the field or method x.

The kinding rule for method types also deserves attention for its contribution to the independence of inheritance and effects in the analysis. Recalling that the type system of [18] 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 types of kind $\mathit{Meth}_{\mathtt{m},\mathtt{C}}$ imposes this restriction, but only on the declared class name component of the type. The inferred effect component H, on the other hand, is not restricted to relate to superclass method effects in any way.

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 meaning of constraints is defined via interpretation in a regular tree model, defined and discussed in Sect. 5.2.

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 $[\bar{\mathtt{T}}\,\bar{\mathtt{C}}]$ 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 $\mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_n : \mathtt{T}_n$. We abbreviate constraints using vector notation, writing $\bar{\mathtt{S}} <: \bar{\mathtt{T}}$ for $\mathtt{S}_1 <: \mathtt{T}_1 \wedge \cdots \wedge \mathtt{S}_n <: \mathtt{T}_n$. Vector notation is similarly extended to regular tree syntax, defined in the next section.

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

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{D}} \to \mathtt{D} \qquad \varsigma = \mathit{Body}_{\bar{\mathtt{D}}}, \mathit{Eff}, \mathit{Body}_{\mathtt{D}}}{[\cdot\ \bar{\mathtt{D}}] \dot{\to} [\cdot\ \mathtt{D}]^{\varsigma} : \mathit{Meth}_{\mathtt{m},\mathtt{C}}}$$

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

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

Figure 8: Regular tree ranked alphabet kinding rules

$$\frac{\varphi \text{ is finite}}{\bot \preccurlyeq_{\text{fin}} \varphi} \qquad \frac{\varphi \text{ is finite}}{\varphi \preccurlyeq_{\text{fin}} \top} \qquad \frac{[\![\mathtt{H}]\!] \subseteq [\![\mathtt{H}']\!]}{\mathtt{H} \preccurlyeq_{\text{fin}} \mathtt{H}'}$$

$$\frac{\varphi_2 \preccurlyeq_{\text{fin}} \varphi_2' \qquad \mathtt{H} \preccurlyeq \mathtt{H}' \qquad \bar{\varphi_1'} \preccurlyeq_{\text{fin}} \bar{\varphi_1} \qquad \varphi_1, \varphi_1', \varphi_2, \varphi_2' \text{ finite}}{\bar{\varphi_1} \xrightarrow{\mathtt{H}} \varphi_2 \preccurlyeq_{\text{fin}} \bar{\varphi_1'} \xrightarrow{\mathtt{H}} \varphi_2'}$$

$$\frac{\bar{\varphi_1} \preccurlyeq_{\text{fin}} \bar{\varphi_1'} \qquad \bar{\varphi_1}, \bar{\varphi_1'} \text{ finite}}{\bar{\mathtt{x}} : \bar{\varphi_1}, \bar{\mathtt{y}} : \bar{\varphi_2} \preccurlyeq_{\text{fin}} \bar{\mathtt{x}} : \bar{\varphi_1'}}$$

$$\frac{\varphi \preccurlyeq_{\text{fin}} \varphi' \qquad \varphi, \varphi' \text{ finite} \qquad \mathtt{C} <: \mathtt{D}}{[\varphi\ \mathtt{C}] \preccurlyeq_{\text{fin}} [\varphi'\ \mathtt{D}]}$$

$$\frac{\varphi|_n \preccurlyeq_{\text{fin}} \varphi'|_n \text{ for all } n \in \mathbb{N}}{\varphi \preccurlyeq \varphi'}$$

Figure 9: Primitive subtyping for regular trees

Vector notations are also used to abbreviate the kinding rules in Fig. 6 and Fig. 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 $\mathit{Meth}_{\mathtt{m}_1,\mathtt{C}}, \ldots, \mathit{Meth}_{\mathtt{m}_n,\mathtt{C}}$. Also, in kinding judgements, we write $\bar{\mathtt{T}} : \mathit{Body}_{\bar{\mathtt{C}}}$ for a possibly empty sequence of judgements $\mathtt{T}_1 : \mathit{Body}_{\mathtt{C}_1}, \ldots, \mathtt{T}_n : \mathit{Body}_{\mathtt{C}_n}$, with $\bar{\mathtt{T}} : \mathit{Meth}_{\bar{\mathtt{m}},\mathtt{C}}$ similarly defined.

