Under consideration for publication in J. Functional Programming

An Extended Account of Contract Monitoring Strategies as Patterns of Communication

Electronic Appendix: Type Safety

CAMERON SWORDS and AMR SABRY and SAM TOBIN-HOCHSTADT
Indiana University, Indiana, United States
(*e-mail:* {cswords,sabry,samth}@indiana.edu)

A Type Safety for $\lambda^{\rightrightarrows}_{/c}$

Type safety follows Reppy (1993)—we present a type system for the term language and define "well-formed" process configurations based on well-formed terms, and use traces to define type safety for a process collection.

Our term-level type system, presented in Figure A 2, is standard: we provide direct types for our built-in features, including base types, form types, and our contract combinators (otherwise we would require universal quantification to, e.g., type **pred/c** as $\forall\alpha.\texttt{con}\ \alpha \to \alpha \to \alpha$). Our type judgments include two environments, Γ and Δ, which give types to variables and channels, respectively.

We include errors of the form **raise** e, typing them at any appropriate type (to allow them to occur anywhere as necessary). We also include a `blame` type for opaque blame values B, which are akin to ML's `exn` types (Milner, 1997) insofar as they are only used to carry error information in the case that an exception is raised (i.e., a contract violation occurs).

The only other term-level oddity is **delay**: to ease the return type of **check**, we type a delayed expression at its internal type; otherwise, we would need to parameterize **check**'s return type based on the input strategy, adding additional user complexity. Since many real-world software contract systems target dynamically-typed languages (Findler & Felleisen, 2002; Findler *et al.*, 2008; Tobin-Hochstadt & Felleisen, 2010), this seems a practical allowance. This allowance means that we need to consider a class of *unforced* terms, however, such as $(\textbf{delay}\ (\lambda x.\ e))\ e'$, which are well-typed but irreducible. This definition is given in Figure A 3.

To type process configurations, we first define *well-formed* configurations:

Definition A.1 (*Well-Formed Configuration*)
A process configuration K, T, P is well-formed if

- for each process $\langle e \rangle_\pi$,
  - — e contains no free variables,
  - — there is no $e' \equiv e$ such that $\langle e' \rangle_\pi \in P$,
- the set of terminating process ids $T \subseteq \mathrm{dom}(P)$;
- and the set of free channels $FC(P) \subseteq K$.

The idea is that a process configuation is well-formed if each individual process is well-formed, each process has a unique process identification number π, T contains only process

Syntax

TYPES	τ	:=	$\texttt{int} \mid \texttt{bool} \mid \texttt{blame} \mid \texttt{unit} \mid \tau \to \tau \mid \tau + \tau \mid \tau \times \tau$
		$\mid$	$\texttt{chan}\ \tau$
TYP ENVS.	Γ	:=	$\cdot \mid (x,\tau),\Gamma$
CHAN ENVS.	Δ	:=	$\cdot \mid (\iota,\tau),\Gamma$
PROC TYPES.	ρ	:=	$\cdot \mid (\pi,\tau),\rho$

Fig. A 1: Type definitions for $\lambda^{\Rightarrow}_{/c}$.

identification numbers for processes in P, and the channels that occur in P are contained in the associated channel list K.

We type processes with a map ρ from process ids π to associated types τ as:

Definition A.2 (*Process Typing*)
A well-formed configuration K,T,P has type ρ under channel environment Δ, written

$$\Delta \vdash K,T,P : \rho$$

if:

- $K \subseteq \mathrm{dom}(\Delta)$
- $\mathrm{dom}(P) \subseteq \mathrm{dom}(\rho)$
- for each $\langle e \rangle_\pi \in P$, $\cdot, \Delta \vdash e : \rho(\pi)$

Furthermore, observe that well-formedness is closed under reduction:

Lemma A.1
If K,T,P is well-formed under some ρ and Δ and $K,T,P \Rightarrow K',T',P'$, then there exist some Δ' and ρ' such that $\Delta \subseteq \Delta'$, $\rho \subseteq \rho'$, and $\Delta' \vdash K',T',P' : \rho'$ (e.g., K',T',P' is well-formed).

