# Blame Prediction: Technical report

Dries Harnie, Christophe Scholliers, and Wolfgang De Meuter

Vrije Universiteit Brussel

This document is a supplement to the paper *Blame Prediction: A Technique for Predicting Must-fail Errors*. It aims to explain how blame prediction can be extended to deal with programs that feature recursive functions (section 1) and (limited) mutation (section 2).

## 1 Recursion

Having explained how blame prediction can be applied to non-recursive programs, this section will demonstrate how to support recursion. Recall that, during check inference, function application expressions give rise to type function applications at the type level. In a recursive function however, the function type can contain recursive applications of itself. Expanding these applications naïvely results in an infinite process. For example, the type of the `factorial` function with an explicit accumulator variable is:

$$\alpha_{fac} = \Pi\alpha_n, \alpha_a.(\alpha_a \vee (\mathsf{int} \mathbin{?=} \alpha_n) \rightarrow (\mathsf{int} \mathbin{?=} \alpha_a) \rightarrow (\alpha_{fac}\ \mathsf{int}\ \mathsf{int}))$$

Expanding the function type application $(\alpha_{fac}\ \mathsf{int}\ \mathsf{int})$ then yields an infinite process:

$$\begin{aligned}(\alpha_{fac}\ \mathsf{int}\ \mathsf{int}) &= \mathsf{int} \vee (\mathsf{int} \mathbin{?=} \mathsf{int}) \rightarrow (\mathsf{int} \mathbin{?=} \mathsf{int}) \rightarrow (\alpha_{fac}\ \mathsf{int}\ \mathsf{int}) \\ &= \mathsf{int} \vee (\alpha_{fac}\ \mathsf{int}\ \mathsf{int}) \\ &= \mathsf{int} \vee \mathsf{int} \vee (\alpha_{fac}\ \mathsf{int}\ \mathsf{int}) \\ &= \ldots\end{aligned}$$

Note that expansions after the first do not contribute any new return types; this is the core insight to resolve these infinite types.

### 1.1 Syntax

We start by adding the letrec keyword to the Scheme$_\beta$ syntax. The binding groups in the letrec define variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$ which are bound to expressions $\mathsf{e}_1, \ldots, \mathsf{e}_n$ (usually functions). In the body as well as every $\mathsf{e}_i$, the variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$ are in scope.

$$\mathsf{e} \in \mathrm{Exp} = \ldots \mid (\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1] \ldots [\mathsf{x}_n\ \mathsf{e}_n])\ \mathsf{e})$$

The expressions $\mathsf{e}_i$ can be any kind of expression. However, to simplify the presentation of the check inference rules we restrict these to operate on groups of mutually recursive function expressions. There exists a transformation to split up a big letrec into a series of nested lets and letrecs containing mutually recursive functions. This transformation is defined in [1, Section 6.2.8] and [2] among others.

### 1.2 Check Inference and Mobility

Inferring types for a letrec expression is different from a let expression, as the functions in the binding part of the letrec can refer to both themselves and other functions bound in the letrec. The types inferred from this process will contain type function applications of the other functions in the letrec. These type function applications cannot simply be expanded, as the expansions will again contain type function applications, yielding an *infinite type*. These infinite types need to be reduced to finite types before they can be used in the letrec body.

Figure 1 defines the check inference rule T-LETREC. First, the various expressions in the binding part of the letrec are check-inferred, with the variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$ bound to fresh type variables $\alpha_1, \ldots, \alpha_n$. This yields *infinite* types $\overline{\tau_1}, \ldots, \overline{\tau_n}$. These types are converted to finite types $\tau_1, \ldots, \tau_n$ using the Solve function (shown later in this section). Finally, the type of the body is inferred with $\mathsf{x}_1, \ldots, \mathsf{x}_n$ bound to the finite types, and the final type of the letrec becomes the type inferred for the body. Unlike the let expression, the type of the body does not need to be Chain-ed to any type tests made in the binding part, as it can only contain function expressions.

