Types and Trace Effects of Higher Order Programs

CHRISTIAN SKALKA
The University of Vermont, Burlington, Vermont, USA
(*e-mail:* `skalka@cs.uvm.edu`)

SCOTT SMITH
The Johns Hopkins University, Baltimore, Maryland, USA
(*e-mail:* `scott@cs.jhu.edu`)

DAVID VAN HORN
Brandeis University, Waltham, Massachusetts, USA
(*e-mail:* `dvanhorn@cs.brandeis.edu`)

Abstract

This paper shows how type effect systems can be combined with model-checking techniques to produce powerful, automatically verifiable program logics for higher order programs. The properties verified are based on the ordered sequence of events that occur during program execution, so called *event traces*. Our type and effect systems infer conservative approximations of the event traces arising at run-time, and model-checking techniques are used to verify logical properties of these histories. Our language model is based on the λ-calculus. Technical results include a type inference algorithm for a polymorphic type effect system, and a method for applying known model-checking techniques to the *trace effects* inferred by the type inference algorithm, allowing static enforcement of history- and stack-based security mechanisms. A type safety result is proven for both unification and subtyping constraint versions of the type system, ensuring that statically well-typed programs do not contain trace event checks that can fail at run-time.

1 Introduction

Safe and secure program execution is crucial for modern information systems, but is difficult to attain in practice due to both programmer errors and intentional attacks. Various programming language-based techniques exist for increasing program safety, by verifying at compile- and/or run-time that programs possess certain safety properties. In addition to narrowly focused systems for verification of specific properties such as memory safety or stack inspection security, recent research has explored security paradigms for enforcing general classes of properties. One such paradigm comprises the class of properties of program *event traces* that can be expressed in temporal logics (Skalka & Smith, 2004; Skalka, 2005; K. Marriott & Sulzmann, 2003; Bartoletti *et al.*, 2005b; Igarashi & Kobayashi, 2002). This paper establishes a foundation for automated verification of higher order programs in this paradigm, using type and effect analysis. We develop a process for automatically predicting program event traces at compile-time and for statically verifying properties of traces.

Events are records of program actions, explicitly inserted into program code either manually (by the programmer) or automatically (by the compiler). Events are 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, and assertions enforce properties of event traces.While such properties can be anything in the abstract, in practice they must be expressed in automatically verifiable logics. Therefore, we use model-checkable linear temporal logics that express regular properties of traces (Steffen & Burkart, 1992).

For example, if a program is sending and receiving data over an SSL socket, the relevant events are opening and closing of sockets, and reading and writing of data packets. An example event trace produced by a program run could be:

```
ssl_open("snork.cs.jhu.edu",4434); ssl_hs_begin(4434);
ssl_hs_success(4434); ssl_put(4434); ssl_get(4434);
ssl_open("moo.cs.uvm.edu",4435); ssl_hs_begin(4434);
ssl_put(4435); ssl_close(4434); ssl_close(4435)
```

Here, `ssl_open` is a sample event with two arguments, a url and a port. Event traces can then be used to detect logical flaws or security violations. For SSL, sockets must first be opened, handshake begun, handshake success, and only then can data be get/put over the socket. Thus, the above trace is illegal because data is put on socket 4435 before notification has been received that handshake was successful on that socket. Codifying this property of event traces as a local check in a decidable logic provides a rigorous definition of well-formedness, *and* allows mechanical verification of it.

Trace based analyses have been shown capable of statically enforcing flow-sensitive security properties such as safe locking behaviour (Foster *et al.*, 2002) and resource usage policies such as file usage protocols and memory management (K. Marriott & Sulzmann, 2003; Igarashi & Kobayashi, 2002). In (Bartoletti *et al.*, 2005a), an analysis adapted from the one presented in this paper is used to enforce secure service composition. The history-based access control model of (Abadi & Fournet, 2003) can be implemented with event traces and checks (Skalka & Smith, 2004), as can be the policies realizable in that model, e.g. sophisticated Chinese Wall policies (Abadi & Fournet, 2003). Stack-based security policies are also amenable to this form of analysis, as shown in (Skalka & Smith, 2004; Skalka *et al.*, 2005) 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.

1.1 A Type-Based Approach

The focus of this paper is on the compile-time verification of higher order program trace properties expressible in linear temporal logic. The verification itself is based on known model checking techniques for program abstractions, and we define a novel type theory to infer program trace abstractions, yielding a fully automated analysis. These abstractions, called *trace effects*, are integrated directly into the type language, so that type reconstruction automates abstract interpretation of program trace behaviour. We develop both a *Hindley-Milner (HM) style* system where types are represented in term form and the implementation relies on unification, and a *subtyping constraint* system where the implementation relies on constraint closure. The flexibility of parametric polymorphism over types and effects is available in both systems. Type theory provides an especially rigorous and extensible foundation for the static analysis of trace-based program properties in a higher order language setting. The main contribution of this paper is the development and study of this foundation, and definition of a sound automated system covering both automatic extraction of program trace approximations and their verification.

To characterise the analysis and prove its correctness, the metatheory of types provides an appealing formalism. We demonstrate subject reduction, progress, and type safety results for constraint subtyping, which extend to the Hindley-Milner system by virtue of conservation of the latter in the former. Conservativity of both systems with respect to the underlying pure Hindley-Milner system is also demonstrated. Soundness and completeness of subtyping constraint inference is proved, as is soundness for unification-based inference. Completeness of constraint subtyping also entails a principal types result for the system. So-called *trace approximation* is demonstrated for both type systems, meaning that trace effects conservatively approximate run-time trace behaviour.

To complete the analysis, we define and prove correct an automatic verification technique for trace effects, based on model checking in the linear μ-calculus. We also develop a variation on the language model, so that stack-based trace policies can be defined. In a stack-based model, trace events are "popped" with their associated activations, allowing definition of policies such as stack inspection. The static analysis and type safety result is modified to accommodate this variation, requiring only a post-processing transformation of inferred trace effects to implement.

An OCaml implementation of our analysis is presented and discussed in (Van Horn, 2006a), and is available for download (Van Horn, 2006b). The implementation includes both the HM style and constraint subtyping systems, and the stack-based post-processing transformation of effects.

1.2 Organisation of the Paper

In Sect. 2, we introduce the language model λ_{trace}, that includes dynamic traces in configurations and dynamic checks on traces. In Sect. 3, we define a language of trace effects for static approximation of dynamic traces. In Sect. 4, we define a Hindley-Milner style polymorphic type system for assigning effects to programs. In Sect. 5, we give a unification-based inference algorithm for automatically reconstructing program effects. In Sect. 6 we define a more flexible constraint subtyping system and in Sect. 7 we define an inference algorithm for reconstructing types in this system. In Sect. 8, we define an alternate stack-based version of the language and

c	$\in$	$\mathcal{C}$	*atomic constants*
b	$::=$	$\mathsf{true} \mid \mathsf{false}$	*boolean values*
v	$::=$	$x \mid \lambda_z x.e \mid c \mid b \mid \neg \mid \vee \mid \wedge \mid ()$	*values*
e	$::=$	$v \mid e\,e \mid ev(e) \mid \mathsf{if}\, e \,\mathsf{then}\, e \,\mathsf{else}\, e \mid \mathsf{let}\, x = v \,\mathsf{in}\, e$	*expressions*
η	$::=$	$\epsilon \mid ev(c) \mid \eta; \eta$	*traces*
E	$::=$	$[\,] \mid v\,E \mid E\,e \mid ev(E) \mid \mathsf{if}\, E \,\mathsf{then}\, e \,\mathsf{else}\, e$	*evaluation contexts*

Fig. 1. λ_{trace} language syntax

show how post-processing inferred effects can approximate traces in this variation. In Sect. 9, we develop an automatic model-checking technique for verification of effects. In Sect. 10, we describe examples of how the system can be applied to language-based security. We conclude with a discussion of related work and final summary in Sect. 11.

2 The Language λ_{trace}

In this Section we develop the syntax and semantics of our language model λ_{trace}.

2.1 Syntax

The syntax of the theory λ_{trace} is given in Fig. 1. The base values include booleans and the unit value (). Expressions $\mathsf{let}\, x = v \,\mathsf{in}\, e$ are included to implement let-polymorphism in the type system (Sect. 4). Functions, written $\lambda_z x.e$, possess a recursive binding mechanism where z is the self variable. We assume the following syntactic sugarings:

$$e_1 \wedge e_2 \triangleq \wedge e_1 e_2 \qquad e_1 \vee e_2 \triangleq \vee e_1 e_2 \qquad \lambda x.e \triangleq \lambda_z x.e \quad z \text{ not free in } e$$

$$\lambda\_.e \triangleq \lambda x.e \quad x \text{ not free in } e \qquad e_1; e_2 \triangleq (\lambda\_.e_2)(e_1)$$

Events ev are named entities parameterised by constants c (we treat only the unary case in this presentation, but the extension to n-ary events is straightforward). These constants $c \in \mathcal{C}$ are abstract, with $\mathcal{C}$ formally defined as a countably infinite set of arbitrary identifiers; this set could for example be strings or IP addresses. Ordered sequences of these events constitute traces η, which maintain the sequence of events experienced during program execution. We let $\hat{\eta}$ denote the string obtained from this sequence by removing delimiters (;). We distinguish a subset of events ev_ϕ identified by assertions ϕ in a to-be-specified logical syntax (defined in Sect. 9). These are *check* events, and the semantics will require that run-time checks succeed in order for computation to progress. We presuppose existence of a meaning function Π such that $\Pi(\phi(c), \hat{\eta})$ holds iff $\phi(c)$ is valid for $\hat{\eta}$; we also leave the meaning function Π abstract until later (Sect. 9).

Parameterising events and checks with constants c allows for a more expressive event language; for example, in Sect. 10 we show how the parameterised privileges of Java stack inspection can be encoded with the aid of these parameters.

2.2 Semantics

The operational semantics of λ_{trace} is defined in Fig. 2 via the call-by-value small step reduction relations $\rightsquigarrow$ and $\rightarrow$ on configurations η, e, where η is the run-time program event trace. We write $\rightarrow^\star$ to denote the reflexive, transitive closure of $\rightarrow$. Note that in the *event* reduction rule, an event $ev(c)$ encountered during execution is added to the end of the trace. The *check* rule specifies that when a configuration $\eta, ev_\phi(c)$ is encountered during execution, it is appended to the end of η like other events, and also $\phi(c)$ is required to be satisfied by the string $(\hat{\eta}\, ev_\phi(c))$, which is the concatenation of $\hat{\eta}$ and $ev_\phi(c)$ (see Definition 3.1), according to our meaning function Π. The reasons for treating checks as dynamic events is manifold; for one, some checks may ensure that other checks have occurred in the trace. Also, this scheme will simplify the definition of Π, as well as typing and verification. In case a check fails at runtime, execution is "stuck"; formally:

Definition 2.1
We say that a configuration η, e is *stuck* iff e is not a value and there does not exist η' and e' such that $\eta, e \rightarrow \eta', e'$. If $\epsilon, e \rightarrow^\star \eta, e'$ and η, e' is stuck, then e is said to *go wrong*.

The following example demonstrates the basics of syntax and operational semantics.

Example 2.1 (*File state*)
Let the function *w/file* be defined as:

$$\mathit{w/file} \triangleq \lambda \mathit{fn}.\lambda f.\mathit{open}(\mathit{fn}); f(\mathit{fn}); \mathit{close}(\mathit{fn})$$

And let *readtwice* be defined as:

$$\mathit{readtwice} \triangleq \lambda \mathit{fn}.\mathit{read}(\mathit{fn}); \mathit{read}(\mathit{fn}); ()$$

Where *open*(*fn*), *close*(*fn*), and *read*(*fn*) each induce events $ev_{open}, ev_{close}, ev_{read}$, respectively, parameterised by the filename *fn*, and *readtwice* reads twice from file *fn* and returns the unit value (), i.e. it is evaluated only for effect. Then in the operational semantics, we have:

$$\epsilon, \mathit{w/file}\ \mathtt{f}\ \mathit{readtwice} \rightarrow^\star ev_{open}(\mathtt{f}); ev_{read}(\mathtt{f}); ev_{read}(\mathtt{f}); ev_{close}(\mathtt{f}), ()$$

3 Trace Effects

The goal of our static analysis is to conservatively approximate trace behaviour of programs, specifically the trace that will be generated by a program during its evaluation, and to predict success or failure of program checks on the basis of this approximation. We will use a type system to reconstruct *trace effects*, which constitute the approximation.

In essence, trace effects H conservatively approximate traces η that may develop during execution, by representing a set of traces containing at least η. Trace effects are generated by the following grammar:

$$H \quad ::= \quad \epsilon \mid h \mid ev(c) \mid H; H \mid H|H \mid \mu h.H \qquad \textit{trace effects}$$

$$
\begin{array}{rcllr}
\eta, (\lambda_z x.e)v & \leadsto & \eta, e[v/x][\lambda_z x.e/z] & & (\beta) \\
\eta, \neg\mathsf{true} & \leadsto & \eta, \mathsf{false} & & (\mathit{notT}) \\
\eta, \neg\mathsf{false} & \leadsto & \eta, \mathsf{true} & & (\mathit{notF}) \\
\eta, \wedge\,\mathsf{true} & \leadsto & \eta, \lambda x.x & & (\mathit{andT}) \\
\eta, \wedge\,\mathsf{false} & \leadsto & \eta, \lambda\_.\mathsf{false} & & (\mathit{andF}) \\
\eta, \vee\,\mathsf{true} & \leadsto & \eta, \lambda\_.\mathsf{true} & & (\mathit{orT}) \\
\eta, \vee\,\mathsf{false} & \leadsto & \eta, \lambda x.x & & (\mathit{orF}) \\
\eta, \mathsf{if\,true\,then}\, e_1\, \mathsf{else}\, e_2 & \leadsto & \eta, e_1 & & (\mathit{ifT}) \\
\eta, \mathsf{if\,false\,then}\, e_1\, \mathsf{else}\, e_2 & \leadsto & \eta, e_2 & & (\mathit{ifF}) \\
\eta, \mathsf{let}\, x = v\, \mathsf{in}\, e & \leadsto & \eta, e[v/x] & & (\mathit{let}) \\
\eta, \mathit{ev}(c) & \leadsto & (\eta; \mathit{ev}(c)), () & & (\mathit{event}) \\
\eta, \mathit{ev}_\phi(c) & \leadsto & (\eta; \mathit{ev}_\phi(c)), () & \text{if } \Pi(\phi(c), \hat{\eta}\, \mathit{ev}_\phi(c)) & (\mathit{check}) \\
\eta, E[e] & \rightarrow & \eta', E[e'] & \text{if } \eta, e \leadsto \eta', e' & (\mathit{context})
\end{array}
$$

Fig. 2. λ_{trace} language semantics

A trace effect may be an event $ev(c)$, a sequencing of trace effects $H_1; H_2$, a nondeterministic choice of trace effects $H_1|H_2$, or a recursively bound trace effect $\mu h.H$ that finitely represents the set of possibly infinite traces that may be generated by recursive functions. Trace effects may contain check events $ev_\phi(c)$, allowing verification of checks at the right points in trace approximations.

We now make precise the way in which a trace effect approximates the run-time trace of a program. We define a Labelled Transition System (LTS) interpretation of trace effects as sets of strings over the alphabet of events plus a $\downarrow$ symbol to denote termination; abusing terminology, we also call these strings traces. Note that traces arising from program evaluation may be infinite, because programs may not terminate, but traces arising from the interpretation of trace effects will always be finite.

Definition 3.1
We write θ to denote possibly $\downarrow$ terminated strings over the alphabet of events:

$$
\begin{array}{rcl}
s & ::= & ev(c) \mid \epsilon \mid s\,s \\
\theta & ::= & s \mid s\downarrow
\end{array}
$$

We endow strings with an equational theory to interpret ϵ as the empty string and string concatenation as usual– more precisely, for all s, s_1, s_2, and s_3 the following equations hold:

$$s\,\epsilon = s \qquad\qquad \epsilon\, s = s \qquad\qquad (s_1 s_2)s_3 = s_1(s_2 s_3)$$

The symbol Θ is defined to range over prefix-closed sets of traces θ.

Trace effects generate traces by viewing them as programs in a simple nondeterministic transition system.

Definition 3.2

The trace effect transition relation on closed trace effects is defined as follows:

$$ev(c) \xrightarrow{ev(c)} \epsilon \qquad H_1|H_2 \xrightarrow{\epsilon} H_1 \qquad H_1|H_2 \xrightarrow{\epsilon} H_2 \qquad \mu h.H \xrightarrow{\epsilon} H[\mu h.H/h]$$

$$\epsilon; H \xrightarrow{\epsilon} H \qquad\qquad H_1; H_2 \xrightarrow{s} H_1'; H_2 \;\text{ if } H_1 \xrightarrow{s} H_1'$$

We formally determine the sets of traces Θ associated with a closed trace effect in terms of the transition relation:

Definition 3.3
The interpretation of trace effects is defined as follows:

$$[\![H]\!] = \; \{s_1 \cdots s_n \mid H \xrightarrow{s_1} \cdots \xrightarrow{s_n} H'\} \cup \{s_1 \cdots s_n \downarrow \mid H \xrightarrow{s_1} \cdots \xrightarrow{s_n} \epsilon\}$$

Any trace effect interpretation is clearly prefix-closed. In this interpretation, an infinite trace is viewed as the set of its finite prefixes.

Note that prefix closure does not cause any loss of information, since the postpending of $\downarrow$ to terminating traces allows them to be distinguished from their prefixes. In particular, this means that $(H_1; H_2) \neq H_1$ for arbitrary closed H_1 and $H_2 \neq \epsilon$.

Equivalence of trace effects is defined via their interpretation, i.e. $H_1 = H_2$ iff $[\![H_1]\!] = [\![H_2]\!]$. This relation is in fact undecidable: traces are equivalent to Basic Process Algebras (BPAs), as demonstrated in Sect. 9, and equivalence of BPAs is known to be undecidable (Burkart *et al.*, 2001).

Some trace effects contain occurrences of checks, i.e. their interpretation contains traces of the form $\theta\, ev_\phi(c)$. We define *validity* of trace effects in terms of the satisfiability of these checks, given their context θ; the check must hold for its immediate prefix:

Definition 3.4
A trace effect H is *valid* iff for all $\theta\, ev_\phi(c) \in [\![H]\!]$, $\Pi(\phi(c), \theta\, ev_\phi(c))$ holds.

3.1 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 3.1
We note the following properties of trace effect equivalence for all closed H, H_1,H_2, and H_3:

1. $H|H = H$
2. $\epsilon; H = H = H; \epsilon$
3. $\mu h.H = H$
4. $H_1|H_2 = H_2|H_1$
5. $H_1; (H_2; H_3) = (H_1; H_2); H_3$
6. $H_1|(H_2|H_3) = (H_1|H_2)|H_3$
7. $H_1; (H_2|H_3) = (H_1; H_2)|(H_1; H_3)$
8. $(H_1|H_2); H_3 = (H_1; H_3)|(H_2; H_3)$

$\alpha \in \mathcal{V}_{Sing}, t \in \mathcal{V}_{Type}, h \in \mathcal{V}_{Eff}$			*type variables*
$\beta \in \mathcal{V}_{Sing} \cup \mathcal{V}_{Type} \cup \mathcal{V}_{Eff}$			
s	$::=$	$c \mid \alpha$	*singleton types*
H	$::=$	$\epsilon \mid h \mid ev(s) \mid H;H \mid H \vert H \mid \mu h.H$	*trace effects*
τ	$::=$	$\beta \mid s \mid \{s\} \mid H \mid \tau \xrightarrow{H} \tau \mid bool \mid unit$	*types*
σ	$::=$	$\forall\bar{\beta}.\tau$	*type schemes*
Γ	$::=$	$\varnothing \mid \Gamma; x : \sigma$	*type environments*

Fig. 3. λ_{trace} type syntax

9. $H'[\mu h.H'/h] = \mu h.H'$ for all closed $\mu h.H'$.
10. Trace effect equivalence is homomorphic for 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 H approximates the traces generated by a program, and $[\![H]\!] \subseteq [\![H']\!]$, then H' is also a sound approximation. Like equality, containment is known to be undecidable.

Lemma 3.2
Writing $H \subseteq H'$ iff $[\![H]\!] \subseteq [\![H']\!]$, the following properties hold:

1. $H \subseteq H'$ is undecidable
2. $H \subseteq H|H'$
3. If $H \subseteq H'$ then $H|H'' \subseteq H'|H''$ and $H''; H \subseteq H''; H'$ and $H; H'' \subseteq H'; H''$ for all closed H''.
4. If $H \subseteq H'$ then validity of H' implies validity of H.

Finally, 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, interpreting traces η as the identical trace effect. This syntax is used, e.g., in the statement of subject reduction, Lemma 4.2 and Lemma 6.10.

4 Hindley-Milner Style Typing for λ_{trace}

The syntax of types for λ_{trace} is given in Fig. 3. This syntax is more liberal than what is actually allowed in type derivations, for example function domain and range types cannot be trace effects or "bare" singletons s, and the three different kinds of variables must occur only in the appropriate positions. Formally, we define as *well-kinded* only those type terms that have an interpretation in the ground tree model given in the sense of Definition 5.2 and Definition 5.3, below. Hereafter we restrict our presentation to well-kinded types. The syntax of types includes forms for booleans, unit, and function types of the form $\tau_1 \xrightarrow{H} \tau_2$, where *latent* effects H represent the traces that may result by use of the function. Events are side-effects, and so these function types are a form of effect type (Talpin & Jouvelot, 1992;

$$\textsc{Var}\ \frac{\Gamma(x) = \sigma}{\Gamma, \epsilon \vdash x : \sigma} \qquad \textsc{Const}\ \Gamma, \epsilon \vdash \mathbf{c} : \Delta(\mathbf{c}) \qquad \textsc{Event}\ \frac{\Gamma, H \vdash e : \{s\}}{\Gamma, H; ev(s) \vdash ev(e) : unit}$$

$$\forall\textsc{-Intro}\ \frac{\Gamma, \epsilon \vdash v : \tau \qquad \bar{\beta}\#\mathrm{fv}(\Gamma)}{\Gamma, \epsilon \vdash v : \forall\bar{\beta}.\tau} \qquad \forall\textsc{-Elim}\ \frac{\Gamma, \epsilon \vdash v : \forall\bar{\beta}.\tau}{\Gamma, \epsilon \vdash v : \tau[\bar{\tau}/\bar{\beta}]} \qquad \textsc{Weaken}\ \frac{\Gamma, H' \vdash e : \tau \qquad H' \sqsubseteq H}{\Gamma, H \vdash e : \tau}$$

$$\textsc{If}\ \frac{\Gamma, H_1 \vdash e_1 : bool \qquad \Gamma, H_2 \vdash e_2 : \tau \qquad \Gamma, H_2 \vdash e_3 : \tau}{\Gamma, H_1; H_2 \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau}$$

$$\textsc{App}\ \frac{\Gamma, H_1 \vdash e_1 : \tau' \xrightarrow{H_3} \tau \qquad \Gamma, H_2 \vdash e_2 : \tau'}{\Gamma, H_1; H_2; H_3 \vdash e_1 e_2 : \tau} \qquad \textsc{Fix}\ \frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash e : \tau'}{\Gamma, \epsilon \vdash \lambda_z x.e : \tau \xrightarrow{H} \tau'}$$

$$\textsc{Let}\ \frac{\Gamma, \epsilon \vdash v : \sigma \qquad \Gamma; x : \sigma, H \vdash e : \tau}{\Gamma, H \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau}$$

Fig. 4. λ_{trace} Hindley-Milner style typing rules

Amtoft *et al.*, 1999). Additionally, since events and predicates are parameterised, we must be especially accurate with respect to our typing of singleton constants. Thus, we adopt a very simple form of singleton type $\{c\}$ (Stone, 2000), where only atomic constants can have singleton type.

Types contain three kinds of variables: regular type variables t, singleton type variables α, and trace effect type variables h. The metavariable β ranges over all kinds of variables. Universal type schemes $\forall\bar{\beta}.\tau$ bind any kind of type variable in τ, where $\bar{\beta}$ is a vector of type variables. Note that let-polymorphism over types, singletons, and trace effects is included in our system. We formally define some convenient notation for vectors of variables:

Definition 4.1
Variable vectors, denoted $\bar{\beta}$, range over sequences of distinct variables $\beta_1 \cdots \beta_n$, with the empty vector denoted $\varnothing$. Vectors are equivalent up to reordering; hence, we may treat vectors $\beta_1 \cdots \beta_n$ as analogous sets $\{\beta_1, \ldots, \beta_n\}$, in particular adapting notation $\beta \in \bar{\beta}$ and $\bar{\beta}_1 \cup \bar{\beta}_2$ and $\bar{\beta}_1 \cap \bar{\beta}_2$ with the obvious meaning. We write $\bar{\beta}_1\#\bar{\beta}_2$ iff $\bar{\beta}_1 \cap \bar{\beta}_2 = \varnothing$. For any type scheme σ, we write $\mathrm{fv}(\sigma)$ to denote the free variables in σ, extending the notation to environments Γ in the obvious manner. We write τ as syntactic sugar for $\forall\varnothing.\tau$.

Without loss of generality, we equate type schemes up to α-renaming and extraneous bindings. That is, we assume the following axiom:

$$\forall\bar{\beta}_1\bar{\beta}_2.\tau = \forall\bar{\beta}_1.\tau \text{ if } \bar{\beta}_2\#\mathrm{fv}(\tau)$$

Source code type derivation rules for *judgements* $\Gamma, H \vdash e : \tau$ are given in Fig. 4, where Γ is an environment of variable typing assumptions, H is the effect of the expression e, and τ is the type of e. To assign types to constants, we posit a constant type binding environment Δ as follows.

Definition 4.2
Constants $\mathbf{c}$ range over the set $\{(), \mathsf{true}, \mathsf{false}, \neg, \vee, \wedge\} \cup \mathcal{C}$. The binding environment Δ contains bindings for all constants, consisting of $c : \{c\}$ for all $c \in \mathcal{C}$, and the following:

$$() : \mathit{unit} \qquad \mathsf{true} : \mathit{bool} \qquad \mathsf{false} : \mathit{bool} \qquad \neg : \mathit{bool} \xrightarrow{\epsilon} \mathit{bool} \qquad \wedge : \mathit{bool} \xrightarrow{\epsilon} \mathit{bool} \xrightarrow{\epsilon} \mathit{bool}$$

$$\vee : \mathit{bool} \xrightarrow{\epsilon} \mathit{bool} \xrightarrow{\epsilon} \mathit{bool}$$

Type safety will be defined in terms of *valid* typing judgements, which are derivable judgements where the top-level effect is valid. A subject reduction result will show that effects assigned by type analysis are conservative approximations of program trace behaviour, while validity of top-level effects will guarantee success of run-time checks.

Definition 4.3
A derivable type judgement $\Gamma, H \vdash e : \sigma$ is *valid* iff H is valid.

Example 4.1
Let *w/file* and *readtwice* be defined as in Example 2.1. Then the following judgements are derivable:

$$\varnothing, \epsilon \vdash \mathit{w/file} : \forall \mathit{fn}, h, \alpha.\{\mathit{fn}\} \xrightarrow{\epsilon} (\{\mathit{fn}\} \xrightarrow{h} \alpha) \xrightarrow{\mathit{ev}_{\mathit{open}};h;\mathit{ev}_{\mathit{close}}} \alpha$$

$$\varnothing, \epsilon \vdash \mathit{readtwice} : \forall \mathit{fn}.\{\mathit{fn}\} \xrightarrow{\mathit{ev}_{\mathit{read}}(\mathit{fn});\mathit{ev}_{\mathit{read}}(\mathit{fn})} \mathit{unit}$$

$$\varnothing, \mathit{ev}_{\mathit{open}}(\mathtt{f}); \mathit{ev}_{\mathit{read}}(\mathtt{f}); \mathit{ev}_{\mathit{read}}(\mathtt{f}); \mathit{ev}_{\mathit{close}}(\mathtt{f}) \vdash \mathit{w/file}\ \mathtt{f}\ \mathit{readtwice} : \mathit{unit}$$

4.1 Properties

One of the basic claims about our type system is that it is *conservative*, in that the addition of trace effects is a conservative extension to an underlying Hindley-Milner (HM) style let-polymorphic type system: by using weakening before each if-then-else typing, any derivation in the underlying effect-free type system may be replayed here. This result is formalised as follows.

Definition 4.4
We define the "event erasure" function erase_η on expressions as follows:

$$\begin{array}{rcl}
\mathrm{erase}_\eta(x) & = & x \\
\mathrm{erase}_\eta(\mathbf{c}) & = & \mathbf{c} \\
\mathrm{erase}_\eta(\mathit{ev}(c)) & = & () \\
\mathrm{erase}_\eta(e_1 e_2) & = & \mathrm{erase}_\eta(e_1)\mathrm{erase}_\eta(e_2) \\
\mathrm{erase}_\eta(\lambda_z x.e) & = & \lambda_z x.\mathrm{erase}_\eta(e) \\
\mathrm{erase}_\eta(\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_2) & = & \mathsf{if}\, \mathrm{erase}_\eta(e_1)\, \mathsf{then}\, \mathrm{erase}_\eta(e_2)\, \mathsf{else}\, \mathrm{erase}_\eta(e_3) \\
\mathrm{erase}_\eta(\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2) & = & \mathsf{let}\, x = \mathrm{erase}_\eta(e_1)\, \mathsf{in}\, \mathrm{erase}_\eta(e_2)
\end{array}$$

We also define the "trace effect erasure" function erase_H on types as follows:

$$\begin{array}{rcll} \mathrm{erase}_H(\tau) & = & \tau & \tau \in \{t, \mathit{bool}, \mathit{unit}, \{s\}\} \\ \mathrm{erase}_H(\tau_1 \xrightarrow{H} \tau_2) & = & \mathrm{erase}_H(\tau_1) \rightarrow \mathrm{erase}_H(\tau_2) & \end{array}$$

The function erase_H is extended to type environments Γ in the obvious manner.