Proof
By inversion on $\Rightarrow$. □

Corollary A.1
The properties above hold for $\Rightarrow^*$

Proof
By induction on the length of the evaluation sequence. □

Traces. To prove process-level type safety, we introduce the notion of traces (Reppy, 1993) to deal with the non-deterministic nature of '$\Rightarrow$' in $\lambda^{\Rightarrow}_{/c}$. Note that we use $\pi 0$ to indicate the process identifier of the initial process in any given computation. We define traces as:

Definition A.3 (*Trace*)

A trace $\mathscr{T}$ is a (possibly infinite) sequence of *well-formed* configurations

$$\mathscr{T} = \langle K_0, T_0, P_0; K_1, T_1, P_1; \cdots \rangle$$

such that $K_i, T_i, P_i \Rightarrow K_{i+1}, T_{i+1}, P_{i+1}$.

By Corollary A.1, if K_0, T_0, P_0 is well-formed, then any sequence of evaluation steps starting with K_0, T_0, P_0 is a trace. Moreover, the *possible* states of a process with respect to a configuration may be defined as:

Definition A.4 (*Process States*)
Let P be a well-formed process set and let $\langle e \rangle_\pi \in P$. The *state* of π in P is either *zombie*, *unforced*, *blocked*, or *ready*, depending on the form of e:

- if $e \in v$, then it is a *zombie*;
- if *unforced*(e), then it is *unforced*;
- if $e = E[e_0]$ and $e_0 =$ **read** ι or $e_0 =$ **write** ι v and there does not exist some $\langle E'[e'] \rangle_{\pi'} \in P$ with $e_0 \overset{\iota}{\frown} e_1$, then π is *blocked* in P;
- otherwise, π is *ready* in P.

We define the set of *ready* processes as *Ready*(P).

Next, we say that a configuration P is terminal as:

Definition A.5 (*Terminal Configuration*)
a configuration P is a *terminal configuration* if *Ready*$(P) = \emptyset$.

Furthermore, any terminal configuration with blocked processes is *deadlocked*. Finally, recall that *unforced*(e) describes only those terms that are *unforced* due to misused **delay** expressions, and do not otherwise describe runtime type errors.

Now, we define a computation as:

Definition A.6 (*Computation*)
A *computation* is a maximal trace, such that it is infinite or it is finite and ends in a terminal configuration. If e is a term, then we define the computations of e to be:

$$\mathit{Comp}(e) = \{\mathscr{T} \mid \mathscr{T} \text{ is a trace with head } \emptyset, \{\pi 0\}, \langle e \rangle_{\pi 0}\}$$

Next, we describe the set of trace processes:

Definition A.7 (*Trace Processes*)

$$\mathit{Procs}(\mathscr{T}) = \{\pi \mid \exists K_i, T_i, P_i \in \mathscr{T} \text{ with } \pi \in \mathrm{dom}(P_i)\}$$

This allows us to describe each process that will appear over the course of a reduction. Finally, our nondeterministic reduction semantics forces us to take on notions of convergence and divergence *relative to a particular computation of a program*:

Definition A.8 (*Convergence and Divergence*)
A process $\pi \in \mathit{Procs}(\mathscr{T})$:

- *converges to a value* v in $\mathscr{T}$, written $\pi \Downarrow_{\mathscr{T}} v$, if $K, T, P + \langle v \rangle_\pi \in \mathscr{T}$;

- is *unforced* in T if it evaluates to some expression e in $\mathscr{T}$, written $\pi \Downarrow_{\mathscr{T}} e$, if $K, T, P + \langle e \rangle_\pi \in \mathscr{T}$ and $\mathit{unforced}(e)$;
- *converges to an error* **raise** B in $\mathscr{T}$, written $\pi \Downarrow_{\mathscr{T}}$ **raise** B, if $K, T, P + \langle \textbf{raise } B \rangle_\pi \in \mathscr{T}$.
- *diverges* in $\mathscr{T}$, written $\pi \Uparrow_{\mathscr{T}}$, if for every $K, T, P \in \mathscr{T}$, with $\pi \in \mathrm{dom}(P)$, π is *ready* or *blocked*.

Our divergence includes deadlocked processes and processes that are not evaluated enough to terminate, as well as those that enter infinite loops.