5.2 Regular Tree Model and Subtyping

To accommodate recursive constraints, we define subtyping in FJ_{sec} via primitive subtyping in a regular tree model, using techniques adapted from [28]. In our model, function type nodes in regular trees are labeled with trace effects, and rather than being constructed from ranked alphabets as in [28] and elsewhere, our regular trees are constructed from kinded alphabets, imposing well-formedness of trees. Trace effect labelings of regular trees require an extension of the primitive subtyping relation to trace effects, which is based on set containment of effect interpretations.

DEFINITION 2 (REGULAR TREE MODEL). *Let the* tree constructor kinds *be defined as:*

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

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

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

where each element of the alphabet is indexed by a signature, written c^{ς}*, and must be* well-kinded *according to the rules given in Fig. 8.*

A tree φ *is a partial function from finite sequences (*paths*)* π *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* $\varphi(\pi n) : \varsigma(n)$*. 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* φ *with* $\top$*. Then,* $\preccurlyeq_{\mathit{fin}}$ *is the partial order over finite* $\varphi \in \mathbb{T}$ *axiomatized in Fig. 9, and* $\preccurlyeq$ *is the partial order over* $\mathbb{T}$ *approximated by* $\preccurlyeq_{\mathit{fin}}$ *axiomatized in Fig. 9.*

$$\begin{aligned}
\rho(\mathtt{h}, hs) &= \rho(\mathtt{h}) && \mathtt{h} \notin hs \\
\rho(\mathtt{h}, hs) &= \mathtt{h} && \mathtt{h} \in hs \\
\rho(\mathtt{ev}[\mathtt{i}], hs) &= \mathtt{ev}[\mathtt{i}] \\
\rho(\mathtt{chk}[\mathtt{i}], hs) &= \mathtt{chk}[\mathtt{i}] \\
\rho(\epsilon, hs) &= \epsilon \\
\rho(\mathtt{H}_1; \mathtt{H}_2, hs) &= \rho(\mathtt{H}_1, hs); \rho(\mathtt{H}_2, hs) \\
\rho(\mathtt{H}_1 | \mathtt{H}_2, hs) &= \rho(\mathtt{H}_1, hs) | \rho(\mathtt{H}_2, hs) \\
\rho(\mu\mathtt{h}.\mathtt{H}, hs) &= \mu\mathtt{h}.\rho(\mathtt{H}, hs \cup \{h\}) \\
\\
\rho([\mathtt{T}\ \mathtt{C}]) &= [\rho(\mathtt{T})\ \mathtt{C}] \\
\rho(\mathtt{x} : \mathtt{T}) &= \mathtt{x} : \rho(\mathtt{T}) \\
\rho(\bar{\mathtt{T}} \xrightarrow{\mathtt{H}} \mathtt{T}) &= \rho(\bar{\mathtt{T}}) \xrightarrow{\rho(\mathtt{H}, \varnothing)} \rho(\mathtt{T}) \\
\rho(\mathtt{T}, \bar{\mathtt{T}}) &= \rho(\mathtt{T}), \rho(\bar{\mathtt{T}}) \\
\rho(\varnothing) &= \varnothing
\end{aligned}$$

Figure 10: Interpretations extended to types and effects

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 ρ *are total mappings from type variables* $\mathtt{X}$ *to* $\mathbb{T}$*. Interpretations are extended to types and effects as in Fig. 10; the interpretation of effects are parameterized by sets* hs *of effect variables, to prevent substitution of* μ*-bound variables. The relation* $\rho \vdash C$*, pronounced* ρ satisfies *or* solves C*, is axiomatized as follows:*

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

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

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

5.2.1 Discussion

The above defines a system of object width and depth subtyping, with method subtyping predicated on subsumption of trace effects. This extension, defined in Fig. 9, is based on containment of trace effect interpretations, reflecting a type soundness requirement that if $\mathtt{T}$ subsumes $\mathtt{S}$, it must also subsume $\mathtt{S}$'s trace effects. Since constraints are defined on the basis of interpretations, constraints on types with abstract components are meaningful; for example, we could assert:

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

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

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

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

but not:

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

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 concerns about decidability of the analysis, since subtyping is predicated on trace effect containment, but equivalence of trace effects was shown to be undecidable in Sect. 4. Indeed, 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.