Definition 4.5
We write $\Gamma \vdash e : \tau$ for judgements in a standard simple type system with let-polymorphism, e.g. that of (Pierce, 2002), extended with singleton constants and our Const rule.

Lemma 4.1 (Conservativity)
$\Gamma, H \vdash e : \tau$ is derivable iff $\mathrm{erase}_H(\Gamma) \vdash \mathrm{erase}_\eta(e) : \mathrm{erase}_H(\tau)$ is.

Proof
Straightforward by definition and induction on typing derivations. $\square$

Our type safety results are stated as follows. The proof of each is immediate by conservation of the HM style system in the constraint subtyping system presented in Sect. 6, Lemma 6.3, and analogous results in that system, which are proven in detail in Sect. 6.

Theorem 4.1 (Type Safety)
If $\Gamma, H \vdash e : \tau$ is valid for closed e, then e does not go wrong.

Theorem 4.2 (Progress)
If $\Gamma, H \vdash e : \tau$ is derivable for closed e and η, e is irreducible with $\eta; H$ valid, then e is a value.

A subject reduction result for the HM style system can also be proved via Lemma 6.3 and subject reduction in the constraint subtyping system.

Lemma 4.2 (Subject Reduction)
If $\Gamma, H \vdash e : \tau$ is derivable for closed e and $\eta, e \rightsquigarrow \eta', e'$, then $\Gamma, H' \vdash e' : \tau$ is derivable with $\eta'; H' \subseteq \eta; H$.

A significant corollary of this result is one of our guiding intuitions: that trace effects conservatively approximate the set of possible run-time traces, formalised as follows. Prefix closure of effect interpretation ensures that traces produced at any point in the computation, not just at termination, will be approximated.

Corollary 4.1
If $\Gamma, H \vdash e : \tau$ is derivable for closed e and $\epsilon, e \rightarrow^\star \eta, e'$ then $\hat{\eta} \in [\![H]\!]$.

5 Unification-Based Type Inference

In this section we show that a type inference algorithm exists for the HM style type system for λ_{trace} presented in the previous section. The type inference algorithm will be partly based on unification, in keeping with the HM subset of the system, but the presence of effect weakening requires a simple form of constraint solution. Thus, our

$$\textsc{Var}\quad \frac{\Gamma(x) = \forall\bar{\beta}.\tau/C}{\Gamma, \epsilon \vdash_{\bar{\beta}'} x : (\tau/C)[\bar{\beta}'/\bar{\beta}]} \qquad \textsc{Const}\quad \Gamma, \epsilon \vdash_{\varnothing} \mathbf{c} : \Delta(\mathbf{c})/\mathbf{true}$$

$$\textsc{Event}\quad \frac{\Gamma, H \vdash_{\bar{\beta}} e : \tau/C}{\Gamma, H; ev(\alpha) \vdash_{\bar{\beta}\cup\{\alpha\}} ev(e) : unit/C \wedge \tau \sqsubseteq \{\alpha\}}$$

$$\textsc{If}\quad \frac{\begin{array}{c}\bar{\beta}_1\#\bar{\beta}_2\#\bar{\beta}_3 \qquad \bar{\beta} = \bar{\beta}_1\bar{\beta}_2\bar{\beta}_3 \cup \{t\} \\ \Gamma, H_1 \vdash_{\bar{\beta}_1} e_1 : \tau_1/C_1 \qquad \Gamma, H_2 \vdash_{\bar{\beta}_2} e_2 : \tau_2/C_2 \qquad \Gamma, H_3 \vdash_{\bar{\beta}_3} e_3 : \tau_3/C_3\end{array}}{\Gamma, H_1; H_2|H_3 \vdash_{\bar{\beta}} \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : t/C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \sqsubseteq bool \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t}$$

$$\textsc{App}\quad \frac{\bar{\beta}_1\#\bar{\beta}_2 \qquad \Gamma, H_1 \vdash_{\bar{\beta}_1} e_1 : \tau_1/C_1 \qquad \Gamma, H_2 \vdash_{\bar{\beta}_2} e_2 : \tau_2/C_2}{\Gamma, H_1; H_2; h \vdash_{\bar{\beta}_1\bar{\beta}_2\cup\{t,h\}} e_1\ e_2 : t/C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t}$$

$$\textsc{Fix}\quad \frac{\Gamma; x : t; z : t \xrightarrow{h} t', H \vdash_{\bar{\beta}} e : \tau/C}{\Gamma, \epsilon \vdash_{\bar{\beta}\cup\{t,h,t'\}} \lambda_z x.e : t \xrightarrow{h} t'/C \wedge \tau \sqsubseteq t' \wedge H \sqsubseteq h}$$

$$\textsc{Let}\quad \frac{\bar{\beta}_1\#\bar{\beta}_2 \qquad \Gamma, \epsilon \vdash_{\bar{\beta}_1} v : \tau'/D \qquad \Gamma; x : \forall\bar{\beta}_1.\tau'/D, H \vdash_{\bar{\beta}_2} e : \tau/C}{\Gamma, H \vdash_{\bar{\beta}_1\bar{\beta}_2} \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau/C \wedge D}$$

Fig. 5. λ_{trace} type constraint inference rules

approach to inference will be constraint based, with type constraints interpreted as equality constraints, and effect constraints interpreted as containment constraints. In addition to being an effective approach in this context, we will see in Sect. 7 that it allows the same type inference algorithm to be used for a subtyping system, by imposing a subtyping interpretation of type constraints, and introducing a new constraint solution algorithm. That is, type inference is modular with respect to interpretation of constraints, in the spirit of systems such as HM(X) (Sulzmann, 2001).

5.1 Syntax and meaning of constraints

We begin by defining the syntax of constraints, where $\sqsubseteq$ is the constraint relation between types.

Definition 5.1
Constraints C are defined as follows:

$$C ::= \mathbf{true} \mid \tau \sqsubseteq \tau \mid C \wedge C$$

Constraints are required to agree in kind, i.e. for any constraint $\tau \sqsubseteq \tau'$, we require that $\tau : k$ and $\tau' : k$ for some k.

$$
\begin{array}{rcll}
\rho(h, hs) & = & \rho(h) & h \notin hs \\
\rho(h, hs) & = & h & h \in hs \\
\rho(ev(s), hs) & = & ev(\rho(s)) & \\
\rho(\epsilon, hs) & = & \epsilon & \\
\rho(H_1; H_2, hs) & = & \rho(H_1, hs); \rho(H_2, hs) & \\
\rho(H_1 | H_2, hs) & = & \rho(H_1, hs) | \rho(H_2, hs) & \\
\rho(\mu h.H, hs) & = & \mu h.\rho(H, hs \cup \{h\}) & \\
\\
\rho(c) & = & c & \\
\rho(\{s\}) & = & \{\rho(s)\} & \\
\rho(unit) & = & unit & \\
\rho(bool) & = & bool & \\
\rho(\tau_1 \xrightarrow{H} \tau_2) & = & \rho(\tau_1) \xrightarrow{\rho(H)} \rho(\tau_2) & \\
\rho(H) & = & \rho(H, \varnothing) &
\end{array}
$$

Fig. 6. Interpretations extended to types and effects

The appropriate meaning of constraints is obtained by interpretation in a model $\mathbb{T}$, which for the HM system we define as a universe of monotypes, endowed with a partial order $\preccurlyeq$ that is an equality relation on types, and a containment relation on effects. Later, to obtain a subtyping interpretation of constraints, we will redefine $\mathbb{T}$ and $\preccurlyeq$ appropriately.

Definition 5.2 (*Ground Type Model*)
Let $\mathbb{T}$ be the set of *ground*, or variable free, types τ, denoted $\hat{\tau}$ and generated by the following grammar, where $\hat{H}$ ranges over variable free trace effects:

$$\hat{\tau} \quad ::= \quad unit \mid bool \mid \{c\} \mid \hat{H} \mid \hat{\tau} \xrightarrow{\hat{H}} \hat{\tau}$$

Further, $\mathbb{T}$ is endowed with a partial order $\preccurlyeq$ axiomatised as follows:

$$\frac{[\![H]\!] \subseteq [\![H']\!]}{H \preccurlyeq H'}$$

Definition 5.3 (*Interpretation of Constraints*)
Interpretations ρ are total mappings from type variables β to $\mathbb{T}$. We impose the sanity condition that each kind of type variable is mapped to elements of $\mathbb{T}$ of the appropriate form, i.e. for all α, t, h, ρ we require that $\rho(\alpha)$ is a singleton c, and $\rho(h)$ is a closed effect H, and $\rho(t)$ is one of the other element forms. Interpretations are extended to types and effects as in Fig. 6, where in abuse of notation the interpretation of effects are parameterised 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 axiomatised as follows:

$$\rho \vdash \mathbf{true} \qquad \frac{\rho(\tau_1) \preccurlyeq \rho(\tau_2)}{\rho \vdash \tau_1 \sqsubseteq \tau_2} \qquad \frac{\rho \vdash C \qquad \rho \vdash D}{\rho \vdash C \wedge D}$$

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 Type inference and constraint solution

Type inference judgements are of the form $\Gamma, H \vdash_{\bar{\beta}} e : \tau/C$, where H, τ, and C are the reconstructed top level effect, type, and constraint, and $\bar{\beta}$ are the variables introduced during the derivation. Type schemes are now constrained, written $\forall\bar{\beta}.\tau/C$, and type environments Γ now bind variables to constrained type schemes. We write τ as shorthand for $\forall\varnothing.\tau/\mathbf{true}$. Type inference rules are defined in Fig. 5; they are deterministic except for arbitrary choice of variables. We call *canonical* those derivations where fresh variables are chosen whenever possible, where a variable β is fresh with respect a judgement $\Gamma, H \vdash_{\bar{\beta}} e : \tau/C$ iff it occurs nowhere in the judgement. Note that disjointness conditions on fresh variables chosen in subderivation are imposed in derivations, by the If, App, and Let rules, ensuring "global" freshness of variable choice in canonical derivations.

To automatically obtain HM style types for λ_{trace} expressions, it is also necessary to convert inferred types and constraints into unified, constraint-free form. To accomplish this, we define a unification algorithm, extended to also apply to effect constraints. Given a constraint generated by type inference, the resulting algorithm will generate a type substitution that solves the constraint.

Definition 5.4
Substitutions, written $[\tau_1/\beta_1, \ldots, \tau_n/\beta_n]$, are mappings from type variables to types, extended to types, type schemes, type environments, and constraints in the usual manner. We write $\mathrm{dom}([\tau_1/\beta_1, \ldots, \tau_n/\beta_n])$ to mean $\beta_1 \cdots \beta_n$. We let ψ range over substitutions $[\tau_1/\beta_1, \ldots, \tau_n/\beta_n]$, though the former is prefix notation while the latter is postfix. We write $\psi_1 \circ \psi_2$ to denote the substitution ψ such that $\psi(\tau) = \psi_1(\psi_2(\tau))$ for all τ. Extending the vector notation of Definition 4.1, we write $[\bar{\tau}/\bar{\beta}]$ to denote $[\tau_1/\beta_1, \ldots, \tau_n/\beta_n]$ where $\bar{\tau} = \tau_1 \cdots \tau_n$ and $\bar{\beta} = \beta_1 \cdots \beta_n$.

Since any solution of a constraint C must unify the type constraints in C, and also satisfy the effect constraints in C, constraint solutions need to be defined generally, as follows:

Definition 5.5
A substitution ψ is a *solution* of a constraint C iff $\rho \vdash \psi(C)$ for all ρ.

To define how to construct solutions of constraint, here and for the subtyping version of our type analysis presented later, it is useful to interpret constraints as sets, rather than conjunctions, of atomic constraints. Thus, we introduce the following notation.

Definition 5.6
Let $\hat{C}$ range over atomic constraints, i.e.:

$$\hat{C} ::= \mathbf{true} \mid \tau \sqsubseteq \tau$$

$$
\begin{array}{rcl}
\mathit{unify}(\mathbf{true}) & = & \mathbf{true} \\
\mathit{unify}(C \wedge \tau \sqsubseteq \tau) & = & \mathit{unify}(C) \\
\mathit{unify}(C \wedge \beta \sqsubseteq \tau) & = & \mathbf{fail} \text{ if } \beta \in \mathrm{fv}(\tau), \text{ else} \\
 & & \mathit{unify}(C[\tau/\beta]) \circ [\tau/\beta] \\
\mathit{unify}(C \wedge \tau \sqsubseteq \beta) & = & \mathbf{fail} \text{ if } \beta \in \mathrm{fv}(\tau), \text{ else} \\
 & & \mathit{unify}(C[\tau/\beta]) \circ [\tau/\beta] \\
\mathit{unify}(C \wedge \{s_1\} \sqsubseteq \{s_2\}) & = & \mathit{unify}(C \wedge s_1 \sqsubseteq s_2) \\
\mathit{unify}(C \wedge \tau_1 \xrightarrow{h} \tau_2 \sqsubseteq \tau_1' \xrightarrow{h'} \tau_2') & = & \mathit{unify}(C \wedge h \sqsubseteq h' \wedge \tau_1' \sqsubseteq \tau_1 \wedge \tau_2 \sqsubseteq \tau_2')
\end{array}
$$

Fig. 7. Constraint set unification

$$
\begin{array}{rcl}
\mathit{soln}(C) & = & \text{let } C_1, C_2 = \mathit{hsplit}(C) \text{ and } \psi_1 = \mathit{unify}(C_1) \text{ in } \mathit{soln}_{\mathbf{H}}(\psi_1(C_2)) \circ \psi_1 \\
\mathit{hsplit}(C) & = & C_1, C_2 \text{ where } \mathit{set}(C_2) = \{H_1 \sqsubseteq H_2 \mid H_1 \sqsubseteq H_2 \in C\} \\
 & & \qquad \text{and } C_1 = C - C_2 \\
\mathit{bounds}(h, C) & = & H_1 | \cdots | H_n \quad \text{where } \{H_1, \ldots, H_n\} = \{H \mid H \sqsubseteq h \in C\} \\
\mathit{soln}_{\mathbf{H}}(\varnothing) & = & \varnothing \\
\mathit{soln}_{\mathbf{H}}(C) & = & \text{let } \psi = [(\mu h.\mathit{bounds}(h, C))/h] \text{ in} \\
 & & \mathit{soln}_{\mathbf{H}}(\psi(C - \{H \sqsubseteq h \mid H \sqsubseteq h \in C\})) \circ \psi
\end{array}
$$

Fig. 8. Unification based solution

and given $C = \hat{C}_1 \wedge \cdots \wedge \hat{C}_n$, let $\mathit{set}(C) = \left\{\hat{C}_1, \ldots, \hat{C}_n\right\}$. Then:

$$
\begin{array}{rcl}
C \subseteq D & \iff & \mathit{set}(C) \subseteq \mathit{set}(D) \\
\hat{C} \in C & \iff & \hat{C} \in \mathit{set}(C) \\
C_1 - C_2 = D & \iff & \mathit{set}(C_1) - \mathit{set}(C_2) = \mathit{set}(D)
\end{array}
$$

Now, since effect containment is known to be undecidable, as observed in Sect. 3, it seems there can be no general constraint solution algorithm. However, type inference generates effect constraints of a particular form, in particular it yields a system of lower bounds $H \sqsubseteq h$ on effect variables h, as is immediately demonstrated by observation of the type inference rules in Fig. 5. For this restricted form of effect constraints, there exists a solution algorithm $\mathit{soln}_{\mathbf{H}}$, defined in Fig. 8, that joins lower bounds in the solution of effect variables h. By composing this with the unification algorithm *unify* defined in Fig. 7, a constraint solution algorithm *soln* is obtained for constraints generated by type inference. Note that inferred constraints need to be split into type and effect components by *hsplit*, before applying *unify* to the former and $\mathit{soln}_{\mathbf{H}}$ to the latter. Soundness of the type inference technique can be stated as follows; a proof of this result is given in (Van Horn, 2006a).

Theorem 5.1 (*Soundness of Inference*)

Given closed e; then if $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable, and $\psi = soln(C)$, then $\varnothing, \psi(H) \vdash e : \psi(\tau)$ is derivable.

6 Constraint Subtyping for $\lambda_{\mathbf{trace}}$

In this section we propose a more expressive system—whereby a strictly larger class of valid programs is verifiable—based on subtyping constraints. The HM style system developed in the previous section suffers from a certain degree of imprecision due to the interpretation of $\sqsubseteq$ as an equivalence relation on types. This imprecision results in program effect characterisations that are sound but not as tight as can be achieved through subtyping. In general, tighter effect characterisations result in the verifiability of more programs. Consider the following example and its typing under the HM style system:

$$f \triangleq \lambda x.(\mathsf{if\,true\,then}\,\lambda\_.\,ev_1(c)\,\mathsf{else}\,x); x$$

Observe that the effect of x and $\lambda\_.ev_1(c)$ must be weakened into equivalence as required by the typing rule for conditionals, hence we derive:

$$f : \forall ht.(t \xrightarrow{ev_1(c)|h} unit) \xrightarrow{\epsilon} (t \xrightarrow{ev_1(c)|h} unit)$$

In other words, the effect of $\lambda\_.ev_1(c)$ is forced to be subsumed by the effect of x, resulting in a lack of precision, further illustrated by application of f:

$$f(\lambda\_.ev_2(c)) : t \xrightarrow{ev_1(c)|ev_2(c)} unit$$

Although this expression evaluates to $\lambda\_.ev_2(c)$, which clearly has only the effect $ev_2(c)$ when applied, the latent effect of this function is given as $ev_1(c) \mid ev_2(c)$ by the type system. Supposing the function is applied with a subsequent check for the occurrence of $ev_2(c)$, the program would fail to validate since $ev_2(c)$ *may*—but not must—occur according to this characterisation of the program's effect.

To remedy the situation and obtain a more expressive system, the subtyping relation will allow types, as well as effects, to be weakened as necessary. In treating the above example, the type of x will not be forced to be equivalent to $\lambda\_.ev_1(c)$, but can be weakened via subsumption to allow typing of the conditional expression in the body of f, while retaining a purely abstract latent effect on x in a most general typing of f. The essence of the subtyping relation is an adaptation of effect weakening to latent effect on function types, for example:

$$t \xrightarrow{h} unit \sqsubseteq t \xrightarrow{ev_1(c)|h} unit$$

Thus we can derive a more precise typing for f:

$$f : \forall ht.(t \xrightarrow{h} unit) \xrightarrow{\epsilon} (t \xrightarrow{h} unit)$$

and also for $f(\lambda\_.ev_2(c))$:

$$f(\lambda\_.ev_2(c)) : t \xrightarrow{ev_2(c)} unit$$

Again supposing the resulting function is applied with a subsequent check for $ev_2(c)$, the program would validate since this tighter characterisation of the program's effect appropriately gives that $ev_2(c)$ *must* occur. Taken together with the conservativity result demonstrated in Lemma 6.3, which states that any HM style derivation may be replayed in the subtyping system, this shows that a strictly larger class of valid programs are verifiable under the constraint subtyping system as compared to the HM style unification based system of the previous section.

Subtleties of the relation include contravariance (covariance) in function domain (range) types, respectively, and treatment of free variables in related types; these issues are resolved in Sect. 6.2 below.

The use of a constraint representation of types has several distinct benefits resulting from the greater precision of constraint types. For one, constraints may be recursive, allowing typing of self-referential expressions such as $\lambda x.xx$. Furthermore, constraints allow a representation of conjunctive and disjunctive types (Eifrig *et al.*, 1995), without requiring conjunctive and disjunctive type syntax and interpretation. While disjunction can be represented in effect terms via the nondeterministic choice operator ($|$), a term representation of effect conjunction would require the addition of a parallel-and operation in effects, which is known to significantly increase the complexity of model-checking (Burkart *et al.*, 2001). In short, a constraint representation simplifies the subtyping analysis; while it yields a more cumbersome, less readable typing than a unified representation, model-checkable effects can still be extracted, as we show in Sect. 7. Since our goal is an automatic program analysis tool, this is agreeable.

6.1 Logical Subtyping Judgements and Derivations

Logical subtyping judgements are of the form $\Gamma, C, H \vdash e : \tau$, with all syntactic forms as defined in previous sections. In any such judgement, we refer to C and H as the *top-level* constraint and effect. The constant type binding environment Δ is also as defined previously. Note that constraints C, previously occurring only in inference judgements, now occur in logical judgements; of course, a significant difference in the current system is the interpretation of $\sqsubseteq$ as a subtyping relation, formalised below in Sect. 6.2. Note well that we adopt the interpretation of constraints given in Definition 5.3; subtyping will be realized purely through a redefinition of the model $\mathbb{T}$ in Definition 6.3.

To allow more flexibility in the typing of singleton constants, we make a small extension to the type and effect language:

Definition 6.1
We extend the language of singleton types in Fig. 3 with the form $s|s$. Additionally, we define the semantics of an effect $ev(c_1|c_2)$ as equivalent to $ev(c_1)|ev(c_2)$.

Anticipating the interpretation, the meaning of a type $s_1|s_2$ subsumes the meaning of s_1 and s_2– i.e., it is a disjunctive singleton type.

While the logical typing rules are based on previous subtyping constraint systems such as (Eifrig *et al.*, 1995) and (Skalka & Pottier, 2003), there is a difference, in that in any judgement $\Gamma, C, H \vdash e : \tau$, the scope of quantified type variables $\bar{\beta}$ in

$$\textsc{Var}\ \frac{\Gamma(x) = \sigma}{\Gamma, C, \epsilon \vdash x : \sigma} \qquad \textsc{Const}\ \Gamma, C, \epsilon \vdash \mathbf{c} : \Delta(\mathbf{c}) \qquad \textsc{Event}\ \frac{\Gamma, C, H \vdash e : \{s\}}{\Gamma, C, H; \mathit{ev}(s) \vdash \mathit{ev}(e) : \mathit{unit}}$$

$$\forall\text{-}\textsc{Intro}\ \frac{\Gamma, C, \epsilon \vdash v : \tau \qquad \bar{\beta} \# \mathrm{fv}(\Gamma)}{\Gamma, C, \epsilon \vdash v : \forall \bar{\beta}.\tau} \qquad \forall\text{-}\textsc{Elim}\ \frac{\Gamma, C, \epsilon \vdash v : \forall \bar{\beta}.\tau \qquad C \Vdash [\bar{\tau}/\bar{\beta}]}{\Gamma, C, \epsilon \vdash v : \tau[\bar{\tau}/\bar{\beta}]}$$

$$\textsc{Sub}\ \frac{\Gamma, C, H' \vdash e : \tau' \qquad C \Vdash \tau' \sqsubseteq \tau \qquad C \Vdash H' \sqsubseteq H}{\Gamma, C, H \vdash e : \tau}$$

$$\textsc{If}\ \frac{\Gamma, C, H_1 \vdash e_1 : \mathit{bool} \qquad \Gamma, C, H_2 \vdash e_2 : \tau \qquad \Gamma, C, H_2 \vdash e_3 : \tau}{\Gamma, C, H_1; H_2 \vdash \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : \tau}$$

$$\textsc{App}\ \frac{\Gamma, C, H_1 \vdash e_1 : \tau' \xrightarrow{H} \tau \qquad \Gamma, C, H_2 \vdash e_2 : \tau'}{\Gamma, C, H_1; H_2; H \vdash e_1\ e_2 : \tau} \qquad \textsc{Fix}\ \frac{\Gamma; x : \tau'; z : \tau' \xrightarrow{H} \tau, C, H \vdash e : \tau}{\Gamma, C, \epsilon \vdash \lambda_z x.e : \tau' \xrightarrow{H} \tau}$$

$$\textsc{Let}\ \frac{\Gamma, C, \epsilon \vdash v : \sigma \qquad \Gamma; x : \sigma, C, H \vdash e : \tau}{\Gamma, C, H \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau}$$

Fig. 9. λ_{trace} constraint typing, logical rules

type schemes in Γ effectively extend to C; in particular, noting the form of the rule $\forall$-Elim, we define:

Definition 6.2
Write $C \Vdash [\bar{\tau}/\bar{\beta}]$ iff $C \Vdash C[\bar{\tau}/\bar{\beta}]$.

Thus, given a top-level constraint $C \triangleq h_1 \sqsubseteq h_2$, the type scheme $\forall h_1 h_2.\mathit{unit} \xrightarrow{h_1} \mathit{unit}$ is more general than $\forall h_1.\mathit{unit} \xrightarrow{h_1} \mathit{unit}$, even though h_2 does not occur in the quantified type. Observe that given the former type scheme, the rule $\forall$-Elim essentially imposes monomorphism on h_1 due to its relation with h_2 specified in C, whereas the latter type scheme allows any instantiation $[H_1/h_1]$ as long as there also exists a substitution $[H_2/h_2]$ such that $C \Vdash H_1 \sqsubseteq H_2$. This state of affairs is implicitly due to a simplification we have made in the type system, where all relevant typing constraints are subsumed in the top-level constraint C. In the type inference system presented in Sect. 7, where constraints must be reconstructed, rather than given *a priori*, we will return to constrained type scheme forms.

The logical subtyping derivation rules defined in Fig. 9 are quite similar to the HM style rules defined in Fig. 4, other than the rule Sub, which incorporates type subsumption as well as weakening of top-level effects. As mentioned above, type subsumption allows weakening of latent effects on function types. The meaning of the subtyping relation and validity of type judgements is defined in terms of constraint solutions, formalised in the next section.

$$\frac{\varphi \text{ is finite}}{\bot \preccurlyeq_{\text{fin}} \varphi} \qquad \frac{\varphi \text{ is finite}}{\varphi \preccurlyeq_{\text{fin}} \top} \qquad \varphi \preccurlyeq_{\text{fin}} \varphi|\varphi' \qquad \varphi \preccurlyeq_{\text{fin}} \varphi'|\varphi \qquad \frac{\varphi \preccurlyeq_{\text{fin}} \varphi'}{\{\varphi\} \preccurlyeq_{\text{fin}} \{\varphi'\}}$$

$$\frac{[\![H]\!] \subseteq [\![H']\!]}{H \preccurlyeq_{\text{fin}} H'} \qquad \frac{\varphi_1' \preccurlyeq_{\text{fin}} \varphi_1 \quad \varphi_2 \preccurlyeq_{\text{fin}} \varphi_2' \quad H \preccurlyeq_{\text{fin}} H' \quad \varphi_1, \varphi_1', \varphi_2, \varphi_2' \text{ finite}}{\varphi_1 \xrightarrow{H} \varphi_2 \preccurlyeq_{\text{fin}} \varphi_1' \xrightarrow{H} \varphi_2'}$$

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

Fig. 10. Primitive subtyping for regular trees

$$\top^{\varnothing}, \bot^{\varnothing} : \mathit{Type} \qquad \mathit{unit}^{\varnothing} : \mathit{Type} \qquad \mathit{bool}^{\varnothing} : \mathit{Type} \qquad \mathit{sc}^{\varnothing} : \mathit{Sing} \qquad \{\mathit{sc}\}^{\varnothing} : \mathit{Type}$$

$$H^{\varnothing} : \mathit{Eff} \qquad (\cdot \xrightarrow{\cdot} \cdot)^{\mathit{Type}, \mathit{Eff}, \mathit{Type}} : \mathit{Type}$$

Fig. 11. Kinding rules for regular tree constructors

6.2 Interpretation of Subtyping Constraints

Following (Trifonov & Smith, 1996), subtyping is defined via interpretation in a regular tree model endowed with a *primitive subtyping* relation, a technique that allows us to accommodate recursive constraints. The relation is defined inductively via finite approximations. Intuitively, the subtyping relation is invariant on *bool* and *unit* types, is contravariant (resp. covariant) on the domain (resp. range) types of functions, and is covariant on latent function effects. As before, the relation $\sqsubseteq$ on trace effects is defined via set containment in the LTS interpretation of related effects.

Definition 6.3 (*Regular Tree Model*)
Let the *tree constructor kinds* be defined as:

$$k ::= \mathit{Type} \mid \mathit{Eff} \mid \mathit{Sing}$$

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

$$\begin{array}{lcl} \mathit{sc} & ::= & c \mid \mathit{sc}|\mathit{sc} \\ \mathit{tc} & ::= & \top \mid \bot \mid \mathit{sc} \mid \mathit{unit} \mid \mathit{bool} \mid \{\mathit{sc}\} \mid H \mid \cdot \xrightarrow{\cdot} \cdot \end{array}$$

where each element of the alphabet is indexed by a signature, written tc^{ς}, and must be *well-kinded* according to the rules given in Fig. 11.

A *tree* φ is a partial function from finite sequences (*paths*) π of natural numbers $\mathbb{N}^\star$ to L such that $\text{dom}(\varphi)$ is prefix-closed. Furthermore, for all $\pi n \in \text{dom}(\varphi)$, with $\mathit{tc}^{\varsigma} = \varphi(\pi)$, it is the case that $\varphi(\pi n) : \varsigma(n)$. The *subtree at* $\pi \in \text{dom}(\varphi)$ 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* $\varphi \mid_n$ for $\varphi \in \mathbb{T}$ as the finite tree obtained by replacing all subtrees at level n of φ with $\top$. Then, $\preccurlyeq_{\mathrm{fin}}$ is the partial order over finite $\varphi \in \mathbb{T}$ axiomatised in Fig. 10, and $\preccurlyeq$ is the partial order over $\mathbb{T}$ approximated by $\preccurlyeq_{\mathrm{fin}}$ axiomatised in Fig. 10.