Figure 1 also defines a check mobility rule F-LETREC which is functionally identical to check mobility for let expressions. Since all the expressions in the binding parts are functions, they cannot contribute any preconditions. As with let, preconditions propagated by the body are masked against the variables in the binding part.

$$\frac{\begin{array}{c}\alpha_1, \ldots, \alpha_n \text{ fresh} \qquad \Gamma, \mathsf{x}_1 : \alpha_1, \ldots, \mathsf{x}_n : \alpha_n \vdash \mathsf{e}_i : \overline{\tau_i} \qquad \forall i \in 1 \ldots n \\ \tau_i = \text{Solve}(\overline{\tau_i}; \alpha_1 : \overline{\tau_1}, \ldots, \alpha_n : \overline{\tau_n}) \qquad \forall i \in 1 \ldots n \\ \Gamma, \mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n \vdash \mathsf{e} : \tau\end{array}}{\Gamma \vdash (\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1] \ldots [\mathsf{x}_n\ \mathsf{e}_n])\ \mathsf{e}) : \tau} \quad \text{(T-LETREC)}$$

$$\frac{\begin{array}{c}\mathsf{e}_i \to \mathsf{e}_i' \uparrow \#\mathsf{t} \qquad \forall i \in 1 \ldots n \\ \mathsf{e} \to \mathsf{e}' \uparrow \mathsf{p} \qquad \mathsf{p}' = \text{mask}(\mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{p})\end{array}}{(\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1] \ldots [\mathsf{x}_n\ \mathsf{e}_n])\ \mathsf{e}) \to \quad (\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1'] \ldots [\mathsf{x}_n\ \mathsf{e}_n'])\ [\mathsf{p}]\mathsf{e}') \uparrow \mathsf{p}'} \quad \text{(F-LETREC)}$$

Fig. 1: Check inference and mobility for letrec

### 1.3 Paths

In the next section we define the Solve function. This function operates on a set of *paths*, which contain the type tests generated by following along a single path in the function and its eventual return type. We start off by defining paths and their operations.

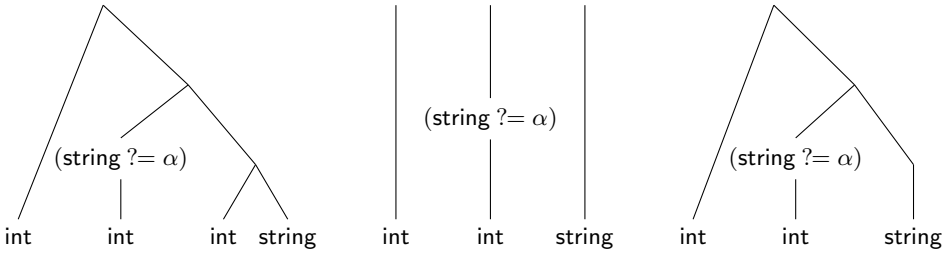
Types can be seen as trees where type tests guard subtrees and union types correspond to branches. Ground types, type variables, type functions or type function applications make up the leaves of a type tree. Individual paths through this tree are valid types themselves. For example: $(\mathsf{int} \mathbin{?=} \alpha) \to (\mathsf{int} \vee \mathsf{string})$ has two paths: $(\mathsf{int} \mathbin{?=} \alpha) \to \mathsf{int}$ and $(\mathsf{int} \mathbin{?=} \alpha) \to \mathsf{string}$ respectively.

We define two functions Unbraid and Braid which respectively split up a type into a set of paths and combine a set of paths back into a type. Their definitions are in fig. 2. Unbraid splits a type into two sets of paths whenever it sees a union type, while type tests are transformed into path tests. As an example of Unbraid and Braid, fig. 3 shows a tree representation of the type $\mathsf{int} \vee ((\mathsf{string} \mathbin{?=} \alpha) \to \mathsf{int}) \vee (\mathsf{int} \vee \mathsf{string})$, the paths resulting from Unbraid, and the type reconstructed with Braid. Note that the reconstructed type has merged the paths ending in $\mathsf{int}$.