5.3 Logical Type Judgements and Properties

To define the logical type system, we introduce constrained type schemes $\forall\bar{\mathtt{X}}[C].\mathtt{T}$ and type environments Γ binding class names and variables to type schemes. If $\bar{\mathtt{X}} \cap \mathrm{fv}(\mathtt{T}) = \varnothing$ we abbreviate $\forall\bar{\mathtt{X}}[\mathbf{true}].\mathtt{T}$ 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 $\Gamma \vdash \mathtt{C} : \forall\bar{\mathtt{X}}[C].\mathtt{T}$ for classes. We say that a constraint, type pair $C, \mathtt{T}$ is *realizable* iff there exists a solution ρ of C such that $\rho(\mathtt{T})$ is well-kinded, and we impose the following sanity conditions on judgements: for any type scheme $\forall\bar{\mathtt{X}}[C].\mathtt{T}$ appearing in a judgement, we require that $C, \mathtt{T}$ be realizable, and 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. Type derivation rules are given in Fig. 11, with some auxiliary functions defined in Fig. 12. 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 $\mathtt{T}_i$, and if $n = 0$, then $\mathtt{H}_1, \ldots, \mathtt{H}_n$ and $C_1, \ldots, C_n$ mean ϵ and $\mathbf{true}$, respectively. We note that $\forall$-intro and -elim are located at class definition and object construction points, respectively. 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 $\mathtt{C}$. Any other objects will cause a dynamic cast check exception, and can therefore be ignored statically without compromising type safety, as in [18]. 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 [18], noting that we essentially follow their approach, and can easily adopt this distinction. We make the following definitions:

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

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

T-SEQ
$$\frac{|\mathtt{e}| = n \qquad \Gamma, C, \mathtt{H}_i \vdash \mathtt{e}_i : \mathtt{T}_i \text{ for all } \mathtt{e}_i \in \bar{\mathtt{e}}}{\Gamma, C, \mathtt{H}_1; \ldots; \mathtt{H}_n \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}}$$

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

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

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

T-CLASS
$$\frac{\Gamma; \mathtt{C} : [\mathtt{T}\,\mathtt{C}]; \mathtt{this} : [\mathtt{T}\,\mathtt{C}], C \vdash \mathtt{m}_i, \mathtt{C} : \mathtt{T}_i \text{ for all } \mathtt{m}_i \in \mathit{meths}(\mathtt{C})}{\Gamma \vdash \mathtt{C} : \forall\bar{\mathtt{X}}[C].[\mathtt{T}\,\mathtt{C}]}$$

Figure 11: FJ$_{\text{sec}}$ logical typing rules

DEFINITION 4. *The judgement $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ is* valid *iff it is derivable and there exists a solution ρ of C such that $\rho(\mathtt{H})$ is valid. An environment Γ is* well-formed *iff $\Gamma \vdash \mathtt{C} : \forall\mathtt{X}[C].\mathtt{T}$ is derivable for all $(\mathtt{C} : \forall\mathtt{X}[C].\mathtt{T}) \in \Gamma$.*

It is demonstrable that for closed, event- and check-free expressions e, there exists a derivable typing for e in FJ$_{\text{sec}}$ iff e is well-typed in the type system of [18]. Inspection of the rules reveals that our type system is defined by layering our type features over theirs, so the result is not surprising. For example, observe that sanity conditions on type judgements require that $\mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C})$ in the T-FIELD rule.