By retaining Definition 5.3 with $\mathbb{T}$ and $\preccurlyeq$ refigured in this manner, and extending interpretations as defined in Fig. 6 to disjunctive singleton types as $\rho(s_1|s_2) = \rho(s_1)|\rho(s_2)$, we obtain a subtyping interpretation of constraints. We immediately note that this imposes transitivity and reflexivity on $\sqsubseteq$:

Lemma 6.1
The relation $\sqsubseteq$ is transitive and reflexive, by which we mean $C \Vdash \tau \sqsubseteq \tau$, and $C \Vdash \tau_1 \sqsubseteq \tau_2 \wedge \tau_2 \sqsubseteq \tau_3$ implies $C \Vdash \tau_1 \sqsubseteq \tau_3$.

It also is useful to characterise what sort of types cannot be placed in a $\sqsubseteq$ relation, especially to establish a canonical forms result (Lemma 6.14).

Lemma 6.2
Let $\asymp$ be the least symmetric relation on types (pronounced "clashes with") axiomatised by the following relational schemas:

$$\mathit{unit} \asymp \mathit{bool} \qquad \mathit{unit} \asymp \{s\} \qquad \mathit{unit} \asymp \tau_1 \xrightarrow{H} \tau_2 \qquad \mathit{bool} \asymp \{s\} \qquad \mathit{bool} \asymp \tau_1 \xrightarrow{H} \tau_2$$

$$\{s\} \asymp \tau_1 \xrightarrow{H} \tau_2$$

If C is solvable then for all τ_1, τ_2 such that $\tau_1 \asymp \tau_2$ it is the case that $C \not\Vdash \tau_1 \sqsubseteq \tau_2$.

Given our interpretation of constraints, we can now define validity of type judgements. Note that top level effects may contain free variables that must be interpreted via top level constraints in order to ascertain validity of expression effects.

Definition 6.4
A judgement $\Gamma, C, H \vdash e : \tau$ is *satisfiable* iff it is derivable and C has a solution. The judgement is *valid* iff it is satisfiable and there exists a solution ρ of C such that $\rho(H)$ is valid.

6.3 Properties

In this section we demonstrate our type safety and progress results, which along with correctness of type reconstruction will constitute the main results of this paper. The result proceeds via a subject reduction argument, modified to account for trace effects. Type safety is explicitly demonstrated for the constraint subtyping system; to allow application to the unified system of Sect. 4, we demonstrate the following conservativity result. Note that this also allows conservativity of Hindley-Milner typings à la Lemma 4.1 to be extended to constraint subtyping.

Lemma 6.3
If $\Gamma, H \vdash e : \tau$ is derivable, then so is $\Gamma, \mathbf{true}, H \vdash e : \tau$.

Proof
Straightforward by induction and case analysis on the last rule instance in the derivation of $\Gamma, H \vdash e : \tau$. Each derivation step in $\Gamma, H \vdash e : \tau$ can be mimicked by a same-named rule instance in the constraint subtyping system, other than instances of WEAKEN, which can be simulated by instances of SUB, by reflexivity of $\sqsubseteq$ and since it is easy to show that $H \sqsubseteq H'$ implies $C \Vdash H \sqsubseteq H'$ for all C. □

In later proofs the following observations about effect subtyping will be important; since interpretation of effect subtyping is defined in terms of containment in the the LTS semantics defined in Sect. 3, the stated properties follow by Lemma 3.2.

Lemma 6.4
The following properties hold:

1. $C \Vdash H \sqsubseteq H|H'$
2. If $C \Vdash H \sqsubseteq H'$ then $C \Vdash H|H'' \sqsubseteq H'|H''$ and $C \Vdash H''; H \sqsubseteq H''; H'$ and $C \Vdash H; H'' \sqsubseteq H'; H''$.

On the way to proving type safety and progress, we demonstrate a suite of standard results for the current type system. This includes weakening and instantiation; both properties follow in a similar manner to corresponding results in (Skalka & Pottier, 2003):

Lemma 6.5 (*Weakening*)
If $\Gamma, C, H \vdash e : \tau$ is derivable and $C' \Vdash C$, then so is $\Gamma, C', H \vdash e : \tau$ by a derivation of the same structure.

Lemma 6.6 (*Instantiation*)
If $\Gamma, C, H \vdash e : \tau$ is derivable, then so is the judgement $\psi(\Gamma), \psi(C), \psi(H) \vdash e : \psi(\tau)$ by a derivation of the same structure.

Although the following Lemma seems obscure, it brings to light some subtle issues related to polymorphism and generalisation, including the requirement that top-level effects be ϵ for generalisation and instantiation, and that generalisation cannot be performed on free environment variables. It also highlights the soundness of generalisation over variables in the top-level constraint, provided the requirements on instantiation. In essence, the result shows that the $\forall$ introduction and elimination forms do not allow over generalisation.

Lemma 6.7
Consecutive instances of $\forall$-INTRO and $\forall$-ELIM may be suppressed.

Proof
Assume the following derivation structure:

$$\cfrac{\cfrac{\Gamma, C, \epsilon \vdash e : \tau \qquad \bar{\beta}\#\mathrm{fv}(\Gamma)}{\Gamma, C, \epsilon \vdash e : \forall\bar{\beta}.\tau \qquad C \Vdash [\bar{\tau}/\bar{\beta}]}}{\Gamma, C, \epsilon \vdash e : \tau[\bar{\tau}/\bar{\beta}]}$$

where $\Gamma, C, \epsilon \vdash e : \tau$ is derivable by assumption. Thus $\Gamma[\bar{\tau}/\bar{\beta}], C[\bar{\tau}/\bar{\beta}], \epsilon[\bar{\tau}/\bar{\beta}] \vdash$

$e : \tau[\bar{\tau}/\bar{\beta}]$ is derivable by a derivation of the same structure, by Lemma 6.6. But $\epsilon[\bar{\tau}/\bar{\beta}] = \epsilon$, and $\Gamma[\bar{\tau}/\bar{\beta}] = \Gamma$ since $\bar{\beta}\#\mathrm{fv}(\Gamma)$ by assumption, therefore the result follows by Lemma 6.5 since $C \Vdash C[\bar{\tau}/\bar{\beta}]$ by assumption and Definition 6.2. □

The following result allows normalisation of type derivations. When inducting on the structure of type derivations, normal form derivations allow type judgements to be *inverted*; their subderivations can be deconstructed from root judgements, given the form of the root expression.

Lemma 6.8 (*Normalisation*)
If $\Gamma, C, H \vdash e : \tau$ is derivable, then it must follow by an instance of Sub from a judgement $\mathcal{J}$ such that:

i. if $e = \lambda_z x.e'$ then $\mathcal{J}$ follows by Fix;
ii. if $e = e_1 e_2$ then $\mathcal{J}$ follows by App;
iii. if $e = \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3$ then $\mathcal{J}$ follows by If;
iv. if $e = ev(e')$ then $\mathcal{J}$ follows by Event;
v. if e is one of $\mathsf{true}, \mathsf{false}, (), \vee, \wedge, \neg, \mathbf{c}$, then $\mathcal{J}$ follows by Const;
vi. if $e = x$ then $\mathcal{J}$ follows by Var and $\forall$-Elim;
vii. if $e = (\mathsf{let}\, x = v \,\mathsf{in}\, e')$ the $\mathcal{J}$ follows by Let.

Proof
By Lemma 6.7, any consecutive instances of $\forall$-Intro and $\forall$-Elim can be suppressed. In effect, this allows us to restrict all instances of $\forall$-Elim to immediately follow instances of Var, and all instances of $\forall$-Intro to immediately precede Let. Furthermore, by Lemma 6.1 it is easy to show that consecutive instances of Sub can be collapsed into a single instance, and due to reflexivity of Sub, trivial instances of Sub can be inserted anywhere in a derivation other than following $\forall$-Intro. Since any derivation of a judgement $\Gamma, C, H \vdash e : \tau$ must contain a syntax-directed rule instance corresponding to the form of e, with a judgement $\mathcal{J}$ as the consequence, the result follows by suppressing, collapsing, or inserting non-syntax-directed rule instances as appropriate between $\mathcal{J}$ and $\Gamma, C, H \vdash e : \tau$. □

Corollary 6.1
If $\Gamma, C, H \vdash v : \tau$ is derivable, then so is $\Gamma, C, \epsilon \vdash v : \tau'$ with $C \Vdash \tau' \sqsubseteq \tau$ and $C \Vdash \epsilon \sqsubseteq H$.

Proof
Straightforward by the previous Lemma, and by observing that if a judgement $\Gamma, C, H' \vdash v : \tau'$ follows by a syntax-directed rule applicable to a value v, then $H' = \epsilon$. □

In order to demonstrate the *let* and β reduction cases of subject reduction, we prove a values substitution Lemma as follows.

Lemma 6.9 (*Substitution*)
If both $\Gamma; x : \sigma', C, H \vdash e : \sigma$ and $\Gamma, C, \epsilon \vdash v : \sigma'$ are derivable, then so is $\Gamma, C, H \vdash e[v/x] : \sigma$.

Proof

By induction on the derivation of $\Gamma; x : \sigma', C, H \vdash e : \sigma$ and case analysis on the last step in the derivation. We restrict our consideration to the most interesting cases.

Case $\forall$-Intro. In this case $\sigma = \forall\bar{\beta}.\tau$, where by definition of $\forall$-Intro we can reconstruct:

$$\frac{\Gamma; x : \sigma', C, H \vdash e : \tau \qquad \bar{\beta}\#\mathrm{fv}(\Gamma; x : \sigma')}{\Gamma; x : \sigma', C, H \vdash e : \forall\bar{\beta}.\tau}$$

But by the induction hypothesis the judgement $\Gamma, C, H \vdash e[v/x] : \tau$ is derivable, and since clearly $\mathrm{fv}(\Gamma) \subseteq \mathrm{fv}(\Gamma; x : \sigma')$ therefore $\bar{\beta}\#\mathrm{fv}(\Gamma)$, so this case holds via an instance of $\forall$-Intro.

Case Var. In this case $e = y$, $H = \epsilon$, and by definition of Var we have that $(\Gamma; x : \sigma')(y) = \sigma$. Now, suppose on the one hand that $x \neq y$. Then $e[v/x] = y$, and $\Gamma(y) = \sigma$, so this case holds via an instance of Var. Suppose on the other hand that $x = y$; then $e[v/x] = v$ and $\sigma = \sigma'$, so the result follows by an assumption of the Lemma.

Case Fix. In this case $e = \lambda_z y.e'$ and $\sigma = \tau_1 \xrightarrow{H'} \tau_2$ and $H = \epsilon$, and by definition of Fix we can reconstruct:

$$\frac{\Gamma; x : \sigma'; y : \tau_1; z : \tau_1 \xrightarrow{H'} \tau_2, C, H' \vdash e' : \tau_2}{\Gamma; x : \sigma', C, \epsilon \vdash \lambda_z y.e' : \tau_1 \xrightarrow{H'} \tau_2}$$

Supposing that $x \neq y$ and $x \neq z$, it is the case that:

$$\Gamma; x : \sigma'; y : \tau_1; z : \tau_1 \xrightarrow{H'} \tau_2 = \Gamma; y : \tau_1; z : \tau_1 \xrightarrow{H'} \tau_2; x : \sigma'$$

Hence the following is derivable by the induction hypothesis:

$$\Gamma; y : \tau_1; z : \tau_1 \xrightarrow{H'} \tau_2, C, H' \vdash e'[v/x] : \tau_2$$

and by Fix:

$$\frac{\Gamma; y : \tau_1; z : \tau_1 \xrightarrow{H'} \tau_2, C, H' \vdash e'[v/x] : \tau_2}{\Gamma; x : \sigma', C, \epsilon \vdash \lambda_z y.(e'[v/x]) : \tau_1 \xrightarrow{H'} \tau_2}$$

so the result follows, since $\lambda_z y.(e'[v/x]) = e[v/x]$ by definition of substitution. Supposing that $x = y$, it is the case that:

$$\Gamma; x : \sigma'; y : \tau_1; z : \tau_1 \xrightarrow{H'} \tau_2 = \Gamma; y : \tau_1; z : \tau_1 \xrightarrow{H'} \tau_2$$

and $e[v/x] = v$ by definition of substitution, so we can reconstruct by Fix:

$$\frac{\Gamma; y : \tau_1; z : \tau_1 \xrightarrow{H'} \tau_2, C, H' \vdash e' : \tau_2}{\Gamma, C, \epsilon \vdash (\lambda_z y.e')[v/x] : \tau_1 \xrightarrow{H'} \tau_2}$$

The result follows similarly supposing that $x = z$. $\square$

Type safety ultimately relies on the preservation of type validity by $\rightarrow^\star$. Since the relation $\rightarrow^\star$ is predicated on both $\rightsquigarrow$ and $\rightarrow$ forms of reduction, we therefore need to prove subject reduction results for each form. First, for $\rightsquigarrow$:

Lemma 6.10 (Subject Reduction $\rightsquigarrow$)

If $\Gamma, C, H \vdash e : \tau$ is derivable and $\eta, e \leadsto \eta', e'$, then $\Gamma, C, H' \vdash e' : \tau$ is derivable with $C \Vdash \eta'; H' \sqsubseteq \eta; H$.

Proof
By induction on the derivation of $\Gamma, C, H \vdash e : \tau$ and case analysis on the rule form of $\eta, e \leadsto \eta', e'$. We restrict our consideration to the most interesting representative cases.

Case β. The following assertions hold by assumption in this subcase:

$$e = (\lambda_z x.e'')v \qquad e' = e''[v/x][\lambda_z x.e''/z] \qquad \eta' = \eta$$

By Lemma 6.8, reconstruct the following, moving from the root upwards; by Sub:

$$\frac{\Gamma, C, H_1; H_2; H_3 \vdash (\lambda_z x.e'')v : \tau_2 \qquad C \Vdash \tau_2 \sqsubseteq \tau \qquad C \Vdash H_1; H_2; H_3 \sqsubseteq H}{\Gamma, C, H \vdash (\lambda_z x.e'')v : \tau}$$

and by App:

$$\frac{\Gamma, C, H_1 \vdash \lambda_z x.e'' : \tau_1 \xrightarrow{H_3} \tau_2 \qquad \Gamma, C, H_2 \vdash v : \tau_1}{\Gamma, C, H_1; H_2; H_3 \vdash (\lambda_z x.e'')v : \tau_2}$$

and by Sub:

$$\frac{\Gamma, C, \epsilon \vdash \lambda_z x.e'' : \tau_1' \xrightarrow{H_3'} \tau_2' \qquad C \Vdash \tau_1' \xrightarrow{H_3'} \tau_2' \sqsubseteq \tau_1 \xrightarrow{H_3} \tau_2 \qquad C \Vdash \epsilon \sqsubseteq H_1}{\Gamma, C, H_1 \vdash \lambda_z x.e'' : \tau_1 \xrightarrow{H_3} \tau_2}$$

and by Fix:

$$\frac{\Gamma; x : \tau_1'; z : \tau_1' \xrightarrow{H_3'} \tau_2', C, H_3' \vdash e'' : \tau_2'}{\Gamma, C, \epsilon \vdash \lambda_z x.e'' : \tau_1' \xrightarrow{H_3'} \tau_2'}$$

and by Sub and Corollary 6.1:

$$\frac{\Gamma, C, \epsilon \vdash v : \tau_3 \qquad C \Vdash \epsilon \sqsubseteq H_2 \qquad C \Vdash \tau_3 \sqsubseteq \tau_1}{\Gamma, C, H_2 \vdash v : \tau_1}$$

Now, observe that by contravariance of $\sqsubseteq$ in function domain types and transitivity of $\sqsubseteq$ we have $C \Vdash \tau_3 \sqsubseteq \tau_1'$, hence by Sub and the above we can derive $\Gamma, C, H_2 \vdash v : \tau_1'$, so by successive applications of Lemma 6.9 we have that $\Gamma, C, H_3' \vdash e' : \tau_2'$ is derivable. But $C \Vdash H_3' \sqsubseteq \epsilon; \epsilon; H_3'$ by Lemma 3.1, and since by the above and definition of $\sqsubseteq$:

$$C \Vdash H_3' \sqsubseteq H_3 \qquad C \Vdash \epsilon \sqsubseteq H_1 \qquad C \Vdash \epsilon \sqsubseteq H_2$$

therefore by Lemma 6.4:

$$C \Vdash H_3' \sqsubseteq H_1; H_2; H_3$$

and since $C \Vdash H_1; H_2; H_3 \sqsubseteq H$ and $C \Vdash \tau_2' \sqsubseteq \tau$ by the above and transitivity of $\sqsubseteq$, by Sub:

$$\frac{\Gamma, C, H_3' \vdash e' : \tau_2' \qquad C \Vdash \tau_2' \sqsubseteq \tau \qquad C \Vdash H_3' \sqsubseteq H}{\Gamma, C, H \vdash e' : \tau}$$

so this case holds.

Case *let*. In this case $e = \mathsf{let}\, x = v \,\mathsf{in}\, e_0$ and $e' = e_0[v/x]$ and $\eta = \eta'$ by definition. By Lemma 6.8, reconstruct the following; by SUB:

$$\frac{\Gamma, C, H_0 \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e_0 : \tau_0 \qquad C \Vdash H_0 \sqsubseteq H \qquad C \Vdash \tau_0 \sqsubseteq \tau}{\Gamma, C, H \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e_0 : \tau}$$

and by LET:

$$\frac{\Gamma, C, \epsilon \vdash v : \sigma \qquad \Gamma; x : \sigma, C, H_0 \vdash e_0 : \tau_0}{\Gamma, C, H_0 \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e_0 : \tau_0}$$

But then $\Gamma, C, H_0 \vdash e' : \tau_0$ is derivable by Lemma 6.9, hence by the above and SUB:

$$\frac{\Gamma, C, H_0 \vdash e' : \tau_0 \qquad C \Vdash H_0 \sqsubseteq H \qquad C \Vdash \tau_0 \sqsubseteq \tau}{\Gamma, C, H \vdash e' : \tau}$$

so this case holds.

Case *event*. In this case $e = \mathit{ev}(c)$ and $e' = ()$ and $\eta' = \eta; \mathit{ev}(c)$. By Lemma 6.8 and definition of the SUB, EVENT, and CONST we can reconstruct:

$$\frac{\Gamma, C, H'; \mathit{ev}(s) \vdash \mathit{ev}(c) : \mathit{unit} \qquad C \Vdash \mathit{unit} \sqsubseteq \tau \qquad C \Vdash H'; \mathit{ev}(s) \sqsubseteq H}{\Gamma, C, H \vdash \mathit{ev}(c) : \tau}$$

and:

$$\frac{\dfrac{\Gamma, C, \epsilon \vdash c : \{c\} \qquad C \Vdash \{c\} \sqsubseteq \{s\} \qquad C \Vdash \epsilon \sqsubseteq H'}{\Gamma, C, H' \vdash c : \{s\}}}{\Gamma, C, H'; \mathit{ev}(s) \vdash \mathit{ev}(c) : \mathit{unit}}$$

But then by applications of CONST and SUB:

$$\frac{\Gamma, C, \epsilon \vdash () : \mathit{unit} \qquad C \Vdash \mathit{unit} \sqsubseteq \tau \qquad C \Vdash \epsilon \sqsubseteq H'}{\Gamma, C, H' \vdash () : \tau}$$

and since $C \Vdash \{c\} \sqsubseteq \{s\}$ by the above, therefore $C \Vdash c \sqsubseteq s$ and hence $C \Vdash \mathit{ev}(c) \sqsubseteq \mathit{ev}(s)$ by Definition 5.3, thus by Lemma 6.4 we have $C \Vdash \eta; \mathit{ev}(c); H' \sqsubseteq \eta; \mathit{ev}(s); H'$, so this case holds. □

Before proving subject reduction for $\rightarrow$, we need to establish some typing results related to evaluation contexts. Firstly, if e is a redex, then the effect of $E[e]$ should reflect that the trace predicted for e is the "first effect that will happen" in the evaluation of $E[e]$. Formally, we characterise this as follows:

Lemma 6.11 (*Context Inversion*)
If $\Gamma, C, H \vdash E[e] : \tau$ is derivable for closed $E[e]$, then there exists $H_1; H_2$ such that $\Gamma, C, H_1; H_2 \vdash E[e] : \tau$ is derivable with a subderivation concluding in $\Gamma, H_1 \vdash e : \tau'$ for e in the hole, where $C \Vdash H_1; H_2 \sqsubseteq H$.

Proof
By induction on E. In the basis $E = []$ and $E[e] = e$ so the result follows by assumption and Lemma 3.1, taking $H_2 = \epsilon$. Otherwise, the remaining forms of E– $E'e'$, vE', $\mathsf{if}\, E' \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$, and $\mathit{ev}(E')$– ensure that H will at least reflect that

the effect of e "happens first". Taking $E = E'e'$ for example, so that $E[e] = E'[e]e'$, by Lemma 6.8 and Sub and App we can reconstruct:

$$\frac{\Gamma, C, H_1'; H_2'; H_3' \vdash E'[e]e' : \tau' \qquad C \Vdash \tau' \sqsubseteq \tau \qquad C \Vdash H_1'; H_2'; H_3' \sqsubseteq H}{\Gamma, C, H \vdash E'[e]e' : \tau}$$

and:

$$\frac{\Gamma, C, H_1' \vdash E'[e] : \tau_1 \xrightarrow{H_3'} \tau_2 \qquad \Gamma, C, H_2' \vdash e' : \tau_1}{\Gamma, C, H_1'; H_2'; H_3' \vdash E'[e]e' : \tau'}$$

But then by the induction hypothesis, there exists H_2'' such that $\Gamma, C, H_1; H_2'' \vdash E'[e] : \tau_1 \xrightarrow{H_3'} \tau_2$ is derivable with $C \Vdash H_1; H_2'' \sqsubseteq H_1'$, so the result follows by an instance of App, Lemma 3.1 and Lemma 6.4, taking $H_2 = H_2''; H_2'; H_3'$. The other cases follow in similar manner, case $E = vE'$ with a little help from Corollary 6.1. □

Corollary 6.2
If $\Gamma, C, H \vdash E[ev(c)] : \tau$ is derivable, there exists H' such that $C \Vdash ev(c); H' \sqsubseteq H$.

We also show that typing is preserved by contextual substitution:

Lemma 6.12 (Context Substitution)
If $\Gamma, C, H_1; H_2 \vdash E[e] : \tau$ is derivable with a subderivation concluding in $\Gamma, C, H_1 \vdash e : \tau'$ for e in the hole, and $\Gamma, C, H_1' \vdash e' : \tau'$ is derivable, then so is $\Gamma, C, H_1'; H_2 \vdash E[e'] : \tau$.

Proof
Straightforward by induction on E and Lemma 6.11. □

Now, the main result for $\rightarrow$ reduction:

Lemma 6.13 (Subject Reduction $\rightarrow$)
If $\Gamma, C, H \vdash e : \tau$ is derivable for closed e and $\eta, e \rightarrow \eta', e'$, then $\Gamma, C, H' \vdash e' : \tau$ is derivable with $C \Vdash \eta'; H' \sqsubseteq \eta; H$.

Proof
The following hold by assumption and definition of $\rightarrow$:

$$e = E[e_1] \text{ with } e_1 \text{ a redex} \qquad e' = E[e_2] \qquad \eta, e_1 \leadsto \eta', e_2$$

Also, by Lemma 6.11 there exists a derivation of $\Gamma, C, H_1; H_2 \vdash e : \tau$ with $C \Vdash H_1; H_2 \sqsubseteq H$, and with a subderivation concluding in $\Gamma, C, H_1 \vdash e_1 : \tau'$ for e_1 in the hole. Furthermore, by Lemma 6.10 we have that $\Gamma, H_1' \vdash e_2 : \tau'$ is derivable with:

$$C \Vdash \eta'; H_1' \sqsubseteq \eta; H_1$$

Now, the following judgement is derivable by Lemma 6.12:

$$\Gamma, C, H_1'; H_2 \vdash E[e_2] : \tau$$

and by Lemma 3.1, Lemma 6.4, preceding facts, and Lemma 6.1:

$$\begin{array}{rcl} C \Vdash \eta; H_1; H_2 & \sqsubseteq & \eta; H \\ C \Vdash \eta'; H_1'; H_2 & \sqsubseteq & \eta; H_1; H_2 \\ C \Vdash \eta'; H_1'; H_2 & \sqsubseteq & \eta; H \end{array}$$

The result follows taking $H' = H_1'; H_2$. □

Note that as a corollary of this result, we may formalise the intuition that "trace effects predict run-time program traces":

Corollary 6.3 (*Trace Approximation*)
If $\Gamma, C, H \vdash e : \tau$ is derivable for closed e and $\epsilon, e \to^\star \eta, e'$ and $C \Vdash H \sqsubseteq H'$ for closed H', then $\hat{\eta} \in [\![H']\!]$.

Proof
By Lemma 6.13 and induction on the length of reduction there exists H'' such that $\Gamma, C, H'' \vdash e' : \tau$ is derivable with $C \Vdash \eta; H'' \sqsubseteq H$. Now, suppose ρ is a solution of C, meaning that $[\![\eta; \rho(H'')]\!] \subseteq [\![\rho(H)]\!]$, therefore clearly $\hat{\eta} \in [\![\rho(H)]\!]$. Since also $[\![\rho(H)]\!] \subseteq H'$, the result follows. □

Using well-known strategy, we demonstrate progress on the basis of a canonical forms lemma, establishing the form of values denoted by a particular type:

Lemma 6.14 (*Canonical Forms*)
If $\Gamma, C, H \vdash v : \tau$ is satisfiable for closed v, then the following conditions hold:

i. If $\tau = \mathit{bool}$ then v is either true or false.
ii. If $\tau = \mathit{unit}$ then $v = ()$.
iii. If there exists s such that $\tau = \{s\}$ then there exists c such that $v = c$.
iv. If there exists τ_1, H, τ_2 such that $\tau = \tau_1 \xrightarrow{H} \tau_2$ then either v is some function $\lambda_z x.e$, or v is in $\{\wedge, \vee, \neg\}$.

Proof
Suppose that $\tau = \mathit{bool}$, and suppose on the contrary that $v \notin \{\mathsf{true}, \mathsf{false}\}$. Then by Lemma 6.8 and remaining possible forms of v it is the case that $\Gamma, C, \epsilon \vdash v : \tau'$ such that $C \Vdash \tau' \sqsubseteq \mathit{bool}$ where τ' is of the form unit, $\{c\}$, or $\tau_1 \xrightarrow{H} \tau_2$. Lemma 6.2 implies the contradiction. The other cases follow in a similar manner. □

Finally, on the basis of the proceeding we can prove our main type safety results.

Theorem 6.1 (*Progress*)
Suppose $\Gamma, C, H \vdash e : \tau$ is satisfiable, there exists a solution ρ of C such that $\rho(\eta; H)$ is valid, and η, e is irreducible. Then e is a value.

Proof
Suppose on the contrary that e is not a value. Then by definition of $\to$, e is of the form $E[f]$, with $\Gamma, C, H' \vdash f : \tau'$ derivable as a subderivation of $\Gamma, C, H \vdash e : \tau$, and where one of the following cases holds by definition of $\rightsquigarrow$:

Case (f is one of $\{\mathsf{if}\, v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2, \wedge v, \vee v, \neg v\}$, where v is not a boolean value).

C-FN
$$(\tau_1 \xrightarrow{H} \tau_2 \sqsubseteq \tau_1' \xrightarrow{H'} \tau_2') \rightsquigarrow_{close} (\tau_1' \sqsubseteq \tau_1 \wedge \tau_2 \sqsubseteq \tau_2' \wedge H \sqsubseteq H')$$

C-TRANS
$$(\tau_1 \sqsubseteq \tau_2 \wedge \tau_2 \sqsubseteq \tau_3) \rightsquigarrow_{close} \tau_1 \sqsubseteq \tau_3$$

C-SINGLETON
$$\{\tau_1\} \sqsubseteq \{\tau_2\} \rightsquigarrow_{close} \tau_1 \sqsubseteq \tau_2$$

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

Fig. 12. Constraint closure rules

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

$$\frac{\tau, \beta : k}{\vdash \tau \sqsubseteq \beta : ok} \qquad \frac{\tau, \beta : k}{\vdash \beta \sqsubseteq \tau : ok} \qquad \vdash \tau_1 \xrightarrow{H} \tau_2 \sqsubseteq \tau_1' \xrightarrow{H'} \tau_2' : ok \qquad \vdash \{\tau_1\} \sqsubseteq \{\tau_2\} : ok$$

Fig. 13. Constraint consistency rules

In this case it is easy to show by inversion of either IF or APP that $\Gamma, C, H'' \vdash v : bool$. The contradiction follows by Lemma 6.14.

Case (f is $ev(v)$ and v is not a singleton constant c). Similar to the previous case.

Case (f is $\phi(c)$ where $\Pi(\phi(c), \hat{\eta}\, ev_\phi(c))$ does not hold). By Corollary 6.2, there exists H_2 such that $C \Vdash ev_\phi(c); H_2 \sqsubseteq H$. By assumption there exists a solution ρ of C such that $\rho(\eta; H)$ is valid, and $[\![\rho(\eta; ev_\phi(c); H_2)]\!] \subseteq [\![\rho(\eta; H)]\!]$ by Definition 5.3, hence $\rho(\eta; ev_\phi(c); H_2)$ is also valid by Lemma 3.2. But interpretations of effects are prefix-closed, so clearly $\hat{\eta}\, ev_\phi(c) \in [\![\rho(\eta; ev_\phi(c); H_2)]\!]$, so $\Pi(\phi(c), \hat{\eta}\, ev_\phi(c))$ holds by Definition 3.4, which is a contradiction. $\square$

Theorem 6.2 (*Type Safety*)
If $\Gamma, C, H \vdash e : \tau$ is valid for closed e then e does not go wrong.