$$
\begin{aligned}
\mathrm{Unbraid}(\tau_1 \vee \tau_2) &= \mathrm{Unbraid}(\tau_1) \cup \mathrm{Unbraid}(\tau_2) \\
\mathrm{Unbraid}((\tau_1 \mathbin{?=} \tau_2)^{l_b}_{l_c} \to \tau) &= \{(\tau_1 \mathbin{?=} \tau_2)^{l_b}_{l_c} \to \mathsf{P} \mid \mathsf{P} \in \mathrm{Unbraid}(\tau)\} \\
\mathrm{Unbraid}(\tau) &= \{\tau\}
\end{aligned}
$$

$$
\mathrm{Braid}(\mathsf{P}^*) = \bigvee_{\mathsf{P} \in \mathsf{P}^*} \mathsf{P}
$$

Fig. 2: Definition of Braid and Unbraid




(a) Input type tree (b) After Unbraid (c) After Braid

Fig. 3: Example of using Unbraid and Braid

Besides Braid and Unbraid, we define two more operations on paths:

**Unfold** replaces a type function application at the end of a path with its expansion. If one of the arguments to the type function application is a union

type, the application is performed separately for each part of the union type. As this expansion can introduce a union type, the resulting type is then Unbraided to produce a set of paths.

**Canon** simplifies a path with respect to type tests, much like the check simplification step in section 4.4. First, it removes redundant type tests on the same variable and removes trivially true type tests (of the form ($\mathsf{int} \mathbin{?=} \mathsf{int}$)). Next, if a contradictory type test is found (such as ($\mathsf{int} \mathbin{?=} \mathsf{string}$)), the entire path is replaced with error. Finally, type variables that appear in the scope of a type test are replaced by a concrete type in the rest of the path.

### 1.4 The Solve function

Having defined paths and their operations, we now turn our attention to the Solve algorithm. Recall that the input to Solve is the inferred infinite type and a mapping of type variables to the other infinite function types in the letrec.

1. First, this type is split into a set of finite paths from the top of the tree to every leaf type (which is either a concrete type or a function application). These paths form the initial working set.
2. Every path that ends in a type function application is expanded by replacing the type function application with every other path in turn. Paths are then *canonicalized* to eliminate duplicate or erroneous paths. This process is repeated until expansion no longer yields new paths.
3. In the resulting set of paths, the paths that end in a concrete type are joined together using union types, yielding a final type. If no paths end in a concrete type the function will always diverge, so its return type is just error.

The canonicalization step guarantees termination of the algorithm: it removes type tests that can be statically resolved (ie. ($\mathsf{int} \mathbin{?=} \mathsf{int}$)), it fills in type variables that have been tested for a specific type (ie. $(\mathsf{int} \mathbin{?=} \alpha) \to \alpha$ becomes $(\mathsf{int} \mathbin{?=} \alpha) \to \mathsf{int}$), and it removes erroneous paths such as $(\mathsf{int} \mathbin{?=} \mathsf{string}) \to \ldots$ .

At the start of every iteration the working set is split into three distinct sets:

1. The paths that end in a concrete type (`concrete`): these contribute to the final type; they are passed unmodified to the next iteration.
2. Paths that end in a type function application, but have been visited before (`seen`): these are ignored. Expanding them will not contribute any new paths.
3. Paths that end in a type function application, not visited yet (`todo`): these are Unfolded, passed to Canon, and added to working set for the next invocation of `loop`.

The Solve function stops when this last set becomes empty. At that point, the return type is constructed as explained above using the paths in the `concrete` set. A pseudocode implementation of the Solve algorithm is given in fig. 4. In this code, `T-APPLY` and `T-FUN` are data constructors for types, and `T-FUN-PARAMETERS` extracts the type variables from a function type. The set-up code on lines 3–4

create the empty `visited` set, and the singleton set `paths` to contain a type function application with its own parameters. Line 21 starts the inner loop of the algorithm.