This means that our system absorbs properties of theirs, including type safety for the FJ subset of FJ$_{\text{sec}}$, implying that the only case we really need to consider, to extend type safety to FJ$_{\text{sec}}$, 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

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

$$\frac{\mathit{fields}(\mathtt{C}) = \mathtt{D}_1\ \mathtt{f}_1, \ldots, \mathtt{D}_n\ \mathtt{f}_n \quad \mathtt{f}_1 : \mathtt{T}_1 \in \mathtt{T}, \ldots, \mathtt{f}_n : \mathtt{T}_n \in \mathtt{T}}{\mathit{fieldsig}[\mathtt{T}\,\mathtt{C}] = \mathtt{T}_1, \ldots, \mathtt{T}_n} \qquad [\mathtt{t}\,\bar{\mathtt{C}}] \xrightarrow{\mathtt{h}} [\mathtt{t}\,\mathtt{C}] \text{ is abstract}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{f}}\ \bar{\mathtt{C}} \qquad \mathit{meths}(\mathtt{C}) = \bar{\mathtt{m}} \qquad \bar{\mathtt{T}} \text{ is abstract}}{[\bar{\mathtt{f}} : [\mathtt{t}\,\bar{\mathtt{C}}]\ \bar{\mathtt{m}} : \bar{\mathtt{T}}\ \mathtt{C}] \text{ is abstract}}$$

Figure 12: Auxiliary functions

preserved by computation. Note that the statement of subject reduction must apply to configurations; the proof, omitted here for brevity, follows by induction on derivations:

LEMMA 1 (SUBJECT REDUCTION). *If* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : [\mathtt{T}\,\mathtt{C}]$ *is derivable for closed* e *and well-formed* Γ*, and* $\eta, \mathtt{e} \rightarrow \eta', \mathtt{e}'$*, then* $\Gamma, C, \mathtt{H}' \vdash \mathtt{e}' : [\mathtt{T}\,\mathtt{C}]$ *is derivable with* $C \Vdash \eta'; \mathtt{H}' <: \eta; \mathtt{H}$.

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* Γ*, and* $\epsilon, \mathtt{e} \rightarrow \eta, \mathtt{e}'$*, then* $C \Vdash \eta <: \mathtt{H}$.

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

DEFINITION 5. Evaluation contexts *are defined as follows:*

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

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

LEMMA 3 (STATIC ENFORCEMENT OF CHECKS). *Given closed* e *and well-formed* Γ*, if the judgement* $\Gamma, C, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is valid , and* $\epsilon, \mathtt{e} \rightarrow^\star \eta, \mathtt{E}[\mathtt{chk}[\mathtt{x}]]$*, then* $\eta \vdash \mathtt{chk}[\mathtt{x}]$.

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

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 $\mathtt{e}; \mathtt{e}$ and a lexically scoped name-to-value binding construct $\mathtt{C\ x} = \mathtt{new\ C}()$. 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:

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

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

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

where PTmplT and BdoorT are defined as above. By the T-CAST typing rule and properties of soft subtyping discussed above, we therefore may assert (as a somewhat contrived example; a Backdoor cast is not likely to occur in practice):

$$\Gamma, C, \epsilon \vdash (\texttt{Backdoor})\mathtt{e} : \mathtt{BdoorT}$$

meaning that the following:

```
Writer w = new Writer();
w.safewrite((Backdoor)e, f);
```

may be assigned the following effect:

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

6. TYPE INFERENCE FOR $\mathrm{FJ}_{\mathrm{sec}}$

We now give an implementation of the $\mathrm{FJ}_{\mathrm{sec}}$ type analysis. This includes a type inference algorithm and constraint closure rules for checking satisfiability of typing judgements. 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 [24]. Model checking is applicable, since we endow trace effects with an LTS semantics (Sect. 4), for which a variety of model checking techniques exist [9]. However, standard techniques expect term, rather than constraint, representation of LTSs. Therefore, it is necessary to define a means of extracting a unified trace effect representation from inferred typing judgements. For this purpose we define the *hextract* algorithm, which obtains a unified representation of trace effects from the inferred constraint representation.