Proof
Suppose on the contrary that $\epsilon, e \rightarrow^n \eta, e'$ with η, e' irreducible and e' not a value. Then $\Gamma, C, H' \vdash e' : \tau$ is derivable with $C \Vdash \eta; H' \sqsubseteq H$ by Lemma 6.13, induction on n, and Lemma 6.1. Now, since $\Gamma, C, H \vdash e : \tau$ is valid by assumption, therefore there exists a solution ρ of C such that $\rho(H)$ is valid by Definition 6.4. But since $[\![\rho(\eta; H')]\!] \subseteq [\![\rho(H)]\!]$ by Definition 5.3, therefore $\rho(\eta; H')$ is valid by Lemma 3.2. The contradiction follows by Theorem 6.1. $\square$

7 Subtyping Constraint Inference

Like the HM style type system of Fig. 4, typings specified by the subtyping system of Fig. 9 are inferable. We now give a type inference algorithm for constraint

$$
\begin{array}{rcll}
\mathit{hextract}\ C\ H & = & \mathit{hextract}_C(H, \varnothing) & \\[6pt]
\mathit{hextract}_C(\epsilon, hs) & = & \epsilon & \\
\mathit{hextract}_C(ev(\alpha), hs) & = & ev(\mathit{bounds}_C\ \alpha) & \\
\mathit{hextract}_C(h, hs) & = & h & h \in hs \\
\mathit{hextract}_C(h, hs) & = & \mu h.\mathit{hextract}_C((\mathit{bounds}_C\ h), hs \cup \{h\}) & h \notin hs \\
\mathit{hextract}_C(H_1; H_2, hs) & = & (\mathit{hextract}_C(H_1, hs)); (\mathit{hextract}_C(H_2, hs)) & \\
\mathit{hextract}_C(H_1 | H_2, hs) & = & (\mathit{hextract}_C(H_1, hs)) | (\mathit{hextract}_C(H_2, hs)) &
\end{array}
$$

$$\mathit{bounds}_C\ \beta \quad = \quad \tau_1 | \cdots | \tau_n \quad \text{where } \{\tau_1, \ldots, \tau_n\} = \{\tau \mid \tau \sqsubseteq \beta \in C \text{ and } \tau \notin \mathcal{V}_k\}$$

Fig. 14. *hextract*, *hextract*$_C$, and *bounds* functions

subtyping, and prove that it is sound and complete with respect to logical judgements. Furthermore, our completeness result yields a principal types property for λ_{trace} constraint subtyping. The inference algorithm and form of judgements are the same as for the HM style system, defined in Fig. 5. The difference is in the constraint solution technique. Rather than a unification-based solver as in in the HM style system, we use a *closure* technique. The technique is standard, with extensions to accommodate trace effects and singleton types. In order to exploit model checking techniques to verify inferred effects, we also develop an *effect extraction* algorithm to represent top-level effects in term form. Our technical development culminates in Corollary 7.1 and Lemma 7.22, formally establishing correctness of the composition of inference, closure, and effect extraction, but before the proofs we provide some initial definitions and intuitions.

Closure and consistency. In contrast to unification, which literally generates a solution to the constraint system, closure only verifies that a constraint is solvable. The closure algorithm is the iteration of a single-step rewrite relation defined in Fig. 12. Each rewrite rules makes explicit constraints that are implicit in existing ones– for example, observe that closure of a constraint $\{s_1\} \sqsubseteq \{s_2\}$ adds $s_1 \sqsubseteq s_2$ to the closure. When no new constraints can be added, closure terminates. Formally:

Definition 7.1 (*Constraint Closure*)
The rewrite relations $\rightsquigarrow_{close}$ and $\rightarrow_{close}$ are defined in Fig. 12. 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).

When closure terminates, all "atomic elements" of the constraint system are made evident. A simple structural consistency check, defined in Fig. 13, can then be used to ensure that all constraints entailed by the closure are solvable. For example, observe that clashing types in the sense of Lemma 6.2 cannot populate a consistent constraint. The definition of consistency exploits kinding rules extended to type terms to ensure consistency of constraints involving variables. Formally:

Definition 7.2
A type term τ has kind k, written $\tau : k$, iff the root constructor of any interpretation of τ has kind k as specified in Fig. 11.

Note that sanity conditions on interpretations imply that if $\beta \in \mathcal{V}_k$, then $\beta : k$.

Extracting effects from constraints. While demonstrating solvability is sufficient for constraints in general, model checking of inferred trace effects requires that they be presented in term form, not constraint form. Thus, we define another algorithm in Fig. 14, called *hextract*, that extracts a term representation of the inferred top level trace effect from a given constraint. The algorithm *hextract* is defined in a curried style, that integrates nicely with a combinator fixpoint construction proof technique, used for proving soundness of *hextract* in Sect. 7.3. The definition of $\mathit{hextract}_C$ specifies a family of functions, each element of which has is defined in terms of a fixed parameter C. Note that $\mathit{hextract}_C$ is not defined on the μ-bound form of trace effects, because this form does not occur in inferred types and constraints. However, $\mathit{hextract}_C$ does introduce μ-bindings, in order to solve recursively constrained effect variables.

Like the solution technique for the HM style system, *hextract* exploits the property that inferred constraints define a system of lower bounds on effect variables. However, in the subtyping constraint system, there is the added complication of singleton union types, as presented in Definition 6.1– although it turns out that singleton constraints also define a system of lower bounds on variables. The function bounds_C obtains the set of lower bounds of a given variable in C.

7.1 Soundness of Inference

Before delving into our main results, here are some auxiliary lemmas concerning variable freshness and constraint entailment that will be useful in later proofs.

Lemma 7.1
If $\Gamma, H \vdash_{\bar\beta} e : \tau/C$ is canonically derivable, then $\bar\beta \# \mathrm{fv}(\Gamma)$ and also $\mathrm{fv}(\tau, C, H) \subseteq \mathrm{fv}(\Gamma) \cup \bar\beta$.

Lemma 7.2
All of the following properties hold:

1. If $C \Vdash \tau' \sqsubseteq \tau$, then $\mathrm{fv}(\tau) \subseteq \mathrm{fv}(\tau', C)$.
2. $C \wedge D \Vdash D$.
3. If $C \Vdash \tau \sqsubseteq \tau'$ then $\psi(C) \Vdash \psi(\tau) \sqsubseteq \psi(\tau')$.

We now demonstrate soundness of type inference with respect to constraint subtyping– that is, we show that inferred judgements are derivable in the logical constraint subtyping system of Fig. 9. For this purpose it is necessary to convert constrained type schemes in inference environments to unconstrained type schemes in logical environments.

Definition 7.3

We define the following environment transformation function:

$$\begin{array}{rcl} \lfloor \varnothing \rfloor & = & \varnothing \\ \lfloor \Gamma; x : \forall\bar{\beta}.\tau/C \rfloor & = & \lfloor \Gamma \rfloor; x : \forall\bar{\beta}.\tau \end{array}$$

Now, soundness can be proved by a straightforward induction on inference derivations. The proof technique essentially rewrites inference derivations into logical derivations by pushing reconstructed constraints from the root to the leaves.

Lemma 7.3 (*Soundness of Subtype Inference*)
If $\Gamma, H \vdash_{\bar{\beta}} e : \tau/C$ is canonically derivable, then so is $\lfloor\Gamma\rfloor, C \wedge D, H \vdash e : \tau$ for any D.

Proof
By induction on the derivation of $\Gamma, H \vdash_{\bar{\beta}} e : \tau/C$ and case analysis on the last rule instance.

Case Var. In this case $e = x$ and $H = \epsilon$ and $\tau/C = \tau_0[\bar{\beta}/\bar{\beta}_0]/C_0[\bar{\beta}/\bar{\beta}_0]$ by definition of inference derivations, where:

$$\frac{\Gamma(x) = \forall\bar{\beta}_0.\tau_0/C_0}{\Gamma, \epsilon \vdash_{\bar{\beta}} x : \tau_0[\bar{\beta}/\bar{\beta}_0]/C_0[\bar{\beta}/\bar{\beta}_0]}$$

But since clearly $C_0[\bar{\beta}/\bar{\beta}_0] \wedge D \Vdash [\bar{\beta}/\bar{\beta}_0]$ for arbitrary D, therefore by an instance of Var and an instance of $\forall$-Elim we can derive:

$$\frac{\dfrac{\lfloor\Gamma\rfloor(x) = \forall\bar{\beta}_0.\tau_0}{\lfloor\Gamma\rfloor, C_0[\bar{\beta}/\bar{\beta}_0], \epsilon \vdash x : \forall\bar{\beta}_0.\tau_0} \qquad C_0[\bar{\beta}/\bar{\beta}_0] \Vdash [\bar{\beta}/\bar{\beta}_0]}{\lfloor\Gamma\rfloor, C_0[\bar{\beta}/\bar{\beta}_0], \epsilon \vdash x : \tau_0[\bar{\beta}/\bar{\beta}_0]}$$

Case Let. In this case $e = \mathsf{let}\, x = v \,\mathsf{in}\, e_0$ and $C = C_1 \wedge C_2$ and $\bar{\beta} = \bar{\beta}_1\bar{\beta}_2$ by definition of inference derivations, and we have:

$$\frac{\bar{\beta}_1 \# \bar{\beta}_2 \qquad \Gamma, \epsilon \vdash_{\bar{\beta}_1} v : \tau_1/C_1 \qquad \Gamma; \forall\bar{\beta}_1.\tau_1/C_1, \epsilon \vdash_{\bar{\beta}_2} e_0 : \tau_2/C_2}{\Gamma, H \vdash_{\bar{\beta}_1\bar{\beta}_2} \mathsf{let}\, x = v \,\mathsf{in}\, e_0 : \tau/C_1 \wedge C_2}$$

But by the induction hypothesis both of:

$$\lfloor\Gamma\rfloor, C_1 \wedge C_2, \epsilon \vdash v : \tau_1 \qquad\qquad \lfloor\Gamma\rfloor; x : \forall\bar{\beta}_1.\tau_1, C_1 \wedge C_2, H \vdash e_0 : \tau_2$$

are derivable, and by Lemma 7.1 we have that $\bar{\beta}_1 \# \mathrm{fv}(\Gamma)$, hence $\bar{\beta}_1 \# \mathrm{fv}(\lfloor\Gamma\rfloor)$, so by applications of $\forall$-Intro and Let:

$$\frac{\dfrac{\lfloor\Gamma\rfloor, C_1 \wedge C_2, \epsilon \vdash v : \tau_1 \qquad \bar{\beta}_1 \# \mathrm{fv}(\lfloor\Gamma\rfloor)}{\lfloor\Gamma\rfloor, C_1 \wedge C_2, \epsilon \vdash v : \forall\bar{\beta}_1.\tau_1} \qquad \lfloor\Gamma\rfloor; x : \forall\bar{\beta}_1.\tau_1, C_1 \wedge C_2, H \vdash e_0 : \tau_2}{\lfloor\Gamma\rfloor, C_1 \wedge C_2, H \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e_0 : \tau_2}$$

Case Fix. In this case:

$$e = \lambda_z x.e' \qquad \bar{\beta} = \bar{\beta}' \cup \{t, h, t'\} \qquad H = h \qquad \tau = t \xrightarrow{h} t' \qquad C = C' \wedge \tau' \sqsubseteq t' \wedge H \sqsubseteq h$$

where we have:

$$\frac{\Gamma; x : t; z : t \xrightarrow{h} t', H' \vdash_{\bar{\beta}} e' : \tau'/C}{\Gamma, \epsilon \vdash_{\bar{\beta}' \cup \{t,h,t'\}} \lambda_z x.e' : t \xrightarrow{h} t'/C' \wedge \tau' \sqsubseteq t' \wedge H' \sqsubseteq h}$$

by definition of inference derivations. By the induction hypothesis, the judgement:

$$\lfloor \Gamma \rfloor; x : t; z : t \xrightarrow{h} t', C, H' \vdash e' : \tau'$$

is derivable, so by applications of Sub and Fix:

$$\frac{\dfrac{\lfloor \Gamma \rfloor; x : t; z : t \xrightarrow{h} t', C, H' \vdash e' : \tau' \qquad C' \Vdash \tau' \sqsubseteq t' \qquad C' \Vdash H' \sqsubseteq h}{\lfloor \Gamma \rfloor; x : t; z : t \xrightarrow{h} t', C, h \vdash e' : t'}}{\lfloor \Gamma \rfloor, C, \epsilon \vdash \lambda_z x.e' : t \xrightarrow{h} t'}$$

Case Event. In this case we have:

$$e = ev(e') \qquad H = H'; ev(\alpha) \qquad \tau = unit \qquad C = C' \wedge \tau' \sqsubseteq \{\alpha\} \qquad \bar{\beta} = \bar{\beta}' \cup \{\alpha\}$$

by definition of inference derivations, and as the last derivation step:

$$\frac{\Gamma, H' \vdash_{\bar{\beta}'} e' : \tau'/C'}{\Gamma, H'; ev(\alpha) \vdash_{\bar{\beta}' \cup \{\alpha\}} ev(e') : unit/C' \wedge \tau' \sqsubseteq \{\alpha\}}$$

By the induction hypothesis, the judgement $\lfloor \Gamma \rfloor, C, H' \vdash e' : \tau$ is derivable, and since $C \Vdash \tau' \sqsubseteq \{\alpha\}$, by instances of Sub and Let:

$$\frac{\dfrac{\lfloor \Gamma \rfloor, C, H' \vdash e' : \tau \qquad C \Vdash \tau' \sqsubseteq \{\alpha\}}{\lfloor \Gamma \rfloor, C, H' \vdash e' : \{\alpha\}}}{\lfloor \Gamma \rfloor, C, H'; \{\alpha\} \vdash ev(e') : unit}$$

Case App. In this case by definition of inference derivations we have:

$$e = e_1 e_2 \qquad C = C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t \qquad H = H_1; H_2; h \qquad \tau = t$$

$$\bar{\beta} = \bar{\beta}_1 \bar{\beta}_2 \cup \{t, h\}$$

and as the last derivation step:

$$\frac{\bar{\beta}_1 \# \bar{\beta}_2 \qquad \Gamma, H_1 \vdash_{\bar{\beta}_1} e_1 : \tau_1/C_1 \qquad \Gamma, H_2 \vdash_{\bar{\beta}_2} e_2 : \tau_2/C_2}{\Gamma, H_1; H_2; h \vdash_{\bar{\beta}_1 \bar{\beta}_2 \cup \{t,h\}} e_1 e_2 : t/C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t}$$

But by the induction hypothesis, both of $\Gamma, H_1, C \vdash e_1 : \tau_1$ and $\Gamma, H_2, C \vdash e_2 : \tau_2$ are derivable, and since $C \Vdash \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t$ by Lemma 7.2, we can derive by applications of Sub and App:

$$\frac{\dfrac{\Gamma, H_1, C \vdash e_1 : \tau_1 \qquad C \Vdash \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t}{\Gamma, H_1, C \vdash e_1 : \tau_2 \xrightarrow{h} t} \qquad \Gamma, H_2, C \vdash e_2 : \tau_2}{\Gamma, H_1; H_2; h, C \vdash e_1 e_2 : t}$$

so this case holds.

Case If. In this case we have:

$$e = \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \qquad \tau = t \qquad H = H_1; H_2|H_3$$

$$C = C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \sqsubseteq \mathit{bool} \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t \qquad \bar{\beta} = \bar{\beta}_1\bar{\beta}_2\bar{\beta}_3 \cup \{t\}$$

by definition of inference derivations, where as the last derivation step:

$$\frac{\begin{array}{c}\bar{\beta}_1\#\bar{\beta}_2\#\bar{\beta}_3\\ \Gamma, H_1 \vdash_{\bar{\beta}_1} e_1 : \tau_1/C_1 \qquad \Gamma, H_2 \vdash_{\bar{\beta}_2} e_2 : \tau_2/C_2 \qquad \Gamma, H_3 \vdash_{\bar{\beta}_3} e_3 : \tau_3/C_3\end{array}}{\Gamma, H_1; H_2|H_3 \vdash_{\bar{\beta}} \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : t/C}$$

By the induction hypothesis, each of:

$$\lfloor\Gamma\rfloor, C, H_1 \vdash e_1 : \tau_1 \qquad \lfloor\Gamma\rfloor, C, H_2 \vdash e_2 : \tau_2 \qquad \lfloor\Gamma\rfloor, C, H_3 \vdash e_3 : \tau_3$$

is derivable, and by Lemma 7.2 and Lemma 6.4:

$$C \Vdash H_2 \sqsubseteq H_2|H_3 \qquad C \Vdash H_3 \sqsubseteq H_3|H_3 \qquad C \Vdash \tau_1 \sqsubseteq \mathit{bool} \qquad C \Vdash \tau_2 \sqsubseteq t$$

$$C \Vdash \tau_3 \sqsubseteq t$$

so by an application of Sub:

$$\frac{\lfloor\Gamma\rfloor, C, H_1 \vdash e_1 : \tau_1 \qquad C \Vdash \tau_1 \sqsubseteq \mathit{bool}}{\lfloor\Gamma\rfloor, C, H_1 \vdash e_1 : \mathit{bool}}$$

and also by an application of Sub:

$$\frac{\lfloor\Gamma\rfloor, C, H_2 \vdash e_2 : \tau_2 \qquad C \Vdash H_2 \sqsubseteq H_2|H_3 \qquad C \Vdash \tau_2 \sqsubseteq t}{\lfloor\Gamma\rfloor, C, H_2|H_3 \vdash e_2 : t}$$

and by another application of Sub:

$$\frac{\lfloor\Gamma\rfloor, C, H_3 \vdash e_3 : \tau_3 \qquad C \Vdash H_3 \sqsubseteq H_2|H_3 \qquad C \Vdash \tau_3 \sqsubseteq t}{\lfloor\Gamma\rfloor, C, H_2|H_3 \vdash e_3 : t}$$

so by an application of If:

$$\frac{\lfloor\Gamma\rfloor, C, H_1 \vdash e_1 : \mathit{bool} \qquad \lfloor\Gamma\rfloor, C, H_2|H_3 \vdash e_2 : t \qquad \lfloor\Gamma\rfloor, C, H_2|H_3 \vdash e_3 : t}{\lfloor\Gamma\rfloor, C, H_1; H_2|H_3 \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : t}$$

So this case holds. Case Const is immediate by nearly exact correspondence of the Const rule forms in the logical and inference systems, so the result follows. □

7.2 Completeness of Inference

As is usually the case, completeness is a more difficult manner than soundness, and the development requires more formal overhead. To wit, we extend the $\sqsubseteq$ relation to constrained types, to relate constrained type schemes (taken from logical judgements) with unconstrained type schemes (taken from inference judgements), and to relate constrained type environments with unconstrained type environments. This is all with a view to relating the generality of type inference judgements with logical type judgements. The extended $\sqsubseteq$ relation between type schemes also takes into account generalisation of type variables.

Definition 7.4
The relation $\sqsubseteq$ is extended to constrained types, type schemes, and type environments as follows:

$$\frac{C \Vdash C' \qquad C \Vdash \tau' \sqsubseteq \tau}{\tau'/C' \sqsubseteq \tau/C}$$

$$\frac{\forall[\bar{\tau}/\bar{\beta}] \ . \ C \Vdash [\bar{\tau}/\bar{\beta}] \Rightarrow \exists[\bar{\tau}'/\bar{\beta}'] \ . \ (\tau'/C')[\bar{\tau}'/\bar{\beta}'] \sqsubseteq (\tau[\bar{\tau}/\bar{\beta}]/C)}{C \Vdash \forall\bar{\beta}'.\tau'/C' \sqsubseteq \forall\bar{\beta}.\tau}$$

$$\frac{\forall x \in \mathrm{dom}(\Gamma) \ . \ C \Vdash \Gamma(x) \sqsubseteq \Gamma'(x)}{C \Vdash \Gamma \sqsubseteq \Gamma'}$$

Lemma 7.4
Definition 7.4 preserves reflexivity and transitivity of $\sqsubseteq$.

To formalise "maximum" generalisation of type schemes, we define the *gen* function, and to specify the generalisable variables in a judgement, we define the *genvars* function.

Definition 7.5
Define $\mathit{gen}(\Gamma, C, \forall\bar{\beta}.\tau) = \forall\bar{\beta}'.\tau$ where $\bar{\beta} = \mathrm{fv}(\tau, C) - \mathrm{fv}(\Gamma)$. Define $\mathit{genvars}(\Gamma, C, H, \forall\bar{\beta}.\tau) = \mathrm{fv}(C, H, \tau) - \mathrm{fv}(\Gamma)$.

Lemma 7.5
Given $\forall\bar{\beta}.\tau$, and Γ such that $\bar{\beta}\#\mathrm{fv}(\Gamma)$, then $C \Vdash \mathit{gen}(\Gamma, C, \forall\bar{\beta}.\tau) \sqsubseteq \forall\bar{\beta}.\tau$ for any C. Further, for all σ and x such that $\Gamma(x) = \sigma$, we have $\mathit{gen}(\Gamma, C, \sigma) = \sigma$.

Note that in the second part of the above definition, while extraneous bindings in σ may be eliminated in its generalisation, the equivalence still holds since we equate type schemes up to elimination of extraneous variables in Sect. 4.

Next, we formalise the principality relation on type judgements, relating inference judgements with logical judgements. We will show that most general types are inferred, according to this relation. The relation is parameterised by a substitution ψ, that may instantiate free variables in inference judgements for relation to logical judgements; intuitively, ψ imposes the unnecessary assumptions made in type environments in logical derivations.

Definition 7.6
Given $\mathit{genvars}(\Gamma, C, H, \forall\bar{\beta}_0.\tau) = \bar{\beta}$. The relation:

$$\Gamma', H' \vdash_{\bar{\beta}'} e : \tau'/C' \sqsubseteq_{\psi} \Gamma, C, H \vdash e : \forall\bar{\beta}_0.\tau$$

holds iff $C \Vdash \psi(\Gamma') \sqsubseteq \Gamma$, and for all $\psi'' = [\bar{\tau}/\bar{\beta}]$ where $C \Vdash [\bar{\tau}/\bar{\beta}]$ there exists $\psi' = [\bar{\tau}'/\bar{\beta}']$ such that:

$$C \Vdash \psi' \circ \psi(C') \qquad C \Vdash \psi' \circ \psi(H') \sqsubseteq \psi''(H) \qquad C \Vdash \psi' \circ \psi(\tau') \sqsubseteq \psi''(\tau)$$

The relation $\sqsubseteq_{\psi}$ takes into account generalisability of top-level effects, but for typing of values, for which effects are ϵ, it is simpler to speak in terms of maximal generalisation of types alone. The next Lemmas establish relevant properties.

Lemma 7.6
Supposing that $\mathrm{dom}(\psi) \subseteq \mathrm{fv}(\Gamma')$ and:

$$\Gamma', H' \vdash_{\bar{\beta}'} e : \tau'/C' \sqsubseteq_\psi \Gamma, C, H \vdash e : \sigma$$

Then $C' \Vdash \forall\bar{\beta}'.\psi(\tau'/C') \sqsubseteq \mathit{gen}(\Gamma, C, \sigma)$, and $H = \epsilon$ implies $H' = \epsilon$.

Lemma 7.7
Given ψ and judgements $\Gamma, C, \epsilon \vdash e : \sigma$ and $\Gamma', \epsilon \vdash_{\bar{\beta}'} e : \tau'/C'$ where $\mathrm{dom}(\psi) \subseteq \mathrm{fv}(\Gamma')$ and $C \Vdash \psi(\Gamma') \sqsubseteq \Gamma$. Then $C \Vdash \forall\bar{\beta}'.\psi(\tau'/C') \sqsubseteq \mathit{gen}(\Gamma, C, \sigma)$ implies:

$$\Gamma', \epsilon \vdash_{\bar{\beta}'} e : \tau'/C' \sqsubseteq_\psi \Gamma, C, \epsilon \vdash e : \sigma$$

Now, a few more auxiliary properties and definitions, including a definition of substitution restriction, before the main completeness result, which follows by induction on logical type derivations.

Lemma 7.8
If $C \Vdash \tau_1 \sqsubseteq \tau_1'$ and for all ψ_1 with $C \Vdash \psi_1$ there exists ψ_2 such that $C \Vdash \psi_2(\tau_2) \sqsubseteq \psi_1(\tau_1)$, then $C \Vdash \psi_2(\tau_2) \sqsubseteq \psi_1(\tau_1')$.

Proof
By Lemma 7.2 it is the case that $\psi_1(C) \Vdash \psi_1(\tau_1) \sqsubseteq \psi_1(\tau_1')$, so $C \Vdash \psi_1(\tau_1) \sqsubseteq \psi_1(\tau_1')$ by Lemma 6.4, thus $C \Vdash \psi_2(\tau_2) \sqsubseteq \psi_1(\tau_1')$ by Lemma 6.4. □

Definition 7.7
Given $\bar{\beta} \subseteq \mathrm{dom}(\psi)$; write $\psi\,|_{\bar{\beta}}$ to denote ψ' such that $\mathrm{dom}(\psi') = \bar{\beta}$ and $\psi'(\bar{\beta}) = \psi(\bar{\beta})$.

Lemma 7.9
For all $\bar{\beta} \subseteq \mathrm{dom}(\psi)$, if $C \Vdash \psi$ then $C \Vdash \psi\,|_{\bar{\beta}}$.

Lemma 7.10 (*Completeness of Subtype Inference*)
Given ψ, Γ', and derivable judgement $\Gamma, C, H \vdash e : \sigma$ such that $C \Vdash \psi(\Gamma') \sqsubseteq \Gamma$ and $\mathrm{dom}(\psi) \subseteq \mathrm{fv}(\Gamma')$. Then a judgement $\Gamma', H' \vdash_{\bar{\beta}'} e : \tau'/C'$ is canonically derivable, where:

$$\Gamma', H' \vdash_{\bar{\beta}'} e : \tau'/C' \sqsubseteq_\psi \Gamma, C, H \vdash e : \sigma$$

Proof
By induction on the derivation of $\Gamma, C, H \vdash e : \sigma$, and case analysis on the last rule instance in the derivation.

Case Var. In this case by definition of subtyping derivations, $H = \epsilon$ and $e = x$ and we can reconstruct:

$$\frac{\Gamma(x) = \sigma}{\Gamma, C, \epsilon \vdash x : \sigma}$$

But by Var in the inference system we can construct:

$$\frac{\Gamma'(x) = \forall\bar{\beta}'.\tau'/C'}{\Gamma', \epsilon \vdash_{\bar{\beta}''} x : \tau'[\bar{\beta}''/\bar{\beta}']/C'[\bar{\beta}''/\bar{\beta}']}$$

where $\bar{\beta}''$ are fresh by canonical form derivations. Since $\mathrm{dom}(\psi) = \mathrm{fv}(\Gamma')$, observe that:

$$\psi(\forall\bar{\beta}'.\tau'/C') = \forall\bar{\beta}'.\psi(\tau'/C')$$

Now, letting $\sigma = \forall\bar{\beta}.\tau$, take some arbitrary $[\bar{\tau}/\bar{\beta}]$ such that $C \Vdash [\bar{\tau}/\bar{\beta}]$; then we have by assumption and Definition 7.4:

$$\exists[\bar{\tau}'/\bar{\beta}'] \ . \ (\psi(\tau'/C'))[\bar{\tau}'/\bar{\beta}'] \sqsubseteq (\tau[\bar{\tau}/\bar{\beta}]/C)$$

Let:

$$\tau'' = \tau'[\bar{\beta}''/\bar{\beta}'] \qquad\qquad C'' = C'[\bar{\beta}''/\bar{\beta}']$$

Since $\bar{\beta}''$ are fresh, therefore $\bar{\beta}'\#\bar{\beta}''\#\mathrm{dom}(\psi)$, hence:

$$(\psi(\tau''/C''))[\bar{\tau}'/\bar{\beta}''] = (\psi(\tau'/C'))[\bar{\tau}'/\bar{\beta}']$$

so:

$$(\psi(\tau''/C''))[\bar{\tau}'/\bar{\beta}''] \sqsubseteq (\tau[\bar{\tau}/\bar{\beta}]/C)$$

Since $\forall\bar{\beta}.\tau = \mathit{gen}(\Gamma, C, \sigma)$ by Lemma 7.5, the result follows by Definition 7.4 and Lemma 7.7.