Lines 7–8 split the working set into the three sets described above (`concrete`, `seen` and `todo`). After that, the end conditions are checked as above, and the function either returns a final type (lines 11–13) or continues with another iteration (lines 15–18)

```
1  def Solve(fun, inftypes)
2  visited = ∅
3  parameters = T-FUN-PARAMETERS(fun)
4  paths = { T-APPLY(fun, parameters) }
5
6  def loop(visited, paths)
7  def concrete, applications = partition-set(paths, is-leaf-concrete?)
8  def seen, todo = partition-set(applications, member-of?(visited))
9  if set-empty?(todo)
10 then
11 if set-empty?(concrete)
12 then return T-FUN(parameters, error)
13 else return T-FUN(parameters, Unbraid(concrete))
14 else
15 def new-visited = visited ∪ todo
16 def extra-paths = { Canon(P₂) | P₁ ← todo, P₂ ← Unfold(inftypes, P₁) }
17 def new-paths   = concrete ∪ extra-paths \ {error}
18 return loop(new-visited, new-paths)
19 end
20
21 return loop(visited, paths)
22 end
```

Fig. 4: Pseudocode for the Solve function

Without the `visited` set, the algorithm would loop forever. It would get stuck in expanding the same type function applications over and over. The Canon helper function additionally limits the growth of paths by eliminating paths that contain failing type tests, and removing trivial or duplicated type tests. This ensures that the working set only ever contains paths that actually perform equivalent type tests, albeit in another order.

**Theorem 3.** *Solve always terminates.*

*Proof.* There are three kinds of paths present in the working set of any invocation of `loop`:

1. Paths that end in a concrete type: These cannot be expanded further. If there are no other kinds of paths, the algorithm terminates.

2. Paths that end in an application and have been visited before: These paths have already added their unfolding to the working set. Unfolding these paths again leads to an infinite loop that does not gain any new information. They are therefore not expanded again.
3. Paths that end in an application and have *not* been visited before: These must be expanded and their paths added to the working set.

This last group has the potential to keep the algorithm going. There is an upper bound on the number of possible types in this group, however:

- For a function with $n$ arguments, a path can contain up to $n$ type tests (one per argument) as the `canon` function already removes trivial or duplicate type tests. Conflicting or erroneous type tests will turn the path into error, in which case it cannot belong to the third group. There can thus only be $(g+1)^n$ type tests on the path, where $g$ is the number of ground types that can be tested and the 1 stands for "no test".
- Finally, the type function application at the end of the path must pass $m$ types as arguments. An argument can be either a ground type or a parameter, so there are $(g+n)^m$ different function applications.

There are a finite number of paths, so Solve will terminate after a fixed number of iterations.

We have now shown how to apply blame prediction to programs that contain recursive functions. In the next section we describe how blame prediction can apply to programs with variable assignment.

## 2 Mutation

In the context of program analysis and type systems, mutation is considered to be a prominent member of the "awkward squad". Mutation severely limits program optimizations, as expressions cannot be reordered freely for example. In the context of type systems, mutation is always limited to be *type-preserving*: an assignment may not change the static type of a variable.

Consider the program in Listing 1.1. Running it in an interpreter will print `0 5 7`. Every call to `add!` changes the value of `counter`, which is accessed in every call to `display`.

```
1 (let* ([counter 0]
2        [add! (lambda (x)
3          (let ((new-counter (+ counter x)))
4            (set! counter new-counter)))])
5   (display counter)
6   (add! 5)
7   (display counter)
8   (add! 2)
9   (display counter))
```

Listing 1.1: Example of side effects using set!

In this section we introduce support for mutation in blame prediction in the form of variable assignment. Unlike mutation in statically typed languages, blame prediction allows variable assignments to change types.

This section is subdivided as follows:

- First, we present a new syntax and semantics that adds support for the set! keyword to Scheme$_\beta$ (section 2.1);
- Next, we augment the blame prediction transformation with effect inference (section 2.2);
- The results of this effect inference are used in the check inference, introduction, and mobility stages (section 2.3);
- We show how the augmented blame prediction transformation handles some common idioms in dynamically typed programming languages (section 2.5);
- We prove that the new blame prediction transformation maintains the same safety properties as functional blame prediction (section 2.6).