Type inference rules are given in Fig. 13; the W subscripting the relation $\vdash_W$ distinguishes type inference from logical typing judgements, and is meant to evoke the prototypical polymorphic type reconstruction algorithm W [13]. Note that all formal method parameters are assigned abstract effects when typing the method body, via the definition of "is abstract" given in Fig. 12. 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. In our formal characterization, we will assume that programs are typed in class typing environments Γ that have been generated by previous inference, as in the following definition:

DEFINITION 6. *Let:*

$$\Gamma = \mathtt{C}_1 : \forall\bar{\mathtt{X}}_1[C_1].\mathtt{T}_1; \ldots; \mathtt{C}_n : \forall\bar{\mathtt{X}}_n[C_n].\mathtt{T}_n$$

Then Γ *is* inferable *iff for all* $0 < i \leq n$ *it is the case that:*

$$\mathtt{C}_1 : \forall\bar{\mathtt{X}}_1[C_1].\mathtt{T}_1; \ldots; \mathtt{C}_{i-1} : \forall\bar{\mathtt{X}}_{i-1}[C_{i-1}].\mathtt{T}_{i-1} \vdash_W \mathtt{C}_i : \Gamma(\mathtt{C}_i)$$

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

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

T-SEQ
$$\frac{|\mathtt{e}| = n \qquad \Gamma, C_i, \mathtt{H}_i \vdash_W \mathtt{e}_i : \mathtt{T}_i \text{ for all } \mathtt{e}_i \in \bar{\mathtt{e}}}{\Gamma, C_1 \wedge \cdots \wedge C_n, \mathtt{H}_1; \ldots; \mathtt{H}_n \vdash_W \bar{\mathtt{e}} : \bar{\mathtt{T}}}$$

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

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

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

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

T-CAST
$$\frac{\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}}{\Gamma, C \wedge \mathtt{T} \lessdot [\mathtt{t}\,\mathtt{D}], \mathtt{H} \vdash_W (\mathtt{D})\mathtt{e} : [\mathtt{t}\,\mathtt{D}]}$$

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

T-CLASS
$$\frac{\mathtt{T} = [\bar{\mathtt{f}} : \bar{\mathtt{R}}\,\bar{\mathtt{m}} : \bar{\mathtt{S}}\,\mathtt{C}] \text{ is abstract and well-kinded} \quad \Gamma; \mathtt{C} : \mathtt{T}; \mathtt{this} : \mathtt{T}, C_i \vdash_W \mathtt{m}_i, \mathtt{C} : \mathtt{T}_i \text{ for all } \mathtt{m}_i \in \bar{\mathtt{m}} \qquad |\mathtt{m}| = n}{\Gamma \vdash_W \mathtt{C} : \forall\bar{\mathtt{X}}[C_1 \wedge \cdots \wedge C_n].[\bar{\mathtt{f}} : \bar{\mathtt{R}}\,\bar{\mathtt{m}} : \bar{\mathtt{T}}\,\mathtt{C}]}$$

Figure 13: $\mathrm{FJ}_{\mathrm{sec}}$ type inference rules

Note that for simplicity in this presentation we disallow mutually recursive class definitions, though this restriction can be easily lifted by modifying the above definition. 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 nodes in inference derivations:

LEMMA 4 (SOUNDNESS OF INFERENCE). *If* $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ *is derivable with* $C \wedge D, \mathtt{T}$ *realizable, then* $\Gamma, C \wedge D, \mathtt{H} \vdash \mathtt{e} : \mathtt{T}$ *is derivable.*

In addition to type inference rules, the type implementation comprises a constraint closure algorithm and consistency check. We

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

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

Figure 14: Constraint closure rules

$$\vdash \mathbf{true} : ok \qquad \frac{\vdash C : ok \qquad \vdash D : ok}{\vdash C \wedge D : ok} \qquad \vdash \mathtt{H} <: \mathtt{H}' : ok$$

$$\frac{\mathtt{C} <: \mathtt{D}}{\vdash [\mathtt{T}\,\mathtt{C}] <: [\mathtt{S}\,\mathtt{D}] : ok} \qquad \vdash [\mathtt{T}\,\mathtt{C}] \lessdot [\mathtt{S}\,\mathtt{D}] : ok$$