Case Let. In this case $e = \mathsf{let}\, x = v \,\mathsf{in}\, e'$ and $\sigma = \tau$ and by definition of subtyping derivations we can reconstruct:

$$\frac{\Gamma, C, \epsilon \vdash v : \sigma' \qquad \Gamma; x : \sigma', C, H \vdash e' : \tau}{\Gamma, C, H \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e' : \tau}$$

By the induction hypothesis , a judgement $\Gamma', \epsilon \vdash_{\bar{\beta}_1} v : \tau_1/C_1$ is canonically derivable, where there exists $\psi_1 = [\bar{\tau}_1/\bar{\beta}_1]$ such that:

$$C \Vdash \psi_1 \circ \psi(C_1)$$

and by Lemma 7.6:

$$C \Vdash \forall\bar{\beta}_1.\psi(\tau_1/C_1) \sqsubseteq \mathit{gen}(\Gamma, C, \sigma')$$

But $\mathrm{dom}(\psi)\#\bar{\beta}_1$ by Lemma 7.1, hence:

$$\forall\bar{\beta}_1.\psi(\tau_1/C_1) = \psi(\forall\bar{\beta}_1.\tau_1/C_1)$$

so by Lemma 7.4 and Lemma 7.5:

$$C \Vdash \psi(\forall\bar{\beta}_1.\tau_1/C_1) \sqsubseteq \sigma'$$

Thus, $C \Vdash \psi(\Gamma'; x : \forall\bar{\beta}_1.\tau_1/C_1) \sqsubseteq \Gamma; x : \sigma$ by Definition 7.4. By Lemma 7.1 and the induction hypothesis, letting $\bar{\beta} = \mathit{genvars}(\Gamma, C, H, \tau)$ we have that:

$$\Gamma'; x : \forall\bar{\beta}_1.\tau_1/C_1, H_2 \vdash_{\bar{\beta}_2} e' : \tau_2/C_2$$

is canonically derivable, where there exists ψ_2 with $\mathrm{dom}(\psi_2) = [\bar{\tau}_2/\bar{\beta}_2]$ such that for arbitrary $\psi'' = [\bar{\tau}/\bar{\beta}]$ with $C \Vdash [\bar{\tau}/\bar{\beta}]$:

$$C \Vdash \psi_2 \circ \psi(C_2) \qquad C \Vdash \psi_2 \circ \psi(\tau_2) \sqsubseteq \psi''(\tau) \qquad C \Vdash \psi_2 \circ \psi(H_2) \sqsubseteq \psi''(H)$$

Assume $\bar{\beta}_2 \# \bar{\beta}_1$ wlog, so that as an instance of Let:

$$\frac{\Gamma', \epsilon \vdash_{\bar{\beta}_1} v : \tau_1/C_1 \qquad \Gamma'; x : \forall\bar{\beta}_1.\tau_1/C_1, H_2 \vdash_{\bar{\beta}_2} e' : \tau_2/C_2}{\Gamma', H_2 \vdash_{\bar{\beta}_1\bar{\beta}_2} \mathsf{let}\, x = v \,\mathsf{in}\, e' : \tau_2/C_1 \wedge C_2}$$

Now, we have by assumption and Lemma 7.1 that $\bar{\beta}_2 \# \bar{\beta}_1 \# \mathrm{fv}(\Gamma)$, and also by Lemma 7.1:

$$\mathrm{fv}(\tau_1, C_1) \subseteq \mathrm{fv}(\Gamma) \cup \bar{\beta}_1 \qquad \mathrm{fv}(\tau_2, C_2, H_2) \subseteq \mathrm{fv}(\Gamma) \cup \bar{\beta}_2$$

so clearly:

$$\psi_1 \circ \psi(C_1) = \psi_1 \circ \psi_2 \circ \psi(C_1) \qquad \psi_2 \circ \psi(C_2) = \psi_1 \circ \psi_2 \circ \psi(C_2)$$

$$\psi_2 \circ \psi(\tau_2) = \psi_1 \circ \psi_2 \circ \psi(\tau_2) \qquad \psi_2 \circ \psi(H_2) = \psi_1 \circ \psi_2 \circ \psi(H_2)$$

therefore by the above:

$$C \Vdash \psi_1 \circ \psi_2 \circ \psi(C_1 \wedge C_2) \qquad C \Vdash \psi_1 \circ \psi_2 \circ \psi(\tau_2) \sqsubseteq \psi''(\tau)$$

$$C \Vdash \psi_1 \circ \psi_2 \circ \psi(H_2) \sqsubseteq \psi''(H)$$

and $\mathrm{dom}(\psi_1 \circ \psi_2) = \bar{\beta}_1\bar{\beta}_2$, so this case holds by Definition 7.6.

Case $\forall$-Intro. In this case by definition of subtyping derivations we have that $e = v$ and $\sigma = \forall\bar{\beta}.\tau$, and we can reconstruct:

$$\frac{\Gamma, C, \epsilon \vdash v : \tau \qquad \bar{\beta} \# \mathrm{fv}(\Gamma)}{\Gamma, C, \epsilon \vdash v : \forall\bar{\beta}.\tau}$$

By the induction hypothesis and Lemma 7.6 there exists derivable $\Gamma', \epsilon \vdash_{\bar{\beta}'} v : \tau'/C'$, with $C \Vdash \forall\bar{\beta}'.\psi(\tau'/C') \sqsubseteq \mathit{gen}(\Gamma, C, \tau)$. But $C \Vdash \mathit{gen}(\Gamma, C, \tau) \sqsubseteq \forall\bar{\beta}.\tau$ by Lemma 7.5, therefore by Lemma 7.4 we have $C \Vdash \forall\bar{\beta}'.\psi(\tau'/C') \sqsubseteq \forall\bar{\beta}.\tau$, so this case holds by Lemma 7.7.

Case $\forall$-Elim. In this case by definition of subtyping derivations we have that $e = v$ and $\sigma = \tau[\bar{\tau}/\bar{\beta}]$, and we can reconstruct:

$$\frac{\Gamma, C, \epsilon \vdash v : \forall\bar{\beta}.\tau \qquad C \Vdash [\bar{\tau}/\bar{\beta}]}{\Gamma, C, \epsilon \vdash v : \tau[\bar{\tau}/\bar{\beta}]}$$

By the induction hypothesis and Lemma 7.6 there exists derivable $\Gamma', \epsilon \vdash_{\bar{\beta}'} v : \tau'/C'$, with $C \Vdash \forall\bar{\beta}'.\psi(\tau'/C') \sqsubseteq \mathit{gen}(\Gamma, C, \forall\bar{\beta}.\tau)$. Let $\forall\bar{\beta}_0.\tau = \mathit{gen}(\Gamma, C, \forall\bar{\beta}.\tau)$ and $\bar{\beta}_1 = \{\beta | \beta \in \bar{\beta}_0 - \bar{\beta}\}$. Then $\tau[\bar{\tau}/\bar{\beta}] = \tau[\bar{\tau}/\bar{\beta}][\bar{\beta}_1/\bar{\beta}_1]$. Let $\psi_0 = [\bar{\tau}/\bar{\beta}][\bar{\beta}_1/\bar{\beta}_1]$ and let $\mathit{gen}(\Gamma, C, \tau[\bar{\tau}/\bar{\beta}]) = \forall\bar{\beta}_2.\tau[\bar{\tau}/\bar{\beta}]$. It is easy to show that $\bar{\beta}_2 \subseteq \bar{\beta}_1 \cup \mathrm{fv}(\bar{\tau})$; hence for arbitrary $[\bar{\tau}_2/\bar{\beta}_2]$ such that $C \Vdash [\bar{\tau}_2/\bar{\beta}_2]$, we have $\mathrm{dom}([\bar{\tau}_2/\bar{\beta}_2] \circ \psi_0) \subseteq \bar{\beta}_0$, hence there exists $\bar{\tau}_3$ such that $[\bar{\tau}_3/\bar{\beta}_0] = [\bar{\tau}_2/\bar{\beta}_2] \circ \psi_0$. Thus by Definition 7.4 and above facts there exists $[\bar{\tau}'/\bar{\beta}']$ such that the following entailments hold:

$$C \Vdash (\psi(C'))[\bar{\tau}'/\bar{\beta}'] \qquad C \Vdash (\psi(\tau'))[\bar{\tau}'/\bar{\beta}'] \sqsubseteq \tau[\bar{\tau}_3/\bar{\beta}_0]$$

the latter of which is to say by the above equivalences:

$$C \Vdash (\psi(\tau'))[\bar{\tau}'/\bar{\beta}'] \sqsubseteq (\tau[\bar{\tau}/\bar{\beta}])[\bar{\tau}_2/\bar{\beta}_2]$$

so this case holds by Definition 7.4 and Lemma 7.7.

Case EVENT. In this case $e = ev(e_0)$ and $\sigma = \mathit{unit}$ and $H = H_0; ev(s)$ by definition of subtyping derivations, where we can reconstruct the following rule instance:

$$\frac{\Gamma, C, H_0 \vdash e_0 : \{s\}}{\Gamma, C, H_0; ev(s) \vdash ev(e_0) : \mathit{unit}}$$

Let $\mathit{genvars}(\Gamma, C, H_0, \{s\}) = \bar{\beta}$. Then by the induction hypothesis there exists derivable $\Gamma', H' \vdash_{\bar{\beta}'} ev(e_0) : \tau'/C'$, where for all $\psi_0 = [\bar{\tau}/\bar{\beta}]$ such that $C \Vdash [\bar{\tau}/\bar{\beta}]$ there exists $\psi' = [\bar{\tau}'/\bar{\beta}']$ such that:

$$C \Vdash \psi' \circ \psi(C') \qquad C \Vdash \psi' \circ \psi(\tau') \sqsubseteq \psi_0(\{s\}) \qquad C \Vdash \psi' \circ \psi(H') \sqsubseteq \psi_0(H_0)$$

and by an instance of EVENT we can canonically infer:

$$\frac{\Gamma', H' \vdash_{\bar{\beta}'} ev(e_0) : \tau'/C'}{\Gamma', H'; ev(\alpha) \vdash_{\bar{\beta}' \cup \{\alpha\}} ev(e') : \mathit{unit}/C' \wedge \tau' \sqsubseteq \{\alpha\}}$$

Now, since α is canonically chosen fresh, by Lemma 7.1 $\alpha \# \bar{\beta}' \# \mathrm{dom}(\psi)$ and:

$$\mathrm{fv}(\tau', C', H') \subseteq \mathrm{dom}(\psi) \cup \bar{\beta}'$$

Therefore letting $\psi'' = \psi' \circ [\psi_0(s)/\alpha]$:

$$\psi'' \circ \psi(\alpha) = \psi_0(s) \qquad \psi'' \circ \psi(C') = \psi' \circ \psi(C') \qquad \psi'' \circ \psi(\tau') = \psi' \circ \psi(\tau')$$

$$\psi'' \circ \psi(H') = \psi' \circ \psi(H')$$

so by the above facts and Lemma 6.4:

$$C \Vdash \psi'' \circ \psi(C' \wedge \tau' \sqsubseteq \{\alpha\}) \qquad C \Vdash \psi'' \circ \psi(H'; ev(\alpha)) \sqsubseteq \psi_0(H_0; ev(s))$$

and $\mathrm{dom}(\psi'') = \bar{\beta}' \cup \{\alpha\}$, so this case holds by Definition 7.6.

Case FIX. In this case we have $e = \lambda_z x.e'$ and $H = \epsilon$ and $\tau = \tau_1 \xrightarrow{H'} \tau_2$ by definition of subtyping derivations, with the following as the last derivation step:

$$\frac{\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{H_1} \tau_2, C, H_1 \vdash e' : \tau_2}{\Gamma, C, \epsilon \vdash \lambda_z x.e' : \tau_1 \xrightarrow{H_1} \tau_2}$$

Let $\psi_0 = [\tau_1/t_1][\tau_2/t_2][H_1/h] \circ \psi$. Since t_1, t_2, h are chosen fresh in canonical derivations, therefore $C \Vdash \psi_0(t_1 \xrightarrow{h} t_2) \sqsubseteq \tau_1 \xrightarrow{H_1} \tau_2$ and by Definition 7.4:

$$C \Vdash \psi_0(\Gamma'; x : t_1; z : t_1 \xrightarrow{h} t_2) \sqsubseteq (\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{H_1} \tau_2)$$

Therefore, since $\mathit{genvars}(\Gamma; x : \tau_1; z : \tau_1 \xrightarrow{H_1} \tau_2, C, H_1, \tau_2) = \varnothing$, by the induction hypothesis a judgement of the following form is derivable:

$$\Gamma'; x : t_1; z : t_1 \xrightarrow{h} t_2, H' \vdash_{\bar{\beta}''} e' : \tau'/C'$$

where there exists $\psi' = [\bar{\tau}/\bar{\beta}'']$ such that:

$$C \Vdash \psi' \circ \psi_0(C') \qquad C \Vdash \psi' \circ \psi_0(H') \sqsubseteq H_1 \qquad C \Vdash \psi' \circ \psi_0(\tau') \sqsubseteq \tau_2$$

But letting $\psi'' = \psi' \circ [\tau_1/t_1][\tau_2/t_2][H_1/h]$, equivalently:

$$C \Vdash \psi'' \circ \psi(C') \qquad C \Vdash \psi'' \circ \psi(H') \sqsubseteq H_1 \qquad C \Vdash \psi'' \circ \psi(\tau') \sqsubseteq \tau_2$$

And freshness of $\{t_1, t_2, h\}$ means $\{t_1, t_2, h\} \# \bar{\beta}'' \# \mathrm{dom}(\psi)$, therefore $\psi'' \circ \psi(t_1) = \tau_1$ and $\psi'' \circ \psi(t_2) = \tau_2$ and $\psi'' \circ \psi(h_1) = H_1$, thus:

$$C \Vdash \psi'' \circ \psi(C' \wedge H' \sqsubseteq h \wedge \tau' \sqsubseteq t_2) \qquad C \Vdash \psi'' \circ \psi(t_1 \xrightarrow{h} t_2) \sqsubseteq \tau_1 \xrightarrow{H_1} \tau_2$$

and $\mathrm{dom}(\psi') = \bar{\beta}'' \cup \{t_1, t_2, h\}$. By an application of Fix:

$$\frac{\Gamma'; x : t_1; z : t_1 \xrightarrow{h} t_2, H' \vdash_{\bar{\beta}''} e' : \tau'/C'}{\Gamma', \epsilon \vdash_{\bar{\beta}'' \cup \{t_1,t_2,h\}} \lambda_z x.e' : t_1 \xrightarrow{h} t_2 / C' \wedge H' \sqsubseteq h \wedge \tau' \sqsubseteq t_2}$$

so this case holds by Definition 7.6.

Case App. In this case $e = e_1 e_2$ and $H = H_1; H_2; H_3$ by definition of subtyping derivations, and as the last step in the derivation:

$$\frac{\Gamma, C, H_1 \vdash e_1 : \tau' \xrightarrow{H_3} \tau \qquad \Gamma, C, H_2 \vdash e_2 : \tau'}{\Gamma, C, H_1; H_2; H_3 \vdash e_1 e_2 : \tau}$$

Let:

$$\mathit{genvars}(\Gamma, C, H_1, \tau' \xrightarrow{H_3} \tau) = \bar{\beta} \qquad \mathit{genvars}(\Gamma, C, H_2, \tau') = \bar{\beta}'$$

$$\mathit{genvars}(\Gamma, C, H_1; H_2; H_3, \tau) = \bar{\beta}''$$

Let ψ' be an arbitrary substitution such that $\mathrm{dom}(\psi') = \bar{\beta}\bar{\beta}'$ and $C \Vdash \psi'$, and let:

$$\psi_{\bar{\beta}} = \psi'|_{\bar{\beta}} \qquad \psi_{\bar{\beta}'} = \psi'|_{\bar{\beta}'} \qquad \psi_{\bar{\beta}''} = \psi'|_{\bar{\beta}''}$$

Then by Lemma 7.9 and the induction hypothesis there exists a canonically derivable judgement $\Gamma', H_1' \vdash_{\bar{\beta}_1} e_1 : \tau_1/C_1$ and substitution $\psi_1 = [\bar{\tau}_1/\bar{\beta}_1]$ such that:

$$C \Vdash \psi_1 \circ \psi(C_1) \qquad C \Vdash \psi_1 \circ \psi(\tau_1) \sqsubseteq \psi_{\bar{\beta}}(\tau' \xrightarrow{H_3} \tau) \qquad C \Vdash \psi_1 \circ \psi(H_1') \sqsubseteq \psi_{\bar{\beta}}(H_1)$$

Also by Lemma 7.9 and the induction hypothesis there exists canonically derivable $\Gamma', H_2' \vdash_{\bar{\beta}_2} e_2 : \tau_2/C_2$ and substitution $\psi_2 = [\bar{\tau}_2/\bar{\beta}_2]$ such that:

$$C \Vdash \psi_2 \circ \psi(C_2) \qquad C \Vdash \psi_2 \circ \psi(\tau_2) \sqsubseteq \psi_{\bar{\beta}'}(\tau') \qquad C \Vdash \psi_2 \circ \psi(H_2') \sqsubseteq \psi_{\bar{\beta}'}(H_2)$$

Note that:

$$\psi_{\bar{\beta}}(H_1) = \psi_{\bar{\beta}''}(H_1) \qquad \psi_{\bar{\beta}'}(H_2) = \psi_{\bar{\beta}''}(H_2) \qquad \psi_{\bar{\beta}}(H_3) = \psi_{\bar{\beta}''}(H_3)$$

$$\psi_{\bar{\beta}}(\tau) = \psi_{\bar{\beta}''}(\tau) \qquad \psi_{\bar{\beta}}(\tau') = \psi_{\bar{\beta}'}(\tau')$$

Now, let $\psi_3 = \psi_2 \circ [\psi_{\bar{\beta}''}(\tau)/t][\psi_{\bar{\beta}''}(H_3)/h]$. Taking $\bar{\beta}_1 \# \bar{\beta}_2$ wlog, by Lemma 7.1 it is the case that $\bar{\beta}_1 \# \bar{\beta}_2 \# \mathrm{dom}(\psi)$ and:

$$\mathrm{fv}(\tau_1, H_1', C_1) = \mathrm{fv}(\Gamma) \cup \bar{\beta}_1 \qquad \mathrm{fv}(\tau_2, H_2', C_2) = \mathrm{fv}(\Gamma) \cup \bar{\beta}_2$$

therefore:

$$\psi_3 \circ \psi(\tau_1) = \psi_1 \circ \psi(\tau_1) \qquad \psi_3 \circ \psi(H_1) = \psi_1 \circ \psi(H_1) \qquad \psi_3 \circ \psi(C_1) = \psi_1 \circ \psi(C_1)$$

$$\psi_3 \circ \psi(\tau_2) = \psi_2 \circ \psi(\tau_2) \qquad \psi_3 \circ \psi(H_2) = \psi_2 \circ \psi(H_2) \qquad \psi_3 \circ \psi(t) = \psi_{\bar{\beta}''}(\tau)$$

$$\psi_3 \circ \psi(h) = \psi_{\bar{\beta}''}(H_3)$$

Combining these equivalences with preceding facts and Lemma 6.4, we have:

$$C \Vdash \psi_3 \circ \psi(H_1'; H_2'; h) \sqsubseteq \psi_{\bar\beta''}(H_1; H_2; H_3) \qquad C \Vdash \psi_3 \circ \psi(t) \sqsubseteq \psi_{\bar\beta''}(\tau)$$

We can also assert:

$$C \Vdash \psi_3 \circ \psi(C_1 \wedge C_2) \qquad C \Vdash \psi_{\bar\beta}(\tau' \xrightarrow{H_3} \tau) \sqsubseteq \psi_3 \circ \psi(\tau_2 \xrightarrow{h} t)$$

$$C \Vdash \psi_3 \circ \psi(\tau_1) \sqsubseteq \psi_{\bar\beta}(\tau' \xrightarrow{H_3} \tau)$$

so by transitivity of $\sqsubseteq$, etc., we have:

$$C \Vdash \psi_3 \circ \psi(C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t)$$

Finally, by App we can derive:

$$\frac{\Gamma', H_1' \vdash_{\bar\beta_1} e_1 : \tau_1/C_1 \qquad \Gamma', H_2' \vdash_{\bar\beta_2} e_2 : \tau_2/C_2}{\Gamma', H_1'; H_2'; h \vdash_{\bar\beta_1\bar\beta_2\{t,h\}} e_1 e_2 : t/C_1 \wedge C_2 \wedge \tau_1 \sqsubseteq \tau_2 \xrightarrow{h} t}$$

and since $\mathrm{dom}(\psi_3) = \bar\beta_1\bar\beta_2\,\{t,h\}$, this case holds.

Case If. In this case $e = \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3$, and $H = H_1; H_2$, and as the last step in the derivation:

$$\frac{\Gamma, C, H_1 \vdash e_1 : \mathit{bool} \qquad \Gamma, C, H_2 \vdash e_2 : \tau \qquad \Gamma, C, H_2 \vdash e_3 : \tau}{\Gamma, C, H_1; H_2 \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau}$$

Let:

$$\mathit{genvars}(\Gamma, C, H_1, \mathit{bool}) = \bar\beta \qquad \mathit{genvars}(\Gamma, C, H_2, \tau) = \bar\beta'$$

and taking arbitrary $\mathrm{dom}(\psi') = \bar\beta\bar\beta'$ such that $C \Vdash \psi'$, define $\psi_{\bar\beta} = \psi' \mid_{\bar\beta}$ and $\psi_{\bar\beta'} = \psi' \mid_{\bar\beta'}$. Then by Lemma 7.9 and the induction hypothesis, there exist derivable judgements:

$$\Gamma', H_1' \vdash_{\bar\beta_1} e_1 : \tau_1/C_1 \qquad \Gamma', H_2' \vdash_{\bar\beta_2} e_2 : \tau_2/C_2 \qquad \Gamma', H_3' \vdash_{\bar\beta_3} e_3 : \tau_3/C_3$$

and substitutions ψ_1, ψ_2, ψ_3 with $\mathrm{dom}(\psi_1) = \bar\beta_1$, $\mathrm{dom}(\psi_2) = \bar\beta_2$, $\mathrm{dom}(\psi_3) = \bar\beta_3$ such that:

$$C \Vdash \psi_1 \circ \psi(C_1) \qquad C \Vdash \psi_1 \circ \psi(H_1') \sqsubseteq \psi_{\bar\beta}(H_1) \qquad C \Vdash \psi_1 \circ \psi(\tau_1) \sqsubseteq \psi_{\bar\beta}(\mathit{bool})$$

$$C \Vdash \psi_2 \circ \psi(C_2) \qquad C \Vdash \psi_2 \circ \psi(H_2') \sqsubseteq \psi_{\bar\beta'}(H_2) \qquad C \Vdash \psi_2 \circ \psi(\tau_2) \sqsubseteq \psi_{\bar\beta'}(\tau)$$

$$C \Vdash \psi_3 \circ \psi(C_3) \qquad C \Vdash \psi_3 \circ \psi(H_3') \sqsubseteq \psi_{\bar\beta'}(H_2) \qquad C \Vdash \psi_3 \circ \psi(\tau_3) \sqsubseteq \psi_{\bar\beta'}(\tau)$$

Assume $\bar\beta_1 \# \bar\beta_2 \# \bar\beta_3$ wlog. Note the following equivalences:

$$\psi_{\bar\beta}(\mathit{bool}) = \mathit{bool} \qquad \psi_{\bar\beta'}(\tau) = \psi'(\tau) \qquad \psi_{\bar\beta}(H_1) = \psi'(H_1) \qquad \psi_{\bar\beta'}(H_2) = \psi'(H_2)$$

and let $\psi_4 = \psi_1 \circ \psi_2 \circ \psi_3 \circ [\tau/t]$. Then since $\bar\beta_1 \# \bar\beta_2 \# \bar\beta_3 \# t \# \mathrm{dom}(\psi)$ by assumption, canonical freshness of t, and Lemma 7.1, we have $\psi_4(t) = \tau$, and for all $i \in [1..3]$:

$$\psi_i \circ \psi(C_i) = \psi_4 \circ \psi(C_i) \qquad \psi_i \circ \psi(H_i') = \psi_4 \circ \psi(H_i') \qquad \psi_i \circ \psi(\tau_i) = \psi_4 \circ \psi(\tau_i)$$

By the above facts and equivalences and Lemma 6.4 we can then assert:

$$C \Vdash \psi_4 \circ \psi(t) \sqsubseteq \psi'(\tau) \qquad C \Vdash \psi_4 \circ \psi(H_1'; H_2' | H_3') \sqsubseteq \psi'(H_1; H_2)$$

$$C \Vdash \psi_4 \circ \psi(\tau_1) \sqsubseteq \mathit{bool} \qquad C \Vdash \psi_4 \circ \psi(\tau_2 \sqsubseteq t) \qquad C \Vdash \psi_4 \circ \psi(\tau_3 \sqsubseteq t)$$

$$C \Vdash \psi_4 \circ \psi(C_1 \wedge C_2 \wedge C_3)$$

and letting $C' = C_1 \wedge C_2 \wedge C_3 \wedge \tau_1 \sqsubseteq \mathit{bool} \wedge \tau_2 \sqsubseteq t \wedge \tau_3 \sqsubseteq t$, observe $C \Vdash \psi_4 \circ \psi(C')$, and by If we can derive:

$$\frac{\begin{array}{c}\bar{\beta}_1 \# \bar{\beta}_2 \# \bar{\beta}_3 \qquad \bar{\beta}_4 = \bar{\beta}_1 \bar{\beta}_2 \bar{\beta}_3 \cup \{t\} \\ \Gamma', H_1' \vdash_{\bar{\beta}_1} e_1 : \tau_1 / C_1 \qquad \Gamma', H_2' \vdash_{\bar{\beta}_2} e_2 : \tau_2 / C_2 \qquad \Gamma', H_3' \vdash_{\bar{\beta}_3} e_3 : \tau_3 / C_3\end{array}}{\Gamma', H_1'; H_2' | H_3' \vdash_{\bar{\beta}_4} \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : t / C'}$$

Therefore since $\mathrm{dom}(\psi_4) = \bar{\beta}_1 \bar{\beta}_2 \bar{\beta}_3 \cup \{t\}$, this case holds.

Case Sub. In this case by definition of subtyping derivations, we have as the last step in the derivation:

$$\frac{\Gamma, C, H' \vdash e : \tau' \qquad C \Vdash \tau' \sqsubseteq \tau \qquad C \Vdash H' \sqsubseteq H}{\Gamma, C, H \vdash e : \tau}$$

Letting $\mathit{genvars}(\Gamma, C, H', \tau') = \bar{\beta}$ and taking arbitrary ψ' such that $\mathrm{dom}(\psi') = \bar{\beta}$ and $C \Vdash \psi'$, by the induction hypothesis there exists derivable $\Gamma', C_0 \vdash_{\bar{\beta}_0} e : \tau_0 / C_0$ and substitution ψ_0 such that:

$$C \Vdash \psi_0 \circ \psi(C_0) \qquad C \Vdash \psi_0 \circ \psi(H_0) \sqsubseteq \psi'(H') \qquad C \Vdash \psi_0 \circ \psi(\tau_0) \sqsubseteq \psi'(\tau')$$

But by Lemma 7.8 we have:

$$C \Vdash \psi_0 \circ \psi(H_0) \sqsubseteq \psi'(H) \qquad C \Vdash \psi_0 \circ \psi(\tau_0) \sqsubseteq \psi'(\tau)$$

and since $\mathit{genvars}(\Gamma, C, H, \tau) = \mathit{genvars}(\Gamma, C, H, \tau)$ by Lemma 7.2, this case holds. Case Const follows trivially, so the result follows. □

7.3 Closure, Consistency, and hextraction

Now we prove correctness of the closure, consistency, and effect extraction techniques. Since our closure and consistency algorithms are standard with respect to type constraints, we mainly appeal to existing results, especially those in (Palsberg & O'Keefe, 1995), to establish correctness in that regard. The focus of our proof development is on satisfaction of singleton and effect constraints, and extraction of effect terms from constraints via *hextract*. In fact, correctness of *hextract* provides a constructive proof of the satisfiability of closed, consistent effect constraints.

We begin by observing that closure does not change the meaning of constraints, i.e. a constraint is logically equivalent to its closure:

Lemma 7.11
$C = \mathit{close}(C)$.

One benefit of closure is that it provides a way to split constraints into distinct type, effect, and singleton components, the solutions for which can be composed to solve the entire constraint. We formalise this with the following definition and Lemma, where $f\,|_S$ denotes the restriction of the domain of a function f to S as usual.

Definition 7.8
For each kind k, the k *component* of a constraint C is a constraint D such that:

$$\mathit{set}(D) = \{\tau_1 \sqsubseteq \tau_2 \mid \tau_1 \sqsubseteq \tau_2 \in C \text{ and } \tau_1, \tau_2 : k\}$$

Given that D is the k component of C, the constraint C has a k *solution* iff there exists ρ such that $\rho \Vdash D$.

Lemma 7.12
Suppose C is closed, and ρ_1, ρ_2, and ρ_3 are *Type*, *Sing*, and *Eff* solutions of C, respectively. Then:

$$(\rho_1\,|_{\mathcal{V}_{\mathit{Type}}}) \circ (\rho_2\,|_{\mathcal{V}_{\mathit{Sing}}}) \circ (\rho_3\,|_{\mathcal{V}_{\mathit{Eff}}}) \Vdash C$$

The following Lemma establishes the crucial property that effect and singleton constraints form a system of lower bounds on variables. This property implies that any consistent closure has a singleton and effect solution. It also suggests a technique for deriving term representations of singletons and effects, by joining lower bounds of variables. This is not just convenient, but demonstrates that while effect constraint solution is not decidable in general, it is so for the restricted form generated by inference.

Lemma 7.13
Suppose $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable and $\tau_1 \sqsubseteq \tau_2 \in \mathit{close}(C)$. If $\tau_2 : \mathit{Sing}$ then τ_2 is a singleton variable α, and if $\tau_2 : \mathit{Eff}$ then τ_2 is an effect variable h.

Now, we show that the singleton component of a closed, consistent constraint has a solution, that can be obtained via the bounds_C function.

Definition 7.9
Suppose f is a partial mapping from type variables to types. Write $f \preccurlyeq \rho$ iff there exists ρ' such that $\rho = \rho' \circ f$.

Lemma 7.14
Suppose $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable and $\mathit{close}(C)$ is consistent. Then there exists a *Sing* solution ρ of $\mathit{close}(C)$ such that $\mathit{bounds}_{\mathit{close}(C)}\,|_{\mathcal{V}_{\mathit{Sing}}} \preccurlyeq \rho$.

Next, we show that the effect component of a closed, consistent constraint has a solution, that can be obtained via the *hextract* function. We show that this is the case constructively, by demonstrating that *hextract* is defined on top-level effects, and that *hextract* is a solution of given constraints. The first step is to show that top-level effects have concrete lower bounds, so that their term form is closed.