Type Safety. Finally, we define type safety for term languages via progress and preservation:

Lemma A.2 (*Term Progress*)
If there exists Δ such that $\cdot, \Delta \vdash e : \tau$, then either:

- $\exists e', e \longmapsto e'$
- $e \in v$
- $e =$ **raise** B, for some B
- $\mathit{unforced}(e)$
- or e is a process reduction.

Proof
Straightforward, by induction on e. □

Lemma A.3 (*Term Preservation*)
If there exists Δ and e' such that $\cdot, \Delta \vdash e : \tau$ and $e \longmapsto e'$, then $\cdot, \Delta' \vdash e' : \tau$.

Proof
Straightforward, by inversion on $\longmapsto$ (and therein, $\rightarrow$). □

Next, we define preservation for process collections:

Lemma A.4 (*Concurrent Type Preservation*)
If a configuration is well-formed with $K, T, P \Rightarrow K', T', P'$, and there exists Δ such that $\Delta \vdash K, T, P : \rho$, then there is some channel typing Δ' and process typing ρ' such that:

- $\Delta \subseteq \Delta'$
- $\rho \subseteq \rho'$
- $\Delta' \vdash K', T', P' : \rho'$
- $\Delta' \vdash K', T', P' : \rho$

Proof (Sketch)
The fourth property follows from the first three; the others proceed by induction on $K, T, P \Rightarrow K', T', P'$. □

Next, we may classify any given expression:

Lemma A.5 (*Uniform Evaluation*)
If e is an expression with some trace $\mathscr{T} \in \mathit{Comp}(e)$ where $\pi \in \mathit{Procs}(\mathscr{T})$, then either $\pi \Uparrow_{\mathscr{T}}$, $\pi \Downarrow_{\mathscr{T}} v$, $\pi \Downarrow_{\mathscr{T}}$ **raise** v, $\pi \Downarrow_{\mathscr{T}} e'$ with $\mathit{unforced}(e')$, or $P_i(\pi)$ is stuck for some $K_i, T_i, P_i \in \mathscr{T}$.

Proof (Sketch)
This follows immediately from the definitions. □

Our next lemma states that any stuck, but not *unforced* term, untypable:

Lemma A.6 (*Untypability of Wrong Configurations*)
If $P(\pi)$ is irreducible and not *unforced*(e) in a well-formed configuration K,T,P, then there are not some Δ, ρ such that $\cdot,\Delta \vdash P(\pi) : \rho(\pi)$. In other words, K,T,P is untypable.

Proof (Sketch)
Proof proceeds by assuming, toward a contradiction that there are some Δ and ρ that correctly type $P(\pi)$. Then $P(\pi) = E[e']$ for some E and e', and it suffices to show that e' is untypable, which is a contradiction. Proof proceeds by induction on the possible structures of e', demonstrating that each redex may be reduced if it is typable, a contradiction since it is stuck, and thus the redex must not be typable. □

Next, we show syntactic soundness by first demonstrating uniform evaluation, e.g., that every program is in one of four state.

Theorem A.1 (*Syntactic Soundness*)
Let e be an expression with $\cdot,\cdot \vdash e : \tau$. Then for any $\mathscr{T} \in Comp(e)$, $\pi \in Procs(\mathscr{T})$, with K_i,T_i,P_i the first occurrence of π in $\mathscr{T}$, there exist Δ,ρ such that

$$\Delta \vdash K_i,T_i,P_i : \rho$$

and $\rho(\pi 0) = \tau$. Then either:

- $\pi \Uparrow_{\mathscr{T}}$
- $\pi \Downarrow_{\mathscr{T}} v$ such that there exists $\Delta',\Delta \subseteq \Delta'$ and $\cdot,\Delta' \vdash v : \rho(\pi)$
- $\pi \Downarrow_{\mathscr{T}}$ **raise** B
- $\pi \Downarrow_{\mathscr{T}} e', unforced(e')$ such that there exists $\Delta',\Delta \subseteq \Delta'$ and $\cdot,\Delta' \vdash e' : \rho(\pi)$

Proof
The existence of Δ and ρ follow from Lemma A.4. By the uniform evaluation lemma (Lemma A.5),we know that then either $\pi \Uparrow_{\mathscr{T}}$, $\pi \Downarrow_{\mathscr{T}} v$, $\pi \Downarrow_{\mathscr{T}}$ **raise** v, $\pi \Downarrow_{\mathscr{T}} e'$ with *unforced*(e'), or $P_i(\pi)$ is stuck for some $K_j,T_j,P_j \in \mathscr{T}$.