### 2.1 Syntax and Semantics

In Scheme$_\beta$, as in Scheme, variable assignment is done using the set! keyword. (set! x s) takes a variable x and an ordinary expression s, evaluates the expression and assigns the result to the variable x. fig. 5 shows the syntax for a set! expression.

| | | |
|---|---|---|
| e ∈ Exp | ::= ... | as in the paper |
| | \| (set! x s) | Variable assignment |

Fig. 5: Definition of the set! keyword in Scheme$_\beta$

The semantics of the paper eagerly substited values in let bodies, which effectively made variables immutable. Programs like Listing 1.1 cannot be expressed using these semantics of the paper, therefore we need a new semantics for Scheme$_\beta$ that incorporates variable assignments.

Semantics for languages with mutable variables add an extra indirection step between variables and values: rather than having variables directly contain values (or substituting a value for the variable), they contain pointers to cells in a shared heap. Evaluation of lambda expressions also becomes more complex, as they need to capture the environment in which they are defined which may differ from the environment in which they are called. We present an updated semantics for Scheme$_\beta$ in fig. 6.

Evaluation rules in this semantics are of the form $\mathcal{H}, \mathcal{E}, \mathsf{e} \rightsquigarrow \mathcal{H}', \mathsf{v}$: given a starting heap $\mathcal{H}$ and environment $\mathcal{E}$, the expression e evaluates to v, returning a potentially modified heap $\mathcal{H}'$. We use the syntax $\mathcal{E}[\mathsf{x}]$ or $\mathcal{H}[\ell]$ to denote lookups in environments or heaps. $\mathcal{E}[\mathsf{x} \to \ell]$ or $\mathcal{H}[\ell \to \mathsf{v}]$ return a new environment or heap with an updated binding. Functions are represented by a triple $\langle \mathcal{E}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle$,

$$
\begin{array}{rcll}
\mathsf{v} \in \mathrm{Val} & = & \mathsf{void} \mid \#\mathsf{f} \mid \#\mathsf{t} \mid \mathsf{n} \mid \langle \mathcal{E}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle & \text{Runtime values} \\
\ell \in \mathrm{Loc} & \subset & \mathbb{N} & \\
\mathcal{E} \in \mathrm{Env} & = & \mathsf{x} \rightharpoonup \mathrm{Loc} & \text{Environments} \\
\mathcal{H} \in \mathrm{Heap} & = & \mathrm{Loc} \rightharpoonup \mathrm{Val} & \text{Heaps}
\end{array}
$$

$$\mathcal{H}, \mathcal{E}, \mathsf{x} \rightsquigarrow \mathcal{H}, \mathcal{H}[\mathcal{E}[\mathsf{x}]] \quad \text{(E-VAR)} \qquad \mathcal{H}, \mathcal{E}, \mathsf{c} \rightsquigarrow \mathcal{H}, \mathsf{c} \quad \text{(E-CONST)}$$

$$\mathcal{H}, \mathcal{E}, (\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e}) \rightsquigarrow \mathcal{H}, \langle \mathcal{E}, \mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e} \rangle \quad \text{(E-LAMBDA)}$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{s} \rightsquigarrow \mathcal{H}, \mathsf{v}_c \\ i = 2 \quad \text{if } \mathsf{v}_c = \#\mathsf{f}, \quad 1 \text{ otherwise} \\ \mathcal{H}, \mathcal{E}, \mathsf{e}_i \rightsquigarrow \mathcal{H}', \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{if}\ \mathsf{s}\ \mathsf{e}_1\ \mathsf{e}_2) \rightsquigarrow \mathcal{H}', \mathsf{v}} \quad \text{(E-IF)}$$