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

Figure 15: Constraint consistency rules

say 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 7. *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 let $C \triangleq \hat{C}_1 \wedge \cdots \wedge \hat{C}_j$ and $D \triangleq \hat{D}_1 \wedge \cdots \wedge \hat{D}_k$. Then define:

$$\hat{C} \in C \iff \hat{C} \in \bigcup_{i=1}^{j} \left\{\hat{C}_i\right\}$$

$$D \subseteq C \iff \bigcup_{i=1}^{k} \left\{\hat{D}_i\right\} \subseteq \bigcup_{i=1}^{j} \left\{\hat{C}_i\right\}$$

Constraint closure is then defined via the rewrite rules given in Fig. 14 and Definition 8. 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 8 (CONSTRAINT 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 $C \rightarrow^{\star}_{close} close(C)$.*

The purpose of closure and consistency checks is twofold. Firstly, these processes algorithmically ensure satisfiability of inferred type constraint, established as follows:

LEMMA 5. *If $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ is derivable, then $C, \mathtt{T}$ is realizable iff $close(C)$ is consistent.*

Secondly, closure generates a constraint that is amenable to decidable trace effect extraction. It is essential to observe that the trace effect constraints in a constraint generated by inference and closure define a system of concrete lower bounds on trace effect variables:

DEFINITION 9. *Let C be closed and consistent. Then $\mathtt{H}'$ is a* component of $\mathtt{H}$ in C *iff $\mathtt{H}'$ is a subterm of $\mathtt{H}$, or there exists a subterm $\mathtt{h}$ of $\mathtt{H}$ such that $\mathtt{H}'' <: \mathtt{h} \in C$ and $\mathtt{H}'$ is a component of $\mathtt{H}''$ in C. The* abstract components of $\mathtt{H}$ in C *are the variable components $\mathtt{h}$ of $\mathtt{H}$ that have no lower bound in C.*

LEMMA 6. *Let $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ be derivable for closed e. Then $\mathtt{H}_1 <: \mathtt{H}_2 \in close(C)$ implies that $\mathtt{H}_2$ is a variable $\mathtt{h}$, and $\mathtt{H}$ has no abstract components in C.*

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 $close(C)$. Formally, the trace effect extraction algorithm is defined in Fig. 16, with associated abbreviations as follows:

DEFINITION 10. *The $hextract$ algorithm in Fig. 16 is defined with respect to a given constraint C. At the "top-level", we write $hextract(\mathtt{H}, C)$ to denote $hextract(\mathtt{H}, \varnothing)$ given C.*

Extraction returns a closed trace effect that is a "best" unified representation of the top-level effect of a given closed expression, in the following sense:

LEMMA 7 (EXTRACTION CORRECTNESS). *If $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ is derivable for closed e and $\mathtt{H}' = hextract(\mathtt{H}, close(C))$, then $\mathtt{H}'$ is closed and $C \Vdash \mathtt{H}' <: \mathtt{H}$.*

Soundness of the analysis is then obtained immediately by Lemma 4, Lemma 5, and Lemma 7.

THEOREM 1 (SOUNDNESS OF ANALYSIS). *If $\Gamma, C, \mathtt{H} \vdash_W \mathtt{e} : \mathtt{T}$ is derivable for closed e and $hextract(\mathtt{H}, close(C))$ is valid, then $\epsilon, \mathtt{e} \rightarrow^{\star} \eta, \mathtt{E}[\mathtt{chk}[\mathtt{x}]]$ implies $\eta \vdash \mathtt{chk}[\mathtt{x}]$.*

7. CONCLUSION

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

7.1 Related Work

The idea of using some form of abstract program interpretation as input to model checking [26] for verification of specified program properties has been explored previously, e.g. in [4, 11, 6]. In these particular 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. However, none of these works defines a rigorous process for extracting an LTS from higher-order or Object Oriented programs.