Assume toward a contradiction that π is stuck in K_j,T_j,P_j. By Lemma A.1, K_j,T_j,P_j is well-formed, and, by Lemma A.6, it must be untypable. But, since the configuration $\emptyset,\{\pi_0\},\langle e\rangle_{\pi_0}$ is typable, by Lemma A.4, there is a Δ', ρ' such that $\Delta' \vdash K_j,P_j,T_j : \rho'$, a contradiction. Thus π cannot be stuck.

Otherwise, $\pi \Uparrow_{\mathscr{T}}$, $\pi \Downarrow_{\mathscr{T}} v$, $\pi \Downarrow_{\mathscr{T}}$ **raise** v, or $\pi \Downarrow_{\mathscr{T}} e'$ with *unforced*(e').

If $\pi \Uparrow_{\mathscr{T}}$ or $\pi \Downarrow_{\mathscr{T}}$ **raise** B, are are done (in the former case due to divergence and in he latter because **raise** B is well-typed at any type).

If $\pi \Downarrow_{\mathscr{T}} v$, then let $K_j,T_j,P_j \in \mathscr{T}$ such that $P_j(\pi) = v$. By Lemma **??**, there is some Δ' and ρ' such that $\Delta \subseteq \Delta'$ and $\rho \subseteq \rho'$ such that $\Delta' \vdash P_j : \rho'$. Since $\rho \subseteq \rho'$, $\rho(\pi) = \rho'(\pi)$, and so $\cdot,\Delta' \vdash v : \rho(\pi)$.

If $\pi \Downarrow_{\mathscr{T}} e', unforced(e')$, then let $K_j,T_j,P_j \in \mathscr{T}$ such that $P_j(\pi) = e'$. By Lemma **??**, there is some Δ' and ρ' such that $\Delta \subseteq \Delta'$ and $\rho \subseteq \rho'$ such that $\Delta' \vdash P_j : \rho'$. Since $\rho \subseteq \rho'$, $\rho(\pi) = \rho'(\pi)$, and so $\cdot,\Delta' \vdash v : \rho(\pi)$. □

Before stating soundness, we define evaluation of a process identification in a trace as

$$\begin{array}{lll} \mathit{eval}_{\mathscr{T}}(\pi) = & v & \text{if } P(\pi) \Downarrow_{\mathscr{T}} v \\ \mathit{eval}_{\mathscr{T}}(\pi) = & e' & \text{if } P(\pi) \Downarrow_{\mathscr{T}} e', \text{ with } \mathit{unforced}(e') \\ \mathit{eval}_{\mathscr{T}}(\pi) = & \text{WRONG} & \text{if } P(\pi) \Downarrow_{\mathscr{T}} e, \text{ with } e \text{ stuck.} \end{array}$$

Finally, we state soundness:

Theorem A.2 (*Soundness*)
If e is a program with $\cdot, \cdot \vdash e : \tau$, then for any computation $\mathscr{T} \in \mathit{Comp}(E)$ and any process ID $\pi \in \mathit{Procs}(\mathscr{T})$:

(a) If $\mathit{eval}_T(\pi) = v$ and K_i, T_i, P_i is the first occurrence of π in $\mathscr{T}$, then for any Δ, ρ such that $\Delta \vdash K_i, T_i, P_i : \rho$ and $\rho(\pi 0) = \tau$, then there exists $\Delta', \Delta \subseteq \Delta'$ such that $\cdot, \Delta' \vdash v : \rho(\pi)$
(b) If $\mathit{eval}_T(\pi) = e'$ with $\mathit{unforced}(e')$ and K_i, T_i, P_i is the first occurrence of π in $\mathscr{T}$, then for any Δ, ρ such that $\Delta \vdash K_i, T_i, P_i : \rho$ and $\rho(\pi 0) = \tau$, then there exists $\Delta', \Delta \subseteq \Delta'$ such that $\cdot, \Delta' \vdash e' : \rho(\pi)$
(c) $\mathit{eval}_{\mathscr{T}}(\pi) \neq \text{WRONG}$

The proof follows directly from Theorem A.1 and the definition of *eval*.