Definition 7.10
Let C be closed and consistent. Then τ_1 is a *component of* τ_2 *in* C iff τ_1 is a subterm of τ_2, or there exists a subterm τ_1' of τ_1 such that $\tau_2' \sqsubseteq \tau_1' \in C$ and τ_2 is a component of τ_2' in C.

$$
\begin{array}{rcll}
HX\ f\ C\ \epsilon & = & \epsilon & \\
HX\ f\ C\ ev(\alpha) & = & ev(bounds_C\ \alpha) & \\
HX\ f\ C\ h & = & f\ C\ (bounds_C\ h) & \text{if } bounds_C\ h \text{ defined} \\
HX\ f\ C\ h & = & f\ C\ h & \text{if } bounds_C\ h \text{ not defined} \\
HX\ f\ C\ H_1; H_2 & = & (f\ C\ H_1); (f\ C\ H_2) & \\
HX\ f\ C\ H_1|H_2 & = & (f\ C\ H_1)|(f\ C\ H_2) &
\end{array}
$$

Fig. 15. The *HX* combinator

Definition 7.11
Let C be closed and consistent. Then the *abstract components of* τ *in* C, denoted $abstract(\tau, C)$ are the variable components of τ that have no lower bound in C. More precisely:

$$abstract(\tau, C) = \{\beta \mid \beta \text{ is a component of } \tau \text{ in } C \wedge \neg\exists\tau'.\tau' \sqsubseteq \beta \in C\}$$

On the basis of these definitions, we can now show that top-level effects have concrete lower bounds in inferred constraints, meaning that *hextract* is defined and closed on top-level effects The result follows by a tedious and unilluminating induction on derivations. Essentially, the property holds because any effect variable occurring in a top-level effect is the upper bound of a function effect by definition of inference, and since top-level expressions are closed, the function must be a closed lambda abstraction, which have defined, concrete effects.

Lemma 7.15 (*Top-level effects are concrete*)
Suppose $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable and $close(C)$ is consistent. Then H has no abstract components in $close(C)$.

Lemma 7.16
Suppose $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable and $close(C)$ is consistent. Then $(hextract\ close(C)\ H)$ is defined and closed.

Having established that *hextract* is defined on top-level effects, we now show that *hextract* is a solution of given constraints. The recursive definition of *hextract* makes a direct proof of this property difficult. Instead, we demonstrate the property via a combinator fixpoint construction. We define a non-recursive combinator *HX* in Fig. 15, that characterises the solution of closed, consistent effect constraints, and then show that *hextract* is a fixpoint of this combinator. The first step is to show that *HX* correctly characterises solutions of closed, consistent effect constraints, as follows.

Lemma 7.17
Suppose $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable and $close(C)$ is consistent. If f is a fixpoint of HX, then there exists an *Eff* solution ρ of $close(C)$ such that $(f\ \ close(C)) \preccurlyeq \rho$.

Proof

Let D be the *Eff* component of $\mathit{close}(C)$, and suppose that f is a fixpoint of HX. Let ρ' be a *Sing* solution of $\mathit{close}(C)$, which must exist by Lemma 7.14, and let:

$$g = \rho' \circ (f\ \mathit{close}(C))$$

To proceed, we number the clauses of HX from top to bottom, starting with 1. Since we assume that f is a fixpoint of HX, clauses 1-2 and 5-6 establish the following equalities:

$$g(\epsilon) = \epsilon \qquad g(ev(\alpha)) = \rho'(ev(\mathit{bounds}_C\ \alpha)) \qquad g(H_1;H_2) = g(H_1);g(H_2)$$

$$g(H_1|H_2) = g(H_1)|g(H_1)$$

Thus, g is an interpretation, as defined in Fig. 6. Furthermore, clause 3 establishes that for all $H \sqsubseteq h \in D$, it is the case that $g(H) \preccurlyeq g(h)$, meaning that g satisfies every constraint in D by Lemma 7.13, and is therefore a solution of D. Letting $\rho = g$, the result follows by Definition 7.9. $\square$

It would now be possible to show that any closed, consistent effect constraint has a solution, by showing that HX is monotonic over a partial ordering of solutions, since classic results in (Tarski, 1955) would then imply that HX has a fixpoint. Instead, we prove that *hextract* is a fixpoint of HX in Lemma 7.17, establishing both correctness of *hextract* and existence of a solution of closed consistent effect constraints. We first prove the essential auxiliary Lemma for proving the tricky μ case of Lemma 7.17, and then Lemma 7.17 itself.

Lemma 7.18
Let $h \notin hs$ and $H' = \mathit{hextract}_C(h, hs)$. Then:

$$\mathit{hextract}_C(H, hs \cup \{h\})[H'/h] = \mathit{hextract}_C(H, hs)$$

Proof
The proof proceeds by induction on the call tree of $\mathit{hextract}_C$ and case analysis on H. In case $H = \epsilon$ or $H = ev(\alpha)$ the result is immediate. In case $H = H_1|H_2$ or $H_1;H_2$ the result follows in a straightforward manner via the induction hypothesis. The more interesting case is that in which H is an effect variable h'. Here we consider subcases $h' \neq h$ and $h' = h$, where the former breaks down into subcases with $h' \in hs$ on the one hand and $h' \notin hs$ on the other. If $h' \neq h$ and $h' \in hs$, then by definition of $\mathit{hextract}_C$:

$$\mathit{hextract}_C(h', hs \cup \{h\})[H'/h] = \mathit{hextract}_C(h', hs) = h'$$

so the result follows.

On the other hand, if $h' \neq h$ and $h' \notin hs$, then by definition of $\mathit{hextract}_C$:

$$\mathit{hextract}_C(h', hs \cup \{h\})[H'/h] = (\mu h'.\mathit{hextract}_C((\mathit{bounds}_C\ h'), hs \cup \{h, h'\}))[H'/h]$$

and:

$$\mathit{hextract}_C(h', hs) = \mu h'.\mathit{hextract}_C((\mathit{bounds}_C\ h'), hs \cup \{h'\})$$

Let $H'' = \mathit{hextract}_C((\mathit{bounds}_C\ h'), hs \cup \{h, h'\})$. Since $h' \neq h$ by assumption:

$$(\mu h'.H'')[H'/h] = \mu h'.(H''[H'/h])$$

and by the induction hypothesis:

$$\mathit{hextract}_C((\mathit{bounds}_C\ h'), \mathit{hs} \cup \{h'\}) = H''[H'/h]$$

Hence:

$$\mathit{hextract}_C(h', \mathit{hs} \cup \{h\})[H'/h] = \mathit{hextract}_C(h', \mathit{hs}) = \mu h'.(H''[H'/h])$$

so the result follows.

Now, consider the subcase $h' = h$. Then by the definition of $\mathit{hextract}_C$:

$$\mathit{hextract}_C(h, \mathit{hs} \cup \{h\}) = h$$

and clearly:

$$\mathit{hextract}_C(h, \mathit{hs} \cup \{h\})[H'/h] = H'$$

and $\mathit{hextract}_C(H, \mathit{hs}) = H'$ in this case, so the result follows. $\square$

Lemma 7.19
hextract is a fixpoint of HX.

Proof
We begin by noting the obvious point $\mathit{hextract}\ C\ H = \mathit{hextract}_C(H, \varnothing)$. To prove the result, it suffices to show that for all C and for all H in the domain of $HX\ f\ C$ we have:

$$HX\ \mathit{hextract}\ C\ H = \mathit{hextract}\ C\ H$$

The proof proceeds by case analysis on H.

Case $H = \epsilon$. By definition of HX and *hextract*:

$$HX\ \mathit{hextract}\ C\ \epsilon = \mathit{hextract}\ C\ \epsilon = \epsilon$$

so this case holds.

Case $H = \mathit{ev}(\alpha)$. By definition of HX and *hextract*:

$$HX\ \mathit{hextract}\ C\ \mathit{ev}(\alpha) = \mathit{hextract}\ C\ \mathit{ev}(\alpha) = \mathit{ev}(\mathit{bounds}_C\ \alpha)$$

so this case holds.

Case $H = (H_1|H_2)$. By definition of HX:

$$HX\ \mathit{hextract}\ C\ (H_1|H_2) = (\mathit{hextract}\ C\ H_1)|(\mathit{hextract}\ C\ H_2)$$

and by definition of *hextract*:

$$\mathit{hextract}\ C\ (H_1|H_2, C) = \mathit{hextract}(H_1, C)|\mathit{hextract}(H_2, C)$$

so this case holds. Case $H = H_1; H_2$ follows similarly.

Case $H = h$. By definition of HX:

$$HX\ \mathit{hextract}\ C\ h = \mathit{hextract}\ C\ (\mathit{bounds}_C\ h)$$

and by definition of *hextract*:

$$\mathit{hextract}\ C\ h = \mathit{hextract}_C(h, \varnothing) = \mu h.\mathit{hextract}_C((\mathit{bounds}_C\ h), \{h\})$$

Now, let $H' = \mu h.\mathit{hextract}_C(\mathit{bounds}_C(h), \{h\})$. By Lemma 7.18 we have:

$$\mathit{hextract}_C((\mathit{bounds}_C\ h), \varnothing) = (\mathit{hextract}_C(\mathit{bounds}_C(h), \{h\}))[H'/h]$$

and by properties of trace effect equivalence noted in Lemma 3.1:

$$\rho((\mathit{hextract}_C((\mathit{bounds}_C\ h), \{h\}))[H'/h]) = \rho(H')$$

so the result follows in this case, which was the last to be proven. □

Having shown that *hextract* is a fixpoint of HX, we can immediately demonstrate that *hextract* implements a solution of given constraints when applied to top-level effects.

Lemma 7.20 (*Correctness of hextract*)
Suppose $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable and $\mathit{close}(C)$ is consistent. Then C has an *Eff* solution ρ such that $(\mathit{hextract}\ \mathit{close}(C)) \preccurlyeq \rho$.

Proof
By Lemma 7.19, *hextract* is a fixpoint of HX. The result follows immediately by Lemma 7.17. □

Composition of the above Lemmas establishing solvability of singleton and effect constraints, with standard results for closure and consistency of type constraints, allows us to obtain that consistency of closure is equivalent to satisfiability.

Lemma 7.21 (*Correctness of Closure*)
If $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable, then C has a solution iff $\mathit{close}(C)$ is consistent.

Proof
That solvability of C entails consistency of C follows via reductio ad absurdum, since if we assume on the contrary that $\mathit{close}(C)$ is inconsistent, this implies that $\mathit{close}(C)$ contains a clashing constraint and thus the contradiction by Lemma 6.2. The other direction proceeds as follows. By Lemma 7.14, there exists a *Sing* solution of $\mathit{close}(C)$. By Lemma 7.20, there exists an *Eff* solution of $\mathit{close}(C)$. Given this, we can show that $\mathit{close}(C)$ has a *Type* solution by appeal to standard results, particularly those in (Palsberg & O'Keefe, 1995), since our closure and consistency techniques for *Type* constraints are standard. The result follows by Lemma 7.12 and Lemma 7.11. □

Finally, we can tie all the results together to state high-level properties of the system. Since we have demonstrated completeness of inference in Lemma 7.10 and have shown that consistent closure is equivalent to satisfiability in Lemma 7.21, we immediately obtain a completeness result for inference.

Corollary 7.1 (*Implementation Completeness*)
If a closed expression e is typable in the system of Fig. 9, then a judgement $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable and $\mathit{close}(C)$ is consistent.

However, since we have only shown that *hextract* is a fixpoint of HX, not the *least* fixpoint, the best we can demonstrate for the algorithm is soundness. We informally

$$\mathtt{S} \quad ::= \quad \mathrm{nil} \mid \mathtt{S} :: \eta \qquad \textit{history stacks}$$

$$\begin{array}{rcll}
\mathtt{S}, (\lambda_z x.e)v & \rightsquigarrow & \mathtt{S} :: \epsilon, \cdot e[v/x][\lambda_z x.e/z]\cdot & (\beta) \\
\mathtt{S} :: \eta, \mathit{ev}(c) & \rightsquigarrow & \mathtt{S} :: \eta; \mathit{ev}(c), () & (\mathit{event}) \\
\mathtt{S} :: \eta, \mathit{ev}_\phi(c) & \rightsquigarrow & \mathtt{S} :: \eta; \mathit{ev}_\phi(c), () & (\mathit{check}) \\
 & & \text{if } \Pi(\phi(c), (\widehat{\mathtt{S} :: \eta})\mathit{ev}_\phi(c)) & \\
\mathtt{S} :: \eta, \cdot v \cdot & \rightsquigarrow & \mathtt{S}, v & (\mathit{pop})
\end{array}$$

Fig. 16. Semantics of $\lambda^{\mathtt{S}}_{\mathrm{trace}}$ (selected rules)

conjecture that *hextract* implements a least solution of the given constraint, since it adds no extraneous constraints in the construction, but in any case soundness is sufficient to obtain a type safety result for the implementation, as follows.

Lemma 7.22 (*Implementation Soundness*)
Suppose $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable and $\mathit{close}(C)$ is consistent. Then $\varnothing, H, C \vdash e : \tau$ is satisfiable, and there exists a solution ρ of C such that $(\mathit{hextract}\ \mathit{close}(C)\ H) = \rho(H)$.

Proof
If $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable then so is $\varnothing, H, C \vdash e : \tau$ by Lemma 7.3. If $\mathit{close}(C)$ is consistent then C has a solution ρ' by Lemma 7.21, and by Lemma 7.20 C has an *Eff* solution ρ'' such that $(\mathit{hextract}\ \mathit{close}(C)) \preccurlyeq \rho''$. Let $\rho = \rho' \circ \rho'' \mid_{\mathcal{V}_{\mathit{Eff}}}$. Since $(\mathit{hextract}\ \mathit{close}(C)\ H)$ is defined and closed by Lemma 7.16, therefore:

$$\rho''(H) = (\mathit{hextract}\ \mathit{close}(C)\ H) = \rho(H)$$

and ρ is a solution of C by Lemma 7.12. $\square$

8 The Stack-Based Variation

In this section we define a stack-based variation on the framework of the previous sections, allowing properties of the runtime stack at a program point to be verified at compile-time. Instead of keeping track of *all* events, only events for functions on the current call stack are maintained, in keeping with a general stack-based security model (as in e.g. (Jensen *et al.*, 1999)). Assertions ϕ in this framework are run-time assertions about the *active* event sequence, not all events. While the stack-based model is somewhat distinct from the trace-based model, we show that this variation requires only a minor "post-processing" of inferred trace effects for a sound analysis. There are results showing how it is possible to directly model-check stack properties of a Push-Down Automata (PDA) computation (Esparza *et al.*, 2001); our approach based on post-processing trace effects represents an alternative method, which may also prove useful for modelling features such as exceptions: the raising of an exception implies any subsequent effect is discarded.

We note that our system captures a more fine-grained stack-based model than has been previously proposed; in particular, the use of stacks of traces allows the ordering of events within individual stack frames to be taken into account, along with the ordering of frames themselves.

8.1 Syntax and Semantics

The values, expressions, and evaluation contexts of $\lambda^{\mathtt{S}}_{\mathrm{trace}}$ are exactly those of λ_{trace}, extended with an expression form $\cdot e \cdot$ and evaluation context form $\cdot E \cdot$ for delimiting the scope of function activations. We impose the requirement that in any function $\lambda_z x.e$, there exist no subexpressions $\cdot e' \cdot$ of e. The operational semantics of $\lambda^{\mathtt{S}}_{\mathrm{trace}}$, is a relation on *configurations* $\mathtt{S}, e$, where $\mathtt{S}$ ranges over stacks of traces, defined in Fig. 16. The active security context for run-time checks is obtained from the trace stack in configurations, by appending traces in the order they appear in the stack. To formalise this, we define the notation $\lfloor \mathtt{S} \rfloor$ inductively as follows:

$$\lfloor \mathrm{nil} \rfloor = \epsilon \qquad\qquad \lfloor \mathtt{S} :: \eta \rfloor = \lfloor \mathtt{S} \rfloor ; \eta$$

Abusing notation, we write $\hat{\mathtt{S}}$ as syntactic sugar for $\hat{\lfloor \mathtt{S} \rfloor}$.

Selected rules for the reduction relations $\rightsquigarrow$ and $\rightarrow$ on configurations are then specified in Fig. 16 (those not specified are obtained by replacing metavariables η with $\mathtt{S}$ in reductions for other expression forms in Fig. 2). The trace interpretation function Π is defined as for λ_{trace}.

8.2 Stackified Trace Effects

Although $\lambda^{\mathtt{S}}_{\mathrm{trace}}$ uses stack rather than trace contexts at run-time, we are able to use the type and verification framework developed previously, assigning types to $\lambda^{\mathtt{S}}_{\mathrm{trace}}$ expressions in the same manner as λ_{trace} expressions. The only additional requirement will be to process trace effects– to *stackify* them, yielding an approximation of the stack contexts that will evolve at run-time. The trick is to use μ-delimited scope in trace types, since this corresponds to function scope in inferred types as discussed in Sect. 7, and function activations and deactivations induce pushes and pops at run-time. The *stackify* algorithm is defined inductively in Figure 17. This algorithm works over effects that are sequences; for effects that are not sequences, the last clause massages into sequence form.

The last three clauses use trace effect equalities characterised in Sect. 3 to "massage" trace effects into appropriate 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 (Esparza *et al.*, 2001).

Example 8.1
With a, b, c representing arbitrary events, and results of stackification simplified via effect equivalences to increase readability:

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

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

$$\begin{array}{rcl}
\mathit{stackify}(\epsilon) & = & \epsilon \\
\mathit{stackify}(\epsilon; H) & = & \mathit{stackify}(H) \\
\mathit{stackify}(\mathit{ev}(c); H) & = & \mathit{ev}(c); \mathit{stackify}(H) \\
\mathit{stackify}(h; H) & = & h|\mathit{stackify}(H) \\
\mathit{stackify}((\mu h.H_1); H_2) & = & (\mu h.\mathit{stackify}(H_1)) \mid \mathit{stackify}(H_2) \\
\mathit{stackify}((H_1|H_2); H) & = & \mathit{stackify}(H_1; H) \mid \mathit{stackify}(H_2; H) \\
\mathit{stackify}((H_1; H_2); H_3) & = & \mathit{stackify}(H_1; (H_2; H_3)) \\
\mathit{stackify}(H) & = & \mathit{stackify}(H; \epsilon)
\end{array}$$

Fig. 17. The *stackify* algorithm

In the third example, since $\mu h.a$ precedes b, but $\mu h.a$ 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 μ-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 ϵs in the last two examples are artifacts of the transformation, that could be cleaned up with some minor alterations to *stackify*.

8.3 Properties

In this section we prove a trace approximation result for $\lambda^{\mathsf{S}}_{\text{trace}}$ (Theorem 8.1), showing how stackified effects generated by λ_{trace} type inference approximate program trace behaviour in the $\lambda^{\mathsf{S}}_{\text{trace}}$ model. The result is obtained by simulation of $\lambda^{\mathsf{S}}_{\text{trace}}$ in λ_{trace} (Lemma 8.8), and reuse of trace approximation (Corollary 6.3) for λ_{trace}. Because we are only concerned with showing that stackification is correct, by proving that it yields an approximation of program event traces, we omit consideration of checks, since simulation of checks would be a needless complication.

As the basis of the simulation, we define an embedding $(\!|\, e \,|\!)$ of $\lambda^{\mathsf{S}}_{\text{trace}}$ expressions in λ_{trace} expressions; expressions are essentially unchanged, except that distinguished events push and pop are inserted in programs to mark function invocation and return. Later, we will define a trace transformation that deduces the stack context from the push and pop markers.

Definition 8.1

Let c_{dummy} be a distinguished singleton constant, and let:

$$\text{push} \triangleq \mathit{ev}_{push}(c_{dummy}) \qquad\qquad \text{pop} \triangleq \mathit{ev}_{pop}(c_{dummy})$$

Then the $\lambda^{\mathsf{S}}_{\text{trace}}$ to λ_{trace} encoding of expressions, denoted $(\!|\, e \,|\!)$, is defined inductively as follows:

$$\begin{array}{rcl}
(\!|\, \lambda_z x.e \,|\!) & = & \lambda_z x.\mathrm{push}; (\lambda x.\mathrm{pop}; x)(\!|\, e \,|\!) \\
(\!|\, \cdot e \cdot \,|\!) & = & (\lambda x.\mathrm{pop}; x)(\!|\, e \,|\!) \\
(\!|\, \mathbf{c} \,|\!) & = & \mathbf{c} \\
(\!|\, ev(e) \,|\!) & = & ev((\!|\, e \,|\!)) \\
(\!|\, x \,|\!) & = & x \\
(\!|\, e_1 e_2 \,|\!) & = & (\!|\, e_1 \,|\!)(\!|\, e_2 \,|\!) \\
(\!|\, \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 \,|\!) & = & \mathsf{if}\, (\!|\, e_1 \,|\!) \,\mathsf{then}\, (\!|\, e_2 \,|\!) \,\mathsf{else}\, (\!|\, e_3 \,|\!) \\
(\!|\, \mathsf{let}\, x = v \,\mathsf{in}\, e \,|\!) & = & \mathsf{let}\, (\!|\, x \,|\!) = (\!|\, v \,|\!) \,\mathsf{in}\, (\!|\, e \,|\!)
\end{array}$$

Given this encoding, it is clear that traces of images in the encoding will be of a particular form ς– every pop event has a matching push, with possible nestings, corresponding to the activation control flow structure. We specify this form grammatically:

$$\begin{array}{lcll}
s & ::= & \mathrm{push}; \mathrm{pop} \mid \mathrm{push}; s; \mathrm{pop} \mid \epsilon \mid ev(c) \mid s; s & \quad ev(c) \notin \{\mathrm{push}, \mathrm{pop}\} \\
\varsigma & ::= & s \mid \mathrm{push} \mid \varsigma; \varsigma &
\end{array}$$

The point of the encoding is that a stack context can faithfully be obtained from traces ς. The proof of simulation will also require that simulation of stack contexts in traces ς include a reflection of stack structure– every stack frame connector :: should be simulated by an unmatched push. To this end we define the following.

Definition 8.2
The stack context simulation function sctx is defined as:

$$\begin{array}{rcl}
\mathrm{sctx}(\epsilon) & = & \epsilon \\
\mathrm{sctx}(\mathrm{push}) & = & \epsilon \\
\mathrm{sctx}(ev(c)) & = & ev(c) \\
\mathrm{sctx}(\mathrm{push}; \mathrm{pop}) & = & \epsilon \\
\mathrm{sctx}(\mathrm{push}; \varsigma; \mathrm{pop}) & = & \epsilon \\
\mathrm{sctx}(\varsigma_1; \varsigma_2) & = & \mathrm{sctx}(\varsigma_1)\, \mathrm{sctx}(\varsigma_2)
\end{array}$$

A trace ς simulates a stack $\mathtt{S}$, written $\mathtt{S} \sim \varsigma$, iff the relation can be derived given the following rules.

$$\frac{\mathrm{sctx}(\varsigma) = \epsilon}{\mathrm{nil} \sim \varsigma} \qquad\qquad \frac{\mathtt{S} \sim \varsigma \qquad \mathrm{sctx}(s) = \hat{\eta}}{\mathtt{S} :: \eta \sim \varsigma; \mathrm{push}; s}$$

Since stack traces ς are special forms of trace effects, interpretations and equivalence of them is defined as for trace effects in general. Thus, we can characterise sctx with the following Lemmas.

Lemma 8.1
If $\mathtt{S} \sim \varsigma$ then $\hat{\mathtt{S}} = \mathrm{sctx}(\varsigma)$.

Lemma 8.2
If $\varsigma = \varsigma'$ then $\mathrm{sctx}(\varsigma) = \mathrm{sctx}(\varsigma')$.

The following auxiliary lemma will allow us to "pop" the simulated stack by postpending a pop event, which is essential to simulating $\lambda^{\mathtt{S}}_{\mathrm{trace}}$ using λ_{trace} machinery.

Lemma 8.3
If $\mathtt{S} :: \eta \sim \varsigma$, then $\mathtt{S} \sim \varsigma; \mathrm{pop}$.

Proof
By definition of $\sim$, we have that $\varsigma = (\varsigma'; \mathrm{push}; s)$ where $\mathrm{sctx}(s) = \hat{\eta}$ and $\mathtt{S} \sim \varsigma'$. The proof then proceeds by case analysis on $\mathtt{S}$.

Case $\mathtt{S} = \mathrm{nil}$. Since $\mathtt{S} \sim \varsigma'$, therefore $\mathrm{sctx}(\varsigma') = \epsilon$ by definition of $\sim$. But also $\mathrm{sctx}(\mathrm{push}; s; \mathrm{pop}) = \epsilon$ by definition of sctx, and also by definition of sctx we have $\mathrm{sctx}(\varsigma; \mathrm{pop}) = \mathrm{sctx}(\varsigma')\,\mathrm{sctx}(\mathrm{push}; s; \mathrm{pop})$, i.e. $\mathrm{sctx}(\varsigma; \mathrm{pop}) = \epsilon$, so $\mathtt{S} \sim \varsigma; \mathrm{pop}$ in this case by definition of $\sim$.

Case $\mathtt{S} = \mathtt{S}' :: \eta$. Since $\mathtt{S} \sim \varsigma'$, therefore $\varsigma' = \varsigma''; \mathrm{push}; s'$ where $\mathtt{S}' \sim \varsigma''$ and $\mathrm{sctx}(s') = \hat{\eta}$ by definition of $\sim$. But since $\mathrm{sctx}(\mathrm{push}; s; \mathrm{pop}) = \epsilon$ by definition of sctx, and also $\mathrm{sctx}(s'; \mathrm{push}; s; \mathrm{pop}) = \mathrm{sctx}(s')\,\mathrm{sctx}(\mathrm{push}; s; \mathrm{pop})$, therefore $\mathrm{sctx}(s'; \mathrm{push}; s; \mathrm{pop}) = \mathrm{sctx}(s') = \hat{\eta}$, so $\mathtt{S} \sim \varsigma; \mathrm{pop}$ in this case by definition of $\sim$. $\square$

After the following auxiliary lemma, we show that $\rightsquigarrow$ reduction is simulated in the encoding $(\!|\cdot|\!)$. Note that the proof allows one step of $\lambda^{\mathtt{S}}_{\mathrm{trace}}$ reduction to be simulated by multiple steps in λ_{trace} reduction. The essential feature of the result is the simulation of stacks by traces ς.

Lemma 8.4
$(\!| e |\!)[(\!| v |\!)/x] = (\!| e[v/x] |\!)$.

Lemma 8.5
Given $\varsigma_\epsilon = \varsigma$ such that $\mathtt{S} \sim \varsigma_\epsilon$. If $\mathtt{S}, e \rightsquigarrow \mathtt{S}', e'$ then $\varsigma, (\!| e |\!) \rightarrow^\star \varsigma', (\!| e' |\!)$ where there exists $\varsigma'_\epsilon = \varsigma'$ such that $\mathtt{S}' \sim \varsigma'_\epsilon$.

Proof
By case analysis on the rule form of the reduction $\mathtt{S}, e \rightsquigarrow \mathtt{S}', e'$.

Case β. In this case $e = (\lambda_z x.e_0)v$, $e' = \cdot e_0[v/x][e/z]\cdot$, and $\mathtt{S}' = \mathtt{S} :: \epsilon$. By Definition 8.1 we have:

$$(\!| e |\!) = (\!| (\lambda_z x.e_0) |\!)(\!| v |\!) \qquad\qquad (\!| \lambda_z x.e_0 |\!) = \lambda_z x.\mathrm{push}; (\lambda x.\mathrm{pop}; x)(\!| e_0 |\!)$$

and since $(\!| v |\!)$ is a value by Definition 8.1, therefore by β:

$$\begin{gathered}\varsigma, (\lambda_z x.\mathrm{push}; (\lambda x.\mathrm{pop}; x)(\!| e_0 |\!))(\!| v |\!) \\ \rightsquigarrow \\ \varsigma, (\mathrm{push}; (\lambda x.\mathrm{pop}; x)(\!| e_0 |\!))[(\!| v |\!)/x][(\!| \lambda_z x.e_0 |\!)/z]\end{gathered}$$

But by definition of substitution and Lemma 8.4:

$$\begin{gathered}(\mathrm{push}; (\lambda x.\mathrm{pop}; x)(\!| e_0 |\!))[(\!| v |\!)/x][(\!| \lambda_z x.e_0 |\!)/z] \\ = \\ \mathrm{push}; (\lambda x.\mathrm{pop}; x)(\!| e_0[v/x][\lambda_z x.e_0/z] |\!)\end{gathered}$$

and by *context*, *event*, and *seq*:

$$\begin{gathered}\varsigma, \mathrm{push}; (\lambda x.\mathrm{pop}; x)(\!| e_0[v/x][\lambda_z x.e_0/z] |\!) \\ \rightarrow^\star \\ \varsigma; \mathrm{push}, (\lambda x.\mathrm{pop}; x)(\!| e_0[v/x][\lambda_z x.e_0/z] |\!)\end{gathered}$$

But since $\mathtt{S} \sim \varsigma_\epsilon$ by assumption, therefore $(\mathtt{S} :: \epsilon) \sim (\varsigma_\epsilon; \mathrm{push}; \epsilon)$ by definition of $\sim$, and by Definition 8.1:

$$(\!|\, \cdot e_0[v/x][\lambda_z x.e_0/z]\cdot \,|\!) = (\lambda x.\mathrm{pop}; x)(\!|\, e_0[v/x][\lambda_z x.e_0/z] \,|\!)$$

so this case holds, since $\varsigma_\epsilon; \mathrm{push}; \epsilon = \varsigma; \mathrm{push}$.

Case *pop*. In this case $e = \cdot v\cdot$, $\mathtt{S} = \mathtt{S}' :: \eta$, and $e' = v$. By Definition 8.1:

$$(\!|\, \cdot v\cdot \,|\!) = (\lambda x.\mathrm{pop}; x)(\!|\, v \,|\!)$$

and since $(\!|\, v \,|\!)$ is a value by Definition 8.1, by β:

$$\varsigma, (\lambda x.\mathrm{pop}; x)(\!|\, v \,|\!) \leadsto \varsigma, \mathrm{pop}; (\!|\, v \,|\!)$$

and by *event*, *seq*, and *context*:

$$\varsigma, \mathrm{pop}; (\!|\, v \,|\!) \rightarrow^\star \varsigma; \mathrm{pop}, (\!|\, v \,|\!)$$

Further, by assumption $\mathtt{S}' :: \eta \sim \varsigma_\epsilon$, therefore $\mathtt{S}' \sim \varsigma_\epsilon; \mathrm{pop}$ by Lemma 8.3, so this case holds since $\varsigma_\epsilon; \mathrm{pop} = \varsigma; \mathrm{pop}$.