Security automata [23] use finite automata for the specification and run-time enforcement of language safety properties, loosely understood as specifying well-formed event sequences. Systems have also been developed for statically verifying correctness of security automata using dependent types [29], and in a more general

$$
\begin{array}{rcll}
hextract(\epsilon, hs) & = & \epsilon & \\
hextract(\texttt{ev[i]}, hs) & = & \texttt{ev[i]} & \\
hextract(\texttt{chk[i]}, hs) & = & \texttt{chk[i]} & \\
hextract(\texttt{h}, hs) & = & \texttt{h} & \texttt{h} \in hs \\
hextract(\texttt{h}, hs) & = & \mu\texttt{h}.hextract(bounds(\texttt{h}), hs \cup \{\texttt{h}\}) & \texttt{h} \notin hs \\
hextract(\texttt{H}_1;\texttt{H}_2, hs) & = & (hextract(\texttt{H}_1, hs));(hextract(\texttt{H}_2, hs)) & \\
hextract(\texttt{H}_1|\texttt{H}_2, hs) & = & (hextract(\texttt{H}_1, hs))|(hextract(\texttt{H}_2, hs)) &
\end{array}
$$

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

Figure 16: $hextract$ **and** $bounds$ **functions**

form as refinement types [22]. These systems do not extract any abstract interpretations, and so are in a somewhat different category than the aforementioned (and our) work.

Perhaps the most closely related work is [20], 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. The system in [5] is also closely related, but verifies so-called local policies, where regions of code are required to satisfy specified invariants.

Their usages U are similar to our trace effects H, but the usages have a much more complex grammar, and appear to have no real gain in expressiveness. Our trace effects can easily be seen to form an LTS for which model-checking is decidable; their usages are significantly more complex, so it is unclear if model-checking will be possible.

The systems in [12, 7, 19, 6] 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, which we believe is critical for application to Object Oriented settings.

Some recent work [21, 2] 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.

7.2 Summary

We have defined the language FJ_{sec}, 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 FJ_{sec} 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.

Acknowledgements

Thanks to Scott Smith and David Van Horn for comments and suggestions on drafts of this paper.

8. REFERENCES

[1] Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03)*, feb 2003.

[2] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 98–109. ACM Press, 2005.

[3] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems*. Imperial College Press, 1999.

[4] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[5] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Policy framings for access control. In *WITS '05: Proceedings of the 2005 workshop on Issues in the theory of security*, pages 5–11. ACM Press, 2005.

[6] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *J. Computer Security*, 9:217–250, 2001.

[7] Frédéric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Secure calling contexts for stack inspection. In *Proceedings of the Fourth ACM SIGPLAN Conference on*

Principles and Practice of Declarative Programming (PPDP'02), pages 76–87. ACM Press, 2002.

[8] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[9] O. Burkart, D. Caucal, F. Moller, , and B. Steffen. Verification on infinite structures. In S. Smolka J. Bergstra, A. Pons, editor, *Handbook on Process Algebra*. North-Holland, 2001.

[10] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292. ACM Press, 1991.

[11] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington, DC, November 18–22, 2002.

[12] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, 2000.

[13] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[14] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1995.

[15] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

[16] Gerard J. Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability*, 11(2):65–79, 2001.

[17] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 331–342, Portland, Oregon, January 2002.

[18] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[19] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[20] P. J. Stuckey K. Marriott and M. Sulzmann. Resource usage verification. In *Proc. of First Asian Programming Languages Symposium, APLAS 2003*, 2003.

[21] Francesco Logozzo. Separate compositional analysis of class-based object-oriented languages. In *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST'2004)*, volume 3116 of *Lectures Notes in Computer Science*, pages 332–346. Springer-Verlag, July 2004.

[22] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, Uppsala, Sweden, August 2003.

[23] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[24] Christian Skalka and Scott Smith. History effects and verification. In *Asian Programming Languages Symposium*, number 3302 in Lecture Notes in Computer Science. Springer, November 2004.

[25] Christian Skalka, Scott Smith, and David Van Horn. 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.

[26] B. Steffen and O. Burkart. Model checking for context-free processes. In *CONCUR'92, Stony Brook (NY)*, volume 630 of *Lecture Notes in Computer Science (LNCS)*, pages 123–137, Heidelberg, Germany, 1992. Springer-Verlag.

[27] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[28] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145, pages 349–365. Springer Verlag, 1996.

[29] David Walker. A type system for expressive security policies. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267, Boston, Massachusetts, January 2000.