References

Findler, Robert Bruce, & Felleisen, Matthias. (2002). Contracts for higher-order functions. *Proceedings of the 7th International Conference on Functional Programming*. ICFP '02. ACM.

Findler, Robert Bruce, Guo, Shu-Yu, & Rogers, Anne. (2008). Lazy contract checking for immutable data structures. *Implementation and Application of Functional Languages*. Springer-Verlag.

Milner, Robin. (1997). *The definition of standard ml: revised*. MIT press.

Reppy, John H. (1993). Concurrent ML: Design, application and semantics. *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*. Springer-Verlag.

Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2010). Logical types for untyped languages. *Proceedings of the 15th International Conference on Functional Programming*. ICFP '10. ACM.

Term Typing Judgments

$\boxed{\Gamma;\Delta \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma;\Delta \vdash x : \tau} \qquad \frac{\Gamma;\Delta \vdash e_1 : \tau_1 \to \tau \quad \Gamma;\Delta \vdash e_2 : \tau_1}{\Gamma;\Delta \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma;\Delta \vdash e_1 : \texttt{bool} \quad \Gamma;\Delta \vdash e_2 : \tau \quad \Gamma;\Delta \vdash e_3 : \tau}{\Gamma;\Delta \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}$$

$$\frac{\delta_\tau(\mathit{binop}) = \tau_1 \to \tau_2 \to \tau \quad \Gamma;\Delta \vdash e_1 : \tau_1 \quad \Gamma;\Delta \vdash e_2 : \tau_2}{\Gamma;\Delta \vdash e_1\ \mathit{binop}\ e_2 : \tau} \qquad \frac{\delta_\tau(\mathit{unop}) = \tau_1 \to \tau \quad \Gamma;\Delta \vdash e : \tau_1}{\Gamma;\Delta \vdash \mathit{unop}\ e : \tau}$$

$$\frac{\Gamma;\Delta \vdash e_1 : \tau_1 \quad \Gamma;\Delta \vdash e_2 : \tau_2}{\Gamma;\Delta \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma;\Delta \vdash e : \tau_1 \times \tau_2}{\Gamma;\Delta \vdash \textbf{fst}\ e : \tau_1} \qquad \frac{\Gamma;\Delta \vdash e : \tau_1 \times \tau_2}{\Gamma;\Delta \vdash \textbf{snd}\ e : \tau_2}$$

$$\frac{\Gamma;\Delta \vdash e : \tau_1 + \tau_2 \quad (x_1, \tau_1), \Gamma;\Delta \vdash e_1 : \tau \quad (x_2, \tau_2), \Gamma;\Delta \vdash e_2 : \tau}{\Gamma;\Delta \vdash \textbf{case}\ (e;\ \textbf{inl}\ x_1 \rhd e_1;\ \textbf{inr}\ x_2 \rhd e_2) : \tau}$$

$$\frac{\Gamma;\Delta \vdash e : \tau_1}{\Gamma;\Delta \vdash \textbf{inl}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma;\Delta \vdash e : \tau_2}{\Gamma;\Delta \vdash \textbf{inr}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma;\Delta \vdash e : \tau}{\Gamma;\Delta \vdash \textbf{force}\ e : \tau} \qquad \frac{\Gamma;\Delta \vdash e : \texttt{blame}}{\Gamma;\Delta \vdash \textbf{raise}\ e : \tau} \qquad \frac{\Gamma;\Delta \vdash e_1 : \texttt{blame} \to \tau \quad \Gamma;\Delta \vdash e_2 : \tau}{\Gamma;\Delta \vdash \textbf{catch}\ e_1\ e_2 : \tau}$$

$$\frac{\Gamma;\Delta \vdash e : \tau}{\Gamma;\Delta \vdash \textbf{fspawn}\ e : \texttt{unit}} \qquad \frac{\Gamma;\Delta \vdash e : \tau}{\Gamma;\Delta \vdash \textbf{spawn}\ e : \texttt{unit}} \qquad \frac{\Gamma;\Delta \vdash e : \texttt{chan}\ \tau' \to \tau}{\Gamma;\Delta \vdash \textbf{chan}e : \tau}$$