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \text{ fresh} \\ \mathcal{E}' = \mathcal{E}[\mathsf{x}_1 \to \ell_1, \ldots, \mathsf{x}_n \to \ell_n] \\ \mathcal{H}, \mathcal{E}', \mathsf{e}_i \rightsquigarrow \mathcal{H}, \mathsf{v}_i \qquad \forall i \in [1 \ldots n] \\ \mathcal{H}_1 = \mathcal{H}[\ell_1 \to \mathsf{v}_1, \ldots, \ell_n \to \mathsf{v}_n] \\ \mathcal{H}_1, \mathcal{E}', \mathsf{e} \rightsquigarrow \mathcal{H}_2, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{letrec}\ ([\mathsf{x}_1\ \mathsf{e}_1] \ldots [\mathsf{x}_n\ \mathsf{e}_n])\ \mathsf{e}) \rightsquigarrow \mathcal{H}_2, \mathsf{v}} \quad \text{(E-LETREC)}$$

$$\frac{\begin{array}{c}\ell \text{ fresh} \\ \mathcal{H}, \mathcal{E}, \mathsf{e}_\mathsf{x} \rightsquigarrow \mathcal{H}_1, \mathsf{v}_\mathsf{x} \\ \mathcal{E}' = \mathcal{E}[\mathsf{x} \to \ell] \qquad \mathcal{H}_2 = \mathcal{H}_1[\ell \to \mathsf{v}_\mathsf{x}] \\ \mathcal{H}_2, \mathcal{E}', \mathsf{e} \rightsquigarrow \mathcal{H}_3, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{let}\ (\mathsf{x}\ \mathsf{e}_\mathsf{x})\ \mathsf{e}) \rightsquigarrow \mathcal{H}_3, \mathsf{v}} \quad \text{(E-LET)}$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{p} \rightsquigarrow \mathcal{H}, \mathsf{v}_\mathsf{p} \\ \text{If } \mathsf{v}_\mathsf{p} = \#\mathsf{f}, \text{raise a } \mathsf{err\text{-}blame(p)} \text{ error.} \\ \mathcal{H}, \mathcal{E}, \mathsf{e} \rightsquigarrow \mathcal{H}_1, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{check}\ \mathsf{p}\ \mathsf{e}) \rightsquigarrow \mathcal{H}_1, \mathsf{v}} \quad \text{(E-CHECK)}$$

$$\frac{\mathcal{H}, \mathcal{E}, \mathsf{s} \rightsquigarrow \mathcal{H}, \mathsf{v} \qquad \mathcal{H}_1 = \mathcal{H}[\mathcal{E}[\mathsf{x}] \to \mathsf{v}]}{\mathcal{H}, \mathcal{E}, (\mathsf{set!}\ \mathsf{x}\ \mathsf{s}) \rightsquigarrow \mathcal{H}_1, \mathsf{void}} \quad \text{(E-SET)}$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{s}_i \rightsquigarrow \mathcal{H}, \mathsf{v}_i \qquad \forall i \in [0 \ldots n] \\ \mathcal{H}_1, \mathcal{E}', \mathsf{e} = \delta(\mathcal{H}, \mathcal{E}, \mathsf{v}_0, \mathsf{v}_1 \ldots \mathsf{v}_n) \\ \mathcal{H}_1, \mathcal{E}', \mathsf{e} \rightsquigarrow \mathcal{H}_2, \mathsf{v}\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n) \rightsquigarrow \mathcal{H}_2, \mathsf{v}} \quad \text{(E-APPLY)}$$

$$
\begin{array}{rcll}
\delta(\mathcal{H}, \mathcal{E}, \mathsf{o}_\#, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathsf{err\text{-}not\text{-}int}(\mathsf{v}_i) & \text{if } \exists i : \neg\mathsf{number?}\ \mathsf{v}_i \\
\delta(\mathcal{H}, \mathcal{E}, \mathsf{o}_\#, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathcal{H}, \mathcal{E}, \mathsf{o}_\#(\mathsf{v}_1, \ldots, \mathsf{v}_m) & \text{otherwise} \\
\delta(\mathcal{H}, \mathcal{E}, \langle \mathcal{E}_\mathsf{f}, \mathsf{x}_1 \ldots \mathsf{x}_m, \mathsf{e} \rangle, \mathsf{v}_1, \ldots, \mathsf{v}_m) & = & \mathcal{H}[\ell_1 \to \mathsf{v}_1, \ldots, \ell_n \to \mathsf{v}_n], \mathcal{E}_\mathsf{f}[\mathsf{x}_1 \to \ell_1, \ldots, \mathsf{x}_n \to \ell_n], \mathsf{e} & \\
\delta(\mathcal{H}, \mathcal{E}, \lambda\mathsf{x}_1 \ldots \mathsf{x}_m.\mathsf{e}, \mathsf{v}_1, \ldots, \mathsf{v}_n) & = & \mathsf{err\text{-}args\text{-}\lambda}(\lambda\mathsf{x}_1 \ldots \mathsf{x}_m.\mathsf{e}) & \text{if } m \neq n \\
\delta(\mathcal{H}, \mathcal{E}, \mathsf{v}, \ldots) & = & \mathsf{err\text{-}not\text{-}\lambda}(\mathsf{v}) & \text{if } \neg\mathsf{function?}(\mathsf{v})
\end{array}
$$