The rest of the proof follows in a straightforward manner by homomorphism of $(\!|\, e \,|\!)$ in the remaining cases. $\square$

To extend the simulation result to $\rightarrow$ reduction, the encoding $(\!|\cdot|\!)$ needs to be extended to evaluation contexts.

Definition 8.3
The encoding $(\!|\cdot|\!)$ is extended to evaluation contexts as follows, where a homomorphic extension to other context forms is elided:

$$\begin{array}{rcl}
(\!|\, [\,] \,|\!) & = & [\,] \\
(\!|\, \cdot E\cdot \,|\!) & = & (\lambda x.\mathrm{pop}; x)(\!|\, E \,|\!) \\
(\!|\, Ee \,|\!) & = & (\!|\, E \,|\!)(\!|\, e \,|\!) \\
(\!|\, vE \,|\!) & = & (\!|\, v \,|\!)(\!|\, E \,|\!) \\
(\!|\, \mathsf{if}\, E\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \,|\!) & = & \mathsf{if}\, (\!|\, E \,|\!)\, \mathsf{then}\, (\!|\, e_1 \,|\!)\, \mathsf{else}\, (\!|\, e_2 \,|\!) \\
 & \vdots &
\end{array}$$

After the following auxiliary lemma, we prove a simulation result for $\rightarrow$ reduction, which follows easily on the basis of Lemma 8.5. This is immediately followed by the main simulation result, for $\rightarrow^\star$ reduction.

Lemma 8.6
$(\!|\, E[e] \,|\!) = (\!|\, E \,|\!)[(\!|\, e \,|\!)]$.

Lemma 8.7
Given $\varsigma_\epsilon = \varsigma$ such that $\mathtt{S} \sim \varsigma_\epsilon$. If $\mathtt{S}, E[e] \rightarrow \mathtt{S}', E[e']$ then $\varsigma, (\!|\, E[e'] \,|\!) \rightarrow^\star \varsigma', (\!|\, E[e'] \,|\!)$ where there exists $\varsigma'_\epsilon = \varsigma'$ such that $\mathtt{S}' \sim \varsigma'_\epsilon$.

Proof

By *context* it is the case that $\mathtt{S}, e \rightsquigarrow \mathtt{S}', e'$. Therefore $\varsigma, (\!| e |\!) \to^\star \varsigma', (\!| e' |\!)$ where there exists $\varsigma'_\epsilon = \varsigma'$ such that $\mathtt{S}' \sim \varsigma'_\epsilon$, by Lemma 8.5. This means $\varsigma, (\!| E |\!)[(\!| e |\!)] \to^\star \varsigma', (\!| E |\!)[(\!| e' |\!)]$ by *context*, and since $(\!| E |\!)[(\!| e |\!)] = (\!| E[e] |\!)$ and $(\!| E |\!)[(\!| e' |\!)] = (\!| E[e'] |\!)$ by Lemma 8.6, the result follows. □

Lemma 8.8 ($\lambda^{\mathtt{S}}_{trace}$ *Simulation*)
If $\mathrm{nil}, e \to^\star \mathtt{S}, e'$, then $\epsilon, (\!| e |\!) \to^\star \varsigma, (\!| e' |\!)$ with $\hat{\mathtt{S}} = \mathrm{sctx}(\varsigma)$.

Proof
Since nil $\sim \epsilon$ by definition of $\sim$, therefore by Lemma 8.7 and induction on the length of the reduction $\mathrm{nil}, e \to^\star \mathtt{S}, e'$, we can assert $\epsilon, (\!| e |\!) \to^\star \varsigma, (\!| e' |\!)$ where there exists $\varsigma_\epsilon = \varsigma$ such that $\mathtt{S} \sim \varsigma_\epsilon$. By Lemma 8.1 it is the case that $\mathrm{sctx}(\varsigma_\epsilon) = \hat{\mathtt{S}}$, so by Lemma 8.2 the result follows. □

Having established the dynamic simulation, we now turn to stackification of trace effects. Our strategy is to show that the sctx transformation of traces ς is contained in the interpretation of stackified effect approximations of ς. The simulation result just proved will then allow us to make the bridge to $\lambda^{\mathtt{S}}_{\mathrm{trace}}$ stack contexts. First, it will ease matters to define an alternate transformation of traces ς, one that "lines up" better with stackification. We do so as follows; note the structural similarity with stackification:

Definition 8.4
Let ς trace sctxs transformation be defined inductively as follows:

$$\begin{array}{rcl}
\mathrm{sctxs}(\epsilon) & = & \epsilon \\
\mathrm{sctxs}(\epsilon;\varsigma) & = & \mathrm{sctxs}(\varsigma) \\
\mathrm{sctxs}(\mathrm{push};\varsigma) & = & \mathrm{sctxs}(\varsigma) \\
\mathrm{sctxs}(\mathit{ev}(c);\varsigma) & = & \mathit{ev}(c);\mathrm{sctxs}(\varsigma) \\
\mathrm{sctxs}((\mathrm{push};\varsigma;\mathrm{pop});\varsigma') & = & \mathrm{sctxs}(\varsigma)|\mathrm{sctxs}(\varsigma') \\
\mathrm{sctxs}((\mathrm{push};\mathrm{pop});\varsigma) & = & \mathrm{sctxs}(\varsigma) \\
\mathrm{sctxs}((\varsigma_1;\varsigma_2);\varsigma_3) & = & \mathrm{sctxs}(\varsigma_1;(\varsigma_2;\varsigma_3)) \\
\mathrm{sctxs}(\varsigma) & = & \mathrm{sctxs}(\varsigma;\epsilon)
\end{array}$$

We immediately observe that sctx transformations are contained in trace effect interpretations of sctxs transformations:

Lemma 8.9
$\mathrm{sctx}(\varsigma) \in [\![\mathrm{sctxs}(\varsigma)]\!]$.

Now, the crux of our technique will be to use push and pop events to line up traces ς with effects H, and inductively show that the sctxs transformation is approximated by stackification. However, we want stackification to be more "aware" of push and pop events, and in particular to remove them in stackification, in keeping with the sctxs transformation, so we define an alternate function *stackify*$'$ with these modifications. The connection with *stackify* is easily made later, since the modifications are minor; this is because μ-scope in effects always corresponds to function scope, which is always delimited by push and pop events in images of $(\!| \cdot |\!)$ (as we prove below).

Definition 8.5
Let *stackify*$'$ be defined equivalently to *stackify*, except for the μ case, defined as follows:

$$\mathit{stackify}'((\mu h.\mathrm{push}; H_1; \mathrm{pop}); H_2) = (\mu h.\mathit{stackify}'(H_1)) \mid \mathit{stackify}'(H_2)$$

The following definition formalises this special form of effects, inferred for images of $(\!|\cdot|\!)$, as the *stackifiable* property of effects.

Definition 8.6
H is *stackifiable* iff one of the following conditions holds inductively:

1. $H \equiv h$
2. $H \equiv ev(c)$ and $ev(c) \neq \mathrm{push}, \mathrm{pop}$
3. $H \equiv H_1 | H_2$ and both H_1 and H_2 are stackifiable
4. $H \equiv H_1; H_2$ and both H_1 and H_2 are stackifiable
5. $H \equiv \mu h.\mathrm{push}; H; \mathrm{pop}$ and H is stackifiable

Now, following an important auxiliary lemma necessary to deal with μ-bound effects, we show that if H approximates a trace ς, then *stackify*$'(H)$ approximates $\mathrm{sctxs}(\varsigma)$.

Lemma 8.10
Given stackifiable H. Then:

$$\mathit{stackify}'(H[\mu h.\mathrm{push}; H; \mathrm{pop}/h]) = \mathit{stackify}'(H)[\mu h.\mathit{stackify}'(H)/h]$$

Proof
Straightforward by induction on h. $\square$

Lemma 8.11
For all H, define:

$$[\![H]\!]_{pre} = \{\theta \mid \theta \in [\![H]\!] \text{ and } \theta \neq (\theta' \downarrow)\}$$

Then, stackifiable H and $\hat{\varsigma} \in [\![H]\!]$ implies $[\![\mathrm{sctxs}(\varsigma)]\!]_{pre} \subseteq [\![\mathit{stackify}'(H)]\!]$.

Proof
(Sketch) By case analysis on the match forms of ς in the definition clauses of sctxs and induction on the recursive call tree of $\mathrm{sctxs}(\varsigma)$. By the definitions of sctxs and *stackify*$'$, and correspondence of the match forms in the defining clauses, it is easy (and tedious) to "line up" the different cases of ς and H, and demonstrate the result by induction. The most interesting case is $\varsigma = \mathrm{push}; \varsigma'; \mathrm{pop}; \varsigma''$. It follows in this case that H must be of the form $H = (\mu h.\mathrm{push}; H'; \mathrm{pop}); H''$, where $\widehat{\mathrm{push}; \varsigma'; \mathrm{pop}} \in [\![\mu h.\mathrm{push}; H'; \mathrm{pop}]\!]$ and $\hat{\varsigma''} \in [\![H'']\!]$, since the push and pop events occur nowhere but at the limits of μ-scope in stackifiable effects, hence $\hat{\varsigma'} \in [\![H'[\mu h.\mathrm{push}; H'; \mathrm{pop}/h]]\!]$. Therefore by the induction hypothesis:

$$\begin{array}{rcl}
[\![\mathrm{sctxs}(\varsigma')]\!]_{pre} & \subseteq & \mathit{stackify}'(H'[\mu h.\mathrm{push}; H'; \mathrm{pop}/h]) \\
[\![\mathrm{sctxs}(\varsigma'')]\!]_{pre} & \subseteq & \mathit{stackify}'(H'')
\end{array}$$

Since:

$$\begin{array}{rcl}\mathrm{sctxs}(\mathrm{push};\varsigma';\mathrm{pop};\varsigma'') & = & \mathrm{sctxs}(\varsigma')|\mathrm{sctxs}(\varsigma'') \\ \mathit{stackify}'((\mu h.\mathrm{push};H';\mathrm{pop});H'') & = & \mu h.\mathit{stackify}'(H')|\mathit{stackify}'(H'')\end{array}$$

It remains to be shown that $[\![\mathrm{sctxs}(\varsigma')]\!]_{pre} \subseteq [\![\mu h.\mathit{stackify}'(H')]\!]$. Observing the equality $\mu h.\mathit{stackify}'(H') = \mathit{stackify}(H')[\mu h.\mathit{stackify}'(H')/h]$ implied by Lemma 3.1, we can assert $\mu h.\mathit{stackify}'(H') = \mathit{stackify}'(H'[\mu h.\mathrm{push};H';\mathrm{pop}/h])$ by Lemma 8.10. The result follows. $\square$

Now, we show that type inference generates stackifiable effects. The typing rules actually don't "cleanly" place push and pop events at the limits of μ-scope, but rather there will be intervening cruft such as ϵs and dummy μ-bindings, due to the nature of the encoding of functions. Therefore, a simple transformation f is necessary to obtain an equivalent stackifiable form of effects.

Lemma 8.12
Given satisfiable $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ for closed e. Then $\varnothing, H' \vdash_{\bar{\beta}'} (\!| e |\!) : \tau/C'$ is satisfiable, where there exists a function f such that each of the following conditions hold:

1. $f(\mathit{hextract}\ C\ H')$ is stackifiable
2. $f(\mathit{hextract}\ C\ H') = (\mathit{hextract}\ C\ H')$
3. $\mathit{stackify}(\mathit{hextract}\ C\ H) = \mathit{stackify}'(f(\mathit{hextract}\ C'\ H'))$

Proof
(Sketch) Observe that any effect is stackifiable if the only occurrence of push and pop is at the limits of μ-bound subeffects, and that all μ-bound subeffects have such delimitation. Also, the only place where μ-bound effects occur in $(\mathit{hextract}\ C\ H)$ will be in *hextract*ion of effect variable components of H in C (in the sense of Definition 7.10). Now, if e is closed, and H has a variable component h, then it is demonstrable by definition of type inference and Lemma 7.15 that h must have the effect H_λ of one or more concrete functions $\lambda_z x.e$ as lower bounds. Furthermore, since $(\!| \lambda_z x.e |\!) = \lambda_z x.\mathrm{push};(\lambda x.\mathrm{pop};x)(\!| e |\!)$, the lower bounds induced by the transformation $(\!| \lambda_z x.e |\!)$ will be equivalent to $\mathrm{push};H_\lambda;\mathrm{pop}$ (this is approximate; the described effect transformation is also carried recursively through H_λ, as the expression transformation is carried recursively through the function body). In short, if $(\mathit{hextract}\ C\ H)$ contains a subeffect $\mu h.H_\lambda$, then $f(\mathit{hextract}(H',C'))$ will have the same shape as $(\mathit{hextract}\ C\ H)$, and will contain $\mu h.\mathrm{push};H_\lambda;\mathrm{pop}$ in the position corresponding to $\mu h.H_\lambda$ in $(\mathit{hextract}\ C\ H)$. Otherwise, observe that $f(\mathit{hextract}(H',C'))$ is homomorphic to $(\mathit{hextract}\ C\ H)$, establishing (2). Since $(\mathit{hextract}\ C\ H)$ contains no occurrences of push or pop, therefore $f(\mathit{hextract}(H',C'))$ contains none aside from those at the limits of μ-scope, and is therefore stackifiable, establishing (1), while (3) follows by definition of *stackify* and *stackify*$'$. $\square$

Finally, we can demonstrate correctness of *stackify*, by demonstrating a trace approximation result for $\lambda^{\mathsf{S}}_{\mathrm{trace}}$ via on the strength of the preceding results and trace approximation in λ_{trace}.

Theorem 8.1 (*Trace Approximation stackify*)
If $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is satisfiable and also nil, $e \to^{\star} \mathtt{S}, e'$, then $\hat{\mathtt{S}} \in [\![\mathit{stackify}(\mathit{hextract}\ C\ H)]\!]$.

Proof
First note that e must be top-level, otherwise it would not be typable as in the premise. Thus by Lemma 8.8, we have $\epsilon, (\!| e |\!) \to^{\star} \varsigma, (\!| e' |\!)$ with $\mathrm{sctx}(\varsigma) = \hat{\mathtt{S}}$. By Lemma 8.12 a there exists a satisfiable judgement $\varnothing, H' \vdash_{\bar{\beta}'} (\!| e |\!) : \tau/C'$, with $\hat{\varsigma} \in [\![\mathit{hextract}(H', C')]\!]$ by Corollary 6.3. Therefore, by Lemma 8.9 and Lemma 8.11 and the equality $\mathrm{sctx}(\varsigma) = \hat{\mathtt{S}}$, we may assert $\hat{\mathtt{S}} \in [\![\mathit{stackify}'(\mathit{hextract}(H', C'))]\!]$, so the result follows by Lemma 8.12. □

9 Verification of Trace Effects

We have described the syntax and semantics of λ_{trace}, that includes a trace component in configurations. We have also described two type and effect systems that conservatively approximate run-time traces, and shown that type and effects can be automatically inferred. However, we have been abstract so far with respect to the definition of checks and effect verification, basing safety of computation and validity of typing on a yet-to-be-defined notion of check and effect validity. To fill in these details, we define in this Section:

- A logic for run-time checks, including a syntax for expressing checks as predicates in the logic, along with a notion of validity for these checks that can be automatically verified in the run-time system.
- A means of verifying validity of trace effects, as defined in Definition 3.4, where check events that are predicted by the trace effect analysis are automatically shown to either succeed or fail in the relevant context.

The latter point is necessary to address because, even though validity of trace effects has been defined, the notion is logical but not algorithmic; in particular, $[\![H]\!]$ may be an infinite set. We accomplish automated verification using a temporal logic and model-checking techniques, allowing us to reuse existing algorithms and results for trace effect verification. We use a linear-time logic because the run-time validity of a predicate is based on one linear trace η only.

9.1 Trace Effects are BPAs

We interpret trace effects as labelled transition systems (Definition 3.3), with a form that is appropriate for our application. This form is close to that of Basic Process Algebra (BPA) terms; in fact the latter can faithfully represent trace types. Such a representation allows us to apply known model-checking algorithms to our analysis, since several exist for verifying properties of BPAs (Steffen & Burkart, 1992).

Definition 9.1
BPA expressions are defined as follows:

$$p \quad ::= \quad a \mid \epsilon \mid p \cdot p \mid p + p \mid X \qquad \textit{BPA expressions}$$

Intuitively, each a denotes a transition action, ϵ is the empty process, $\cdot$ denotes sequencing, and $+$ denotes nondeterministic choice. Each X in a BPA process is defined via a set of declarations Δ of the form $X \triangleq p$. We let P range over sets of BPA expressions. The definition of BPA processes is completed via the following operational semantics, where $\epsilon \cdot p$ is considered equivalent to p:

$$\frac{p \xrightarrow{a} p'}{p + q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p + q \xrightarrow{a} q'} \qquad \frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q} \qquad \frac{p \xrightarrow{a} p' \qquad X \triangleq p \in \Delta}{X \xrightarrow{a} p'} \qquad a \xrightarrow{a} \epsilon$$

Now, we define a straightforward transformation from trace effects to BPA processes. We define actions a in the BPA encoding to be events $ev(c)$. Note that since BPA declarations may include only guarded expressions, the translation is meaningful just in case the transformation of a μ-bound trace effect yields a guarded declaration; we therefore assume a trivial pre-processing of traces to ensure this property, that e.g. could insert ϵ before every trace effect variable h:

Definition 9.2
The translation from closed, ϵ-free trace effects H to BPA expression, declaration pairs p, Δ is inductively defined as follows. The translation is parameterised by an environment Ψ that maps trace effect variables h to their BPA variable representations X in the encoding:

$$\begin{array}{rcl}
\mathrm{BPA}(\epsilon, \Psi) & = & \epsilon, \varnothing \\
\mathrm{BPA}(ev(c), \Psi) & = & ev(c), \varnothing \\
\mathrm{BPA}(H_1; H_2, \Psi) & = & \mathrm{let}\, p_1, \Delta_1 = \mathrm{BPA}(H_1, \Psi)\, \mathrm{in} \\
 & & \mathrm{let}\, p_2, \Delta_2 = \mathrm{BPA}(H_2, \Psi)\, \mathrm{in} \\
 & & p_1 \cdot p_2, \Delta_1 \cup \Delta_2 \\
\mathrm{BPA}(H_1 | H_2, \Psi) & = & \mathrm{let}\, p_1, \Delta_1 = \mathrm{BPA}(H_1, \Psi)\, \mathrm{in} \\
 & & \mathrm{let}\, p_2, \Delta_2 = \mathrm{BPA}(H_2, \Psi)\, \mathrm{in} \\
 & & p_1 + p_2, \Delta_1 \cup \Delta_2 \\
\mathrm{BPA}(\mu h.H, \Psi) & = & \mathrm{let}\, p, \Delta = \mathrm{BPA}(H, (\Psi; h : X))\, \mathrm{in} \\
 & & X, \Delta \cup \left\{ X \triangleq p \right\} \qquad X \text{ fresh} \\
\mathrm{BPA}(h, \Psi) & = & \Psi(h), \varnothing
\end{array}$$

For brevity, we write $\mathrm{BPA}(H)$ for $\mathrm{BPA}(H, \varnothing)$

The correctness of the translation falls out as a simple corollary:

Corollary 9.1
$[\![H]\!] = [\![\mathrm{BPA}(H)]\!]$.

9.2 Verified Checks in the Linear μ-calculus

While a variety of model-checking logics are available, we use the μ-calculus (Kozen, 1983) because it is powerful and is syntactically close to trace effects H. Further, efficient techniques for the automated verification of μ-calculus formulas on BPA

$$\begin{array}{rcl}
[\![\mathbf{true}]\!]_V & = & \Theta^\infty \\
[\![x]\!]_V & = & V(x) \\
[\![\neg\phi]\!]_V & = & \Theta^\infty - [\![\phi]\!]_V \\
[\![\phi_1 \wedge \phi_2]\!]_V & = & [\![\phi_1]\!]_V \cap [\![\phi_2]\!]_V \\
[\![(a)\phi]\!]_V & = & \{\theta^\infty \in \Theta^\infty \mid \theta^\infty = a;\theta_1^\infty \text{ and } \theta_1^\infty \in [\![\phi]\!]_V\} \\
[\![\boldsymbol{\nu} x.\phi]\!]_V & = & \bigcup\{W \subseteq \Theta^\infty \mid W \subseteq [\![\phi]\!]_{V[x\mapsto W]}\}
\end{array}$$

Fig. 18. Semantics of the linear-time μ-calculus

processes have been developed (Burkart *et al.*, 2001; Esparza, 1994). We use the *linear* variant of the μ-calculus (Esparza, 1994) because at run-time only one linear trace of events is known, and so trace effects H and run-time traces η can only be compared in a linear-time logic.

Definition 9.3
The syntax of the linear μ-calculus is:

$$\phi \quad ::= \quad x \mid \mathbf{true} \mid \mathbf{false} \mid (a)\phi \mid \boldsymbol{\mu} x.\phi \mid \boldsymbol{\nu} x.\phi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$$

Here, a ranges over arbitrary transition labels; in particular, we can let a range over events $ev(c)$. The semantics of the linear μ-calculus is defined in Fig. 18. This semantics is defined over potentially infinite traces $\theta^\infty \in \Theta^\infty$ that, unlike sets of traces θ, may not be prefix-closed. V denotes a mapping from μ-calculus variables to sets of infinite traces, $\varnothing$ denotes the empty mapping. $[\![\phi]\!]$ is shorthand for $[\![\phi]\!]_\varnothing$. Several formulae are not given in this figure because they can be defined in terms of the others:

$$\phi_1 \vee \phi_2 \iff \neg(\neg\phi_1 \wedge \neg\phi_2) \qquad \mathbf{false} \iff \neg\mathbf{true} \qquad \boldsymbol{\mu} x.\phi \iff \neg(\boldsymbol{\nu} x.\neg\phi)$$

Since our trace effect semantics is prefix-closed, we will explicitly prefix-close $[\![\phi]\!]$ so the two sets are compatible.

Definition 9.4
The prefix closure $\Theta^\infty\!\downarrow$ of a set of infinite traces Θ^∞ is:

$$\begin{array}{c}
\{\theta \mid \theta \text{ is a prefix of some } \theta^\infty \in \Theta^\infty\} \\
\cup \\
\{\theta\!\downarrow \mid \theta \text{ is finite and } \theta \in \Theta^\infty\}
\end{array}$$

Validity of a formulae with respect to a trace effect is defined via set containment of the trace interpretations of each:

Definition 9.5
The formula ϕ is valid for H, written $H \Vdash \phi$, iff $[\![H]\!] \subseteq [\![\phi]\!]\downarrow$.

This relation is decidable by known model-checking results and the above equivalence of BPA processes and trace effects:

Lemma 9.1
The relation is $H \Vdash \phi$ is decidable.

Proof
Model-checking linear μ-calculus formulae over BPA's is shown decidable in (Esparza, 1994). The key in this construction is that the valid traces of each linear μ-calculus formula ϕ can be characterised by a finite automaton and Büchi automaton that accept the finite and infinite traces of $\neg\phi$, respectively. Histories can be translated to equivalent BPA's by Corollary 9.1. $\square$

9.3 Relating Trace Effect and Trace Runtime Properties

We now instantiate the logic of trace checks in λ_{trace} with linear μ-calculus formulae ϕ. In Lemma 9.1, the relation $H \Vdash \phi$ was shown to be decidable, so this will make a natural foundation for trace effect verification.

One important requirement of this logic is that formulae must have truth values for a given trace effect H, *and* for a trace runtime η. The meaning of a formula ϕ under a run-time trace η is taken by verifying $\hat{\eta} \in [\![\phi]\!]\downarrow$. We will define the trace check interpretation function Π of Fig. 2 in terms of this relation (Definition 9.6).

In Corollary 4.1, trace effects were shown to approximate dynamic histories—in particular, if $\epsilon, e \rightarrow^{\star} \eta, e'$ and $H, \Gamma \vdash e : \tau$ is derivable, then $\hat{\eta} \in [\![H]\!]$. The key result linking the static and dynamic histories is the following:

Lemma 9.2
If $\hat{\eta} \in [\![H]\!]$ and $H \Vdash \phi$, then $\hat{\eta} \in [\![\phi]\!]\downarrow$.

Proof
Immediate from Definition 9.5 and Definition 3.3. $\square$

Corollary 4.1 ensures that by the above Lemma, verifying a trace effect entails verifying all trace checks that may occur at run-time, meaning a combination of type inference and automated verification yields a sound static analysis for λ_{trace}, as we state formally in Theorem 9.1.

9.4 Soundness of Verification

We have almost completed the definition of our logical framework for trace checks and trace effect verification. However, a few small points remain.

Firstly, in λ_{trace}, we are able to parameterise checks with constants with the syntax $\phi(c)$. To implement this, we specify a distinguished variable χ that may occur free in trace checks, which is assumed to be instantiated with a trace check parameter during verification.

Secondly, checks ϕ should express expected trace patterns that may occur up to the point of the check. This is the same as requiring that if an event $ev_{\phi}(c)$ occurs in a trace (resp. is predicted statically), the sub-trace η that immediately precedes it (resp. any trace that may precede it as predicted by typing) must exhibit the pattern specified by ϕ. However, there is an infinite regress lurking here: a property ϕ that mentions an event $ev_{\phi}(c)$ suggests circularity in the syntax of ϕ. Thus, we introduce a distinguished label Now, that in any formula ϕ represents the relevant checkpoint of ϕ. This label is interpreted appropriately during verification.

The mechanics of χ and Now are fully specified in Definition 9.6 below.

We now stand ready to instantiate our logic and verification framework. In the dynamic system, this is accomplished by defining the language of trace checks, and by defining the implementation of Π, our previously abstract representation of trace check verification:

Definition 9.6 (*Definition of* Π)
The framework of Section 2 is instantiated to let ϕ range over linear μ-calculus formulae, where labels a in ϕ are events $ev(c)$. Furthermore, we distinguish one event Now to represent checkpoints within the check specification (hence avoiding circularity), and letting $\chi \in \mathcal{V}_s$ be a distinguished variable for parameterising constants c in ϕ, we define Π as follows:

$$\Pi(\phi(c), \theta) \iff \theta \in [\![\phi[c/\chi][ev_\phi(c)/\text{Now}]]\!] \downarrow$$

Now, we specify what it means for a trace effect to be verified; intuitively, it means that if a trace effect H predicts the occurrence of a check event $ev_\phi(c)$, then H semantically entails ϕ instantiated with c. Formally:

Definition 9.7 (*Trace Effect Verification*)
A trace effect H is *verifiable* iff for all subterms $ev_\phi(c)$ of H it is the case that:

$$H \Vdash \phi[c/\chi][ev_\phi(c)/\text{Now}]$$

Here is a simple example that demonstrates the system in action:

Example 9.1
Assume we want to define a policy whereby if a particular check $\phi(c)$ is encountered at run-time, the only allowable previous events are $ev_1(c)$ or $ev_2(c)$, and $ev_2(c)$ must be the most recent event prior to the check. For brevity, let:

$$\begin{array}{rcl} (ev_{1|2}(\chi))\phi & \triangleq & (ev_1(\chi))\phi \vee (ev_2(\chi))\phi \\ (ev_{1|2}(\chi)*)\phi & \triangleq & \boldsymbol{\mu} x.(ev_{1|2}(\chi))x \vee \phi \end{array}$$

The above policy can then be expressed in our scheme with the following μ-calculus formula ϕ:

$$\phi \triangleq (ev_{1|2}(\chi)*)(ev_2(\chi))(\text{Now})\mathbf{true}$$

Now, assuming $\varnothing, \epsilon \vdash e' : \mathit{bool}$, let:

$$e \triangleq \mathsf{if}\, e' \,\mathsf{then}\, ev_1(c); ev_2(c) \,\mathsf{else}\, ev_2(c)$$

Thus, depending on the valuation of e', either:

$$\epsilon, e; \phi(c) \to^\star ev_1(c); ev_2(c), \phi(c)$$

or:

$$\epsilon, e; \phi(c) \to^\star ev_2(c), \phi(c)$$

and we see that the check in the program $(e; \phi(c))$ will always succeed at runtime, since these relations hold:

$$\begin{array}{rcl} ev_2(c)ev_\phi(c) & \in & [\![\phi[c/\chi][ev_\phi(c)/\text{Now}]]\!] \downarrow \\ ev_1(c)ev_2(c)ev_\phi(c) & \in & [\![\phi[c/\chi][ev_\phi(c)/\text{Now}]]\!] \downarrow \end{array}$$

We also have that the following judgement is derivable:

$$\varnothing, (ev_2(c) | ev_1(c); ev_2(c)); ev_\phi(c) \vdash e; \phi(c) : \mathit{unit}$$

and this entailment holds:

$$(ev_2(c) | (ev_1(c); ev_2(c))); ev_\phi(c) \Vdash \phi[c/\chi][ev_\phi(c)/\mathrm{Now}]$$

Therefore, the judgement is valid, predicting the run-time safety of the program $(e; \phi(c))$.

The preceding construction completes the definition of the framework. Lemma 9.2 and definition of Π together yield the desired formal property for trace effect verification as a Corollary:

Corollary 9.2 (*Verification Soundness*)
If H is verified, then H is valid.