$$\frac{\Gamma;\Delta \vdash e : \texttt{chan}\ \tau}{\Gamma;\Delta \vdash \textbf{read}\ e : \tau} \qquad \frac{\Gamma;\Delta \vdash e_1 : \texttt{chan}\ \tau \quad \Gamma;\Delta \vdash e_2 : \tau}{\Gamma;\Delta \vdash \textbf{write}\ e_1\ e_2 : \texttt{unit}}$$

$$\frac{(x, \tau_1), \Gamma;\Delta \vdash e : \tau}{\Gamma;\Delta \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau} \qquad \frac{\Delta(\iota) = \tau}{\Gamma;\Delta \vdash \iota : \texttt{chan}\ \tau} \qquad \frac{\Gamma;\Delta \vdash e : \tau}{\Gamma;\Delta \vdash \textbf{delay}\ e : \tau} \qquad \frac{}{\Gamma;\Delta \vdash n : \texttt{int}}$$

$$\frac{}{\Gamma;\Delta \vdash \textbf{true} : \texttt{bool}} \qquad \frac{}{\Gamma;\Delta \vdash \textbf{false} : \texttt{bool}} \qquad \frac{}{\Gamma;\Delta \vdash \textbf{unit} : \texttt{unit}} \qquad \frac{}{\Gamma;\Delta \vdash b : \texttt{blame}}$$

Fig. A 2: Term-Language Typing Judgements for $\lambda^{\Rightarrow}_{/c}$.

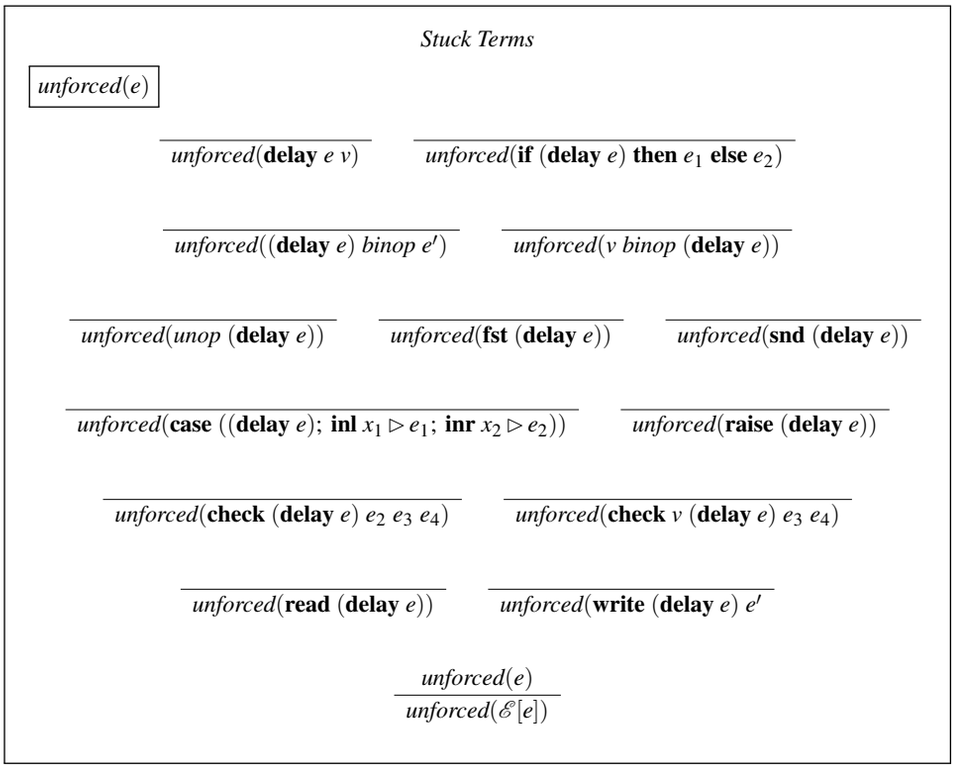

Fig. A 3: Unforced terms due to misplaced delay expressions for $\lambda^{\Rightarrow}_{/c}$.