Proof
Immediate by Definition 3.4, Definition 9.6, Definition 9.7, and Lemma 9.2. □

This result can then be composed with Lemma 7.22 and Theorem 6.2 to yield a type safety property for the complete automated analysis.

Theorem 9.1
Suppose $\varnothing, H \vdash_{\bar{\beta}} e : \tau/C$ is derivable, *close*(C) is consistent, and *hextract* $C\ H$ is verifiable. Then e does not go wrong.

9.5 Incompleteness of Verification

Completeness of verification is a different matter than soundness, and the reader may notice that our verification technique is approximate. In particular, verification of a trace effect H (Definition 9.7) with respect to a predicate ϕ always ensures that $\theta \in [\![H]\!]$ implies $\theta \in [\![\phi]\!]$, regardless of whether ϕ occurs in θ, whereas validity (Definition 3.4) only requires that (roughly speaking) $\theta ev_\phi \in [\![H]\!]$ implies $\theta ev_\phi \in [\![\phi]\!]$, i.e. validity with respect to a given ϕ is concerned only with traces that end in ev_ϕ.

Example 9.2
With ϕ defined as in Example 9.1, the effect $(ev_1(c); \phi(c))$ is both valid and verifiable, as is the effect $ev_3(c)$ since it contains no checks. However, the join of the two is not verifiable, since $ev_3(c)$ does not satisfy $\phi(c)$, even though it is valid since $ev_3(c)$ precedes no checks and hence is irrelevant to validity:

$$ev_3(c) | (ev_1(c); \phi(c)) \text{ is valid but not verifiable}$$

While this is sound, more precise and complete verification can be obtained with some modifications. We now show how to enhance verification with a technique that will remove from consideration those traces containing no occurrence of a given formula.

We begin by defining a convenient μ-calculus formula abbreviation, using familiar notation. This abbreviation will be reused in Sect. 10:

Definition 9.8
Since verification of predicates ϕ is always with respect to a particular trace effect H, and any event $ev(c)$ occurring in $[\![H]\!]$ must occur in H for the class of trace effects we consider for verification (see Sect. 9.1), we can easily obtain the set of possible events in a trace set $[\![H]\!]$ as the set of events in H. Letting $ev_1(c_1), \cdots, ev_n(c_n)$ be the events for given H, we define the following abbreviations:

$$\begin{array}{rcl}(.)\phi & \triangleq & (ev_1(c_1))\phi \vee \cdots \vee (ev_n(c_n))\phi \\ (.*)\phi & \triangleq & \boldsymbol{\mu} x.(.)x \vee \phi\end{array}$$

A refined, more complete verification can then be defined as follows:

Definition 9.9 (Refined Verification)
Let the *occurs* predicate be defined as follows; a trace θ is in the interpretation of $\text{occurs}(ev(c))$ just in case $ev(c)$ occurs in θ:

$$\text{occurs}(ev(c)) \triangleq (.*)(ev(c))(.*)\mathbf{true}$$

A trace effect H is then *verified* iff for all subterms $ev_\phi(c)$ of H, letting:

$$\phi' = \phi[c/\chi][ev_\phi(c)/\text{Now}]$$

it is the case that:

$$H \Vdash \phi' \vee \neg\text{occurs}(\phi')$$

The soundness results in the previous section can then be replayed in the presence of this refined verification.

10 Applications to Language-Based Security

In this section we show that our program logic is sufficiently expressive to be useful in practice applications including security. In particular, we show how both stack inspection and history-based security paradigms can be statically enforced in our system. Singleton types allow a precise typing of resource parameters in the model.

In our examples we will be interested in unparameterised events and checks; in such cases we will write ev and ϕ for $ev(c_{\text{dummy}})$ and $ev_\phi(c_{\text{dummy}})$ respectively, where c_{dummy} is a distinguished dummy constant. Also, we will abbreviate events ev_i by their subscripts i, and the notation:

$$(\vee \{ev_1(c_1), \ldots, ev_n(c_n)\})\phi \triangleq ev_1(c_1)\phi \vee \cdots \vee ev_n(c_n)\phi$$

will be convenient.

10.1 Stack inspection with parameterised privileges

Java stack inspection (Wallach & Felten, 1998; Skalka & Smith, 2000; Pottier *et al.*, 2001) is a language-based security mechanism that uses an underlying security model of principals and resources– all code is annotated with a principal identifier p, and a local ACL policy $\mathcal{A}$ mapping principals p to resources $r(c)$ for which they are authorised is taken as given. An event ev_p is issued whenever a codebase annotated with p is entered. One additional feature of Java is that any principal may also explicitly *enable* a resource for which it is authorised. When a function is activated, its associated principal identifier is pushed on the stack, along with any resource enablings that occur in its body. Stack inspection for a particular resource $r(c)$ then checks the stack for an enabling of $r(c)$, searching frames from most to least recent, and failing if a principal unauthorised for $r(c)$ is encountered before an enabling of $r(c)$, or if no such enabling is encountered.

Stack inspection can be modelled in the stack-based variant of our programming logic defined in Section 8. Rather than defining the general encoding, we develop one particular example that illustrates all issues. Consider the following function checkit:

$$\text{checkit} \quad \triangleq \quad \lambda x.p\text{:system}; \phi_{\text{inspect},r:\text{filew}}(x)$$

Every function upon execution first issues an owner (principal) event, in this case p:system indicating "system" is the principal p that owns checkit. The function takes a parameter x (a file name) and inspects the stack for the "filew" resource with parameter x, via embedded μ-calculus assertion $\phi_{\text{inspect},r:\text{filew}}(x)$. This assertion is defined below; it enforces the fact that all functions on the call stack back to the nearest enable must be owned by principals p that according to ACL $\mathcal{A}$ are authorised for the r:filew(x) resource.

To model resource enabling we use a check event $\phi_{\text{enable},r}(x)$; the use of a check event allows to simultaneously mark a temporary enabling, and to check that enabling is performed by an authorised principal. We illustrate use of explicit enabling via an example "wrapper" function enableit, owned by the "accountant" principal p:acct, that takes a function f and a constant x, and enables r:filew(x) for the application of f to x:

$$\text{enableit} \quad \triangleq \quad \lambda f.p\text{:acct}; (\lambda x.p\text{:acct}; \phi_{\text{enable},r:\text{filew}}(x); \text{let}\, y = f(x)\, \text{in}\, y)$$

The definition of $\phi_{\text{inspect},r}(c)$, for fixed $r(c)$, is generalised over parameterised resources $r(c)$. For trace effect H containing only the events $ev_1(c_1), \ldots, ev_n(c_n)$, and parameterised resource $r(c)$, Fig. 19 gives the definition of $\phi_{\text{inspect},r}(c)$. The check ensures that it is neither the case that the check occurs without previous enablings of $r(c)$, nor that a principal unauthorised for the privilege is interposed between the check and the most recent enabling; this is logically equivalent to stack inspection.

Returning to our previous example expressions checkit and enableit, the following most general types are inferred in our system:

$$\begin{array}{rcl} \text{checkit} & : & \forall\alpha.\{\alpha\} \xrightarrow{p:\text{system};\phi_{\text{inspect},r:\text{filew}}(\alpha)} \mathit{unit} \\ \text{enableit} & : & \forall\alpha h t.(\{\alpha\} \xrightarrow{h} t) \xrightarrow{p:\text{acct}} \{\alpha\} \xrightarrow{p:\text{acct};\phi_{\text{enable},r:\text{filew}}(\alpha);h} t \end{array}$$

The stackification of the *hextract*ed effect of the application:

$$\text{enableit checkit (/accts/ledger.txt)}$$

$$
\begin{array}{rcl}
(p_{\neg r(c)})\phi & \triangleq & (\vee\{p | r(c) \notin \mathcal{A}(p)\})\phi \\
(\bar{p})\phi & \triangleq & (\vee(\{ev_1(c_1), \ldots, ev_n(c_n)\} \setminus \mathrm{dom}(\mathcal{A})))\phi \\
(\neg ev(c))\phi & \triangleq & (\vee(\{ev_1(c_1), \ldots, ev_n(c_n)\} \setminus \{ev(c)\}))\phi \\
(a*)\phi & \triangleq & \boldsymbol{\mu}x.(a)x \vee \phi \qquad \text{for } a \in \{\bar{p}, \neg ev(c)\} \\
\phi_{\mathrm{enable},r}(c) & \triangleq & \neg((.*)(p_{\neg r(c)})(\bar{p}*)(\mathrm{Now})\mathbf{true}) \\
\phi_{\mathrm{amplify},r}(c) & \triangleq & \phi_{\mathrm{enable},r}(c) \\
\phi_{\mathrm{inspect},r}(c) & \triangleq & \neg((\neg\phi_{\mathrm{enable},r}(c)*)(\mathrm{Now})\mathbf{true}) \wedge \\
 & & \neg((.*)(p_{\neg r(c)})(\neg\phi_{\mathrm{enable},r}(c)*)(\mathrm{Now})\mathbf{true}) \\
\phi_{\mathrm{demand1},r}(c) & \triangleq & \neg((.*)(p_{\neg r(c)})(.*)(\mathrm{Now})\mathbf{true}) \\
\phi_{\mathrm{demand2},r}(c) & \triangleq & \neg((\neg\phi_{\mathrm{amplify},r}(c)*)(\mathrm{Now})\mathbf{true}) \wedge \\
 & & \neg((.*)(p_{\neg r(c)})(\neg\phi_{\mathrm{amplify},r}(c)*)(\mathrm{Now})\mathbf{true})
\end{array}
$$

Fig. 19. Definitions of $\phi_{\mathrm{demand},r}$ and $\phi_{\mathrm{inspect},r}$

will yield the effect p:acct$|H$, where:

$$H = p{:}\mathrm{acct}; \phi_{\mathrm{enable},r:\mathrm{filew}}(/\mathrm{accts/ledger.txt}); p{:}\mathrm{system}; \phi_{\mathrm{inspect},r:\mathrm{filew}}(/\mathrm{accts/ledger.txt})$$

This reflects that the call and return of the application (enableit checkit) is assigned the effect p:acct, while the subsequent application to /accts/ledger.txt is assigned effect H. Assuming that both p:system and p:acct are authorised for r:filew(/accts/ledger.txt) in $\mathcal{A}$, verification will clearly succeed on this expression. On the other hand, stackification of the application checkit(/accts/ledger.txt) will generate the following trace effect:

$$p{:}\mathrm{system}; \phi_{\mathrm{inspect},r:\mathrm{filew}}(/\mathrm{accts/ledger.txt})$$

for which verification will fail: there is no required $\phi_{\mathrm{enable},r:\mathrm{filew}}(/\mathrm{accts/ledger.txt})$ on the stack.

10.2 History-based access control

History-based access control is a generalisation of Java's notion of stack inspection that takes into account all past events, not just those on the stack (Abadi & Fournet, 2003). Our language is well suited to the static typechecking of such security policies. In the basic history model of (Abadi & Fournet, 2003), some initial *current rights* are given, and with every new activation the static rights of that activation are automatically intersected with the current rights to generate the new current rights. Unlike stack inspection, removal of the activation does not return the current rights to its state prior to the activation.

We assume the same underlying model of principals as above. A demand of a resource r with parameter c, denoted $\phi_{\mathrm{demand1},r}(c)$, requires that all invoked functions possess the right for that resource. This general check may be expressed in our logic as in Fig. 19, where we assume given for verification some trace effect H containing events $ev_1(c_1), \ldots, ev_n(c_n)$. For example, validity of the following type judgement requires $r(c) \in \mathcal{A}(p_1) \cap \mathcal{A}(p_2)$:

$$\Gamma, p_1; p_2; \phi_{\mathrm{demand1},r}(c) \vdash p_1; (\lambda x.p_2; \phi_{\mathrm{demand1},r}(x))\, c : \mathit{unit}$$

The model in (Abadi & Fournet, 2003) also allows for a combination of stack- and history-based properties, by allowing the *amplification* of a right on the stack. When a right is amplified, it remains active after function return regardless of the current rights generated by the invocation, provided that amplification is performed by an authorised principal. Such assertions can be expressed in our framework using a combination of stack- and history-based assertions.

Taking the same strategy as for enabling in the stack inspection model, we can define amplification as a check event $\phi_{\mathrm{amplify},r}$; in fact, this can be defined equivalent to $\phi_{\mathrm{enable},r}$, and enforced by stackifying effects before verifying. In other words, the semantics of amplification are identical to the semantics of enabling. Then, making the simplifying assumption that by convention all resources are initially amplified, the demand predicate can be redefined in the presence of amplification as $\phi_{\mathrm{demand2},r}$ in Fig. 19. Observe that this predicate is syntactically identical to $\phi_{\mathrm{inspect},r}$, except that effects should *not* be stackified before verifying $\phi_{\mathrm{demand2},r}$. The check $\phi_{\mathrm{demand2},r}(c)$ ensures that an amplification of $r(c)$ occurs prior to it, without any interposed principal that is unauthorised for $r(c)$.

Given that history-based access control requires an interplay of stack- and history-based checks, we argue that a contribution of our model is its ability to deal with these variations in a uniform manner, within one rigorous formal framework.

10.3 Discussion

The examples applications of our system developed in this section illustrate its flexibility and power. The distinction of traces and predicates on traces allows a general class of security properties to be defined. Trace effects themselves are amenable to interpretation as abstractions of either history traces or stack traces. The examples, stack inspection and history-based access control, show particular applications in stack trace and history trace settings respectively, but any property expressible in a temporal logic could be used to specify desired history or stack trace behavior. This follows the JDK security model (Gong *et al.*, 1997) in which stack inspection is the default predicate on stack traces, but other predicates can be programmed and used instead. In practice this offers a great deal of flexibility in the kinds of security properties that can be enforced and the ability to build security "libraries" for various domains, such as the Java sandboxing model for mobile code. Again, we should emphasise our system places stack- and history-based mechanisms on a single unified foundation.

A limitation of our system is that event and check parameters must be syntactic constants. In Java, for example, privilege parameters can be dynamically generated, e.g. "file read" privileges may be paramaterized by a new filehandle. Other researchers have addressed this issue in the similat static program verification contexts, in particular (Igarashi & Kobayashi, 2002), and (Hamlen *et al.*, 2006) which considers reference monitors in the .NET security model. In (Higuchi & Ohori,

2007) the authors deal specifically with stack inspection and define privilege parameters to be either "may" sets of literal values, or a distinguished "unknown" value. Rather than generate privilege constraints that must all be statically satisfied, run time checks are inserted into the program at checks that *may* fail. A similar soft-typing approach is not precluded by our system, and extending it to treat dynamically generated constants in this manner is an interesting direction for future work.

11 Conclusion

In this section we conclude with a discussion of related work and a summary of the paper.

11.1 Related Work

Previous work relevant to the application of trace based security models has been noted in Sect. 1. Here we discuss related theories and systems designed to enforce trace-based properties of program execution.

Compile-time vs. Run-time Verification Perhaps the principal division between previous approaches to the enforcement of trace based program properties is between those systems that detect errors at run-time (Nierstrasz, 1993; Rossie, Jr., 1998; Schneider, 2000; Bauer *et al.*, 2002b; Bauer *et al.*, 2002a; Abadi & Fournet, 2003; Edjlali *et al.*, 1998), *vs.* those that detect errors at compile-time (Ball & Rajamani, 2000; Chen & Wagner, 2002; Besson *et al.*, 2001; Schmidt, 1998). Run-time approaches are more accurate since a compile-time analysis must conservatively approximate what events could occur; the compile-time analysis will also reject some safe programs, due to the need to be conservative. On the other hand, liveness properties ("all SSL sockets are eventually closed") can be verified at compile-time but not at run-time.

Run-time Verification Some early work in the area was carried out by Nierstrasz and other object-oriented language designers (Nierstrasz, 1993; Rossie, Jr., 1998). Their goal was to specify only the legal patterns of messages that could be passed to a particular object. The SSL example in Sect. 1 could be cast in that view by taking the events to be messages to some SSL control object. Regular expressions and finite automata were used for the specification logic in this work. Security automata (Schneider, 2000; Bauer *et al.*, 2002b; Bauer *et al.*, 2002a) also use finite automata for specification. Security automata are targeted at specification and verification of security policies. All of these run-time approaches assume there is a run-time monitoring process that will either abort or raise an exception if a bad trace is realized. Some automata transitions can be optimised away using simple static program analyses (Erlingsson & Schneider, 2000), but the analysis is still mostly dynamic: errors will not be realized until run-time.

Several groups (Abadi & Fournet, 2003; Edjlali *et al.*, 1998) have proposed using event traces for access control. A given access is only permitted under a restricted set of previous circumstances, recorded in the history. These projects propose no particular specification language—the policies are directly coded. The stack inspection algorithm used in the .Net CLR and Java Security Architecture is another specific access control policy based on execution history (Gong *et al.*, 1997).

Compile-time Verification The MOPS system (Chen & Wagner, 2002) compiles C programs to Push-down Automata (PDA) reflecting the program control flow, where transitions are program transitions and the automaton stack abstracts the program call stack. Similarly, the ESP system (Das *et al.*, 2002) checks temporal properties of programs (encoded as finite state machines) by simulating an approximation of property transistions that will occur at run-time. (Jensen *et al.*, 1999; Besson *et al.*, 2001) assume that some (undefined) algorithm has already converted a program to a control flow graph, expressed as a form of PDA. The Vault system (DeLine & Fähndrich, 2001) enforces resource usage constraints with *type guards*, temporal specifications of when an operator may be applied— similar in spirit to the trace checks presented here—and verification is performed via type checking that ensures well-typed programs satisfy their guard specifications.

These aforementioned abstractions work well for procedural programs, but are not powerful enough to fully address advanced language features such as higher order functions. Our approach, based on type and effect theory (Talpin & Jouvelot, 1992; Amtoft *et al.*, 1999), does allow abstract interpretation of higher order programs. Trace effects yielded by the analysis provide a conservative approximation of trace behaviour via a Labelled Transition System (LTS) interpretation. Program assertions can be expressed as temporal logical formulae that can be automatically verified by model-checking (Steffen & Burkart, 1992).

Systems have also been developed for statically verifying correctness of security automata using dependent types (Walker, 2000), and in a more general form as refinement types (Mandelbaum *et al.*, 2003). These systems do not extract any abstract interpretations, and so are in a somewhat different category than the others. They also lack type inference and do not use a logic that can be automatically verified, so user intervention is needed.

Trace Specification Logics and Model Checking Some of the aforecited systems also automatically verify assertions at compile-time via model-checking, including (Ball & Rajamani, 2000; Chen & Wagner, 2002; Besson *et al.*, 2001), though none of these define a rigorous process for extracting an LTS from higher-order 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.

Perhaps the most closely related work is (K. Marriott & Sulzmann, 2003), 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 and subtyping polymorphism, restricting expressiveness in practice, and verifies global, rather than local, assertions. Furthermore, their system analyses only history-based properties, not stack-based properties as in our system. The system of (Igarashi & Kobayashi, 2002) is based on linear types, not effect types. 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. Their specification logic is left abstract, thus they provide no automated mechanism for expressing or deciding assertions.

The systems in (Colcombet & Fradet, 2000; Besson *et al.*, 2002; Jensen *et al.*, 1999; Besson *et al.*, 2001) 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 higher-order functions and other features. Furthermore, polymorphic effects are inferable in our system and events may be parameterised 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 (Skalka *et al.*, 2005; Skalka, 2005).

Local and Global reasoning Logical assertions can be local, concerning a particular program point, or global, defining the whole behaviour required. However, access control systems (Wallach & Felten, 1998; Abadi & Fournet, 2003; Edjlali *et al.*, 1998), use local checks. Since we are interested in the static enforcement of access control mechanisms (see Sect. 10), the focus in this paper is on local, compile-time checkable assertions, though in principle the verification of global properties is also possible in our system. Related work has also modified our basic approach to enforce "policy framings" that support so-called local liveness properties (Bartoletti *et al.*, 2005b).

11.2 Summary

This paper establishes a foundation for the enforcement of local well-formedness properties on program event traces. Our language model λ_{trace} extends a functional core with constructs to label program points, *aka* events, and with local assertions over program event traces up to the point of the assertion during evaluation. The language model is parameterised by a logic of event trace assertions and has been instantiated here with the linear μ-calculus.

We have developed a static type and effect theory for λ_{trace} programs, whereby computational traces arising at run-time can be conservatively approximated by trace effects, components of the type language that are interpreted as model-checkable LTSs. We have presented two variations on the system: the first uses a unified term syntax, while the second uses constraint subtyping. Both include parametric let-polymorphism over types and effects. The unified representation allows for weakening of trace effects through trace containment, whereas the constraint form allows for more expressive type subsumption. The constraint system is shown to be a conservative extension of both the unified term system and the underlying effect-free ML-style type system. Since trace effects can be model-checked, verification of trace effect approximations obtained by typing guarantees that all trace assertions will hold during execution of a given program. Thus, our type safety result guarantees that statically well-typed programs do not contain assertions that will fail dynamically.

Type reconstruction algorithms have been defined for both type systems. Type inference for constraint subtyping has been shown to be sound and complete, the latter result entailing a principal types property. Inference is decidable, despite the general undecidability of trace effect equality and containment, due to a restricted form of constraints generated by inference. Term forms of trace effects are extractable from effect constraints via the *hextract* algorithm, obtaining a compliant form for existing model checking techniques.

We have shown that trace effects can be transformed to reflect a stack discipline for function calls and returns via the *stackify* algorithm, allowing the expression of stack-based policies such as stack inspection and rights amplification for history-based access control. We have also shown how these policies can be defined in our system, as particular applications of a general framework for enforcing trace-based properties expressible in temporal logics. A primary strength of our approach is that the analysis is fully automated—no program abstraction or type annotations are presupposed—and each component of this automation has been rigorously justified. Our static analysis and verification techniques for trace based properties in higher order languages provide well-founded, flexible, and general tools for enforcing program safety and security.

References

Abadi, M., & Fournet, C. (2003). Access control based on execution history. *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03).*

Amtoft, T., Nielson, F., & Nielson, H. R. (1999). *Type and effect systems: Behaviours for concurrency.* Imperial College Press.

Ball, T., & Rajamani, S. K. (2000). Bebop: A symbolic model checker for boolean programs. *Pages 113–130 of: SPIN.*

Bartoletti, M., Degano, P., & Ferrari, G. L. (2005a). Enforcing secure service composition. *Pages 211–223 of: CSFW.* IEEE Computer Society.

Bartoletti, M., Degano, P., & Ferrari, G. L. (2005b). History-based access control with local policies. *Pages 316–332 of:* Sassone, Vladimiro (ed), *FoSSaCS.* Lecture Notes in Computer Science, vol. 3441. Springer.

Bauer, L., Ligatti, J., & Walker, D. (2002a). More enforceable security policies. *Proceedings of the foundations of computer security workshop.*

Bauer, L., Ligatti, J., & Walker, D. (2002b). Types and effects for non-interfering program monitors. *Proceedings of the International Symposium on Software Security.*

Besson, F., Jensen, T., Métayer, D. Le, & Thorn, T. (2001). Model checking security properties of control flow graphs. *J. Computer Security*, **9**, 217–250.

Besson, F., de Grenier de Latour, T., & Jensen, T. (2002). Secure calling contexts for stack inspection. *Pages 76–87 of: PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming.*

Burkart, O., Caucal, D., Moller, F., & Steffen, B. (2001). Verification on infinite structures. J. Bergstra, A. Pons, S. Smolka (ed), *Handbook on process algebra*. North-Holland.

Chen, H., & Wagner, D. (2002). MOPS: an infrastructure for examining security properties of software. *Pages 235–244 of: CCS '02: Proceedings of the 9th ACM conference on Computer and communications security.*

Colcombet, T., & Fradet, P. (2000). Enforcing trace properties by program transformation. *Pages 54–66 of: POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.*

Das, M., Lerner, S., & Seigle, M. (2002). ESP: path-sensitive program verification in polynomial time. *Pages 57–68 of: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation.*

DeLine, R., & Fähndrich, M. (2001). Enforcing high-level protocols in low-level software. *Pages 59–69 of: PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation.*

Edjlali, G., Acharya, A., & Chaudhary, V. (1998). History-based access control for mobile code. *Pages 38–48 of: ACM conference on computer and communications security.*

Eifrig, J., Smith, S., & Trifonov, V. (1995). Type inference for recursively constrained types and its application to OOP. *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*. Electronic Notes in Computer Science, vol. 1. Elsevier.

Erlingsson, Ú., & Schneider, F. B. (2000). SASI enforcement of security policies: a retrospective. *Pages 87–95 of: NSPW '99: Proceedings of the 1999 workshop on new security paradigms.*

Esparza, J. (1994). On the decidability of model checking for several mu-calculi and Petri nets. *Proceeding of CAAP '94*. Lecture Notes in Computer Science, vol. 787.

Esparza, J., Kucera, A., & Schwoon, S. (2001). Model-checking LTL with regular valuations for pushdown systems. *TACS: 4th international conference on Theoretical aspects of computer software.*

Foster, J. S., Terauchi, T., & Aiken, A. (2002). Flow-Sensitive Type Qualifiers. *Pages 1–12 of: Proceedings of the 2002 ACM SIGPLAN conference on Programming language design and implementation.*

Gong, L., Mueller, M., Prafullchandra, H., & Schemers, R. (1997). Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. *Pages 103–112 of: USENIX symposium on internet technologies and systems.*

Hamlen, Kevin W., Morrisett, Greg, & Schneider, Fred B. (2006). Certified in-lined reference monitoring on .net. *Pages 7–16 of: Plas '06: Proceedings of the 2006 workshop on programming languages and analysis for security*. New York, NY, USA: ACM Press.

Higuchi, Tomoyuki, & Ohori, Atsushi. (2007). A static type system for jvm access control. *Acm trans. program. lang. syst.*, **29**(1), 4.

Igarashi, A., & Kobayashi, N. (2002). Resource usage analysis. *Pages 331–342 of: Conference record of POPL'02: The 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.*

Jensen, T., Métayer, D. Le, & Thorn, T. (1999). Verification of control flow based security properties. *Proceedings of the 1999 IEEE symposium on Security and privacy.*

K. Marriott, P. J. Stuckey, & Sulzmann, M. (2003). Resource usage verification. *Proc. of first Asian programming languages symposium, APLAS 2003.*

Kozen, D. (1983). Results on the propositional mu-calculus. *Theoretical computer science*, **27**(Dec.), 333–354.

Mandelbaum, Y., Walker, D., & Harper, R. (2003). An effective theory of type refinements. *Proceedings of the the eighth ACM SIGPLAN international conference on Functional programming (ICFP'03).*

Nierstrasz, O. (1993). Regular types for active objects. *Pages 1–15 of: Proceedings of the OOPSL '93 conference on Object-oriented programming systems, languages, and applications.*

Palsberg, J., & O'Keefe, P. (1995). A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, **17**(4), 576–599.

Pierce, B. C. (2002). *Types and programming languages.* The MIT Press. Chap. 22.

Pottier, F., Skalka, C., & Smith, S. (2001). A systematic approach to static access control. *Pages 30–45 of:* Sands, D. (ed), *Proceedings of the 10th European symposium on programming (ESOP'01).* Lecture Notes in Computer Science, vol. 2028. Springer Verlag.

Rossie, Jr., J. G. (1998). Logical observable entities. *Pages 154–165 of: OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.*

Schmidt, D. A. (1998). Trace-based abstract interpretation of operational semantics. *Lisp and symbolic computation*, **10**(3), 237–271.

Schneider, F. B. (2000). Enforceable security policies. *Information and system security*, **3**(1), 30–50.

Skalka, C. (2005). Trace effects and object orientation. *Pages 139–150 of: PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming.*

Skalka, C., & Pottier, F. (2003). Syntactic type soundness for HM(X). *Electronic notes in theoretical computer science*, **75**.

Skalka, C., & Smith, S. (2000). Static enforcement of security with types. *Pages 34–45 of: Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00).*

Skalka, C., & Smith, S. (2004). History effects and verification. *Asian programming languages symposium.*

Skalka, C., Smith, S., & Van Horn, D. 2005 (January). A type and effect system for flexible abstract interpretation of Java. *Proceedings of the ACM workshop on Abstract interpretation of object oriented languages.* Electronic Notes in Theoretical Computer Science.

Steffen, B., & Burkart, O. (1992). Model checking for context-free processes. *Pages 123–137 of: CONCUR'92, Stony Brook (NY).* Lecture Notes in Computer Science (LNCS), vol. 630. Heidelberg, Germany: Springer-Verlag.

Stone, C. (2000). *Singleton types and singleton kinds.* Tech. rept. CMU-CS-00-153. Carnegie Mellon University.

Sulzmann, M. (2001). A general type inference framework for Hindley/Milner style systems. *Pages 246–263 of: International symposium on Functional and logic programming.* Lecture Notes in Computer Science, vol. 2024. Springer Verlag.

Talpin, J.-P., & Jouvelot, P. (1992). The type and effect discipline. *Pages 162–173 of: Seventh annual IEEE symposium on Logic in computer science, Santa Cruz, California.* Los Alamitos, California: IEEE Computer Society Press.

Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of mathematics*, **5**(2), 285–309.

Trifonov, V., & Smith, S. (1996). Subtyping constrained types. *Pages 349–365 of: Proceedings of the third international Static analysis symposium*, vol. 1145. Springer Verlag.

Van Horn, D. (2006a). *Algorithmic trace effect analysis.* M.Phil. thesis, University of Vermont.

Van Horn, D. (2006b). *Trace effect analyis, OCaml implementation.* `http://www.cs.uvm.edu/~dvanhorn/trace/`.

Walker, D. (2000). A type system for expressive security policies. *Pages 254–267 of: Conference record of POPL'00: The 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.*

Wallach, D. S., & Felten, E. (1998). Understanding Java stack inspection. *Proceedings of the 1998 IEEE symposium on security and privacy.*