Fig. 6: Semantics for Scheme$_\beta$ with support for mutation

where $\mathcal{E}$ is the environment in which it is defined, $\mathsf{x}_1 \ldots \mathsf{x}_n$ are the names of its arguments, and $\mathsf{e}$ is the body.

We explain some of the rules:

- E-CONST: evaluating a constant simply returns the constant.
- E-VAR: to evaluate a variable, its location is first looked up in the environment, and this location is then looked up in the heap.
- E-LAMBDA: a lambda expression captures the environment in which it is defined.
- E-IF: the correct subexpression is chosen depending on $\mathsf{s}$ being #f or not.
- E-LETREC: every expression $\mathsf{e}_i$ is evaluated in the environment $\mathcal{E}'$ which maps all bound variables to a new location $\ell_i$. The resulting values (all functions, see section 1.1) are assigned to the corresponding locations in heap $\mathcal{H}_1$, which is used to evaluate the body.
- E-LET: after evaluating the expression $\mathsf{e}_\mathsf{x}$, a new memory location $\ell$ is initialized with the value $\mathsf{v}_\mathsf{x}$. The variable $\mathsf{x}$ is bound to this memory location, and the body is evaluated in this new heap and environment.
- E-CHECK: the check expression raises an error if its precondition evaluates to #f.
- E-SET: finally, evaluating set! updates the heap.
- E-APPLY: after evaluating the function $\mathsf{v}_f$ and arguments $\mathsf{v}_1 \ldots \mathsf{v}_n$, the $\delta$ function returns an expression, along with an environment and heap to evaluate it in.

In this semantics we allow a rule to evaluate subexpressions directly, where the previous semantics would rely on evaluation contexts and substitution. If any of these subexpressions raises an error — or in the case of E-VAR, the variable is not bound in the environment — evaluation immediately stops with the raised error.

The extended syntax and semantics are sufficient to describe and evaluate programs that perform side effects. For the sake of simplicity (and presentation) we will also allow begin expressions into the language. A begin expression evaluates its expressions from left to right and returns the value of the last expression. Any begin can be translated to the standard let expression as shown in fig. 7, where `ignore` is a fresh variable name. Functions or let-expressions with multiple expressions in the body are implicitly wrapped in a begin expression.

$$
\begin{aligned}
[\![\texttt{(begin)}]\!] &= \texttt{(void)} \\
[\![\texttt{(begin e)}]\!] &= \texttt{e} \\
[\![\texttt{(begin e es ...)}]\!] &= \texttt{(let (ignore e) } [\![\texttt{(begin es ...)}]\!]\texttt{)}
\end{aligned}
$$

Fig. 7: Translation of **`begin`** to **`let`**

### 2.2 Effect Inference for Scheme$_\beta$

Now that we have explained the syntax and semantics of the extended Scheme$_\beta$, we turn to effect inference. In order to ensure correctness of the transformed program, effect inference is used to estimate which variables are affected by evaluating a given subexpression. The inferred effects will enable check mobility to move checks upwards in the program without changing its meaning. For example, consider the snippet in Listing 1.2, taken right after applying `check` introduction:

```
1 (begin (display (check (string? name) (string-length name)))
2        (f)
3        (check (string? name) (string-append "Hello, " name)))
```

Listing 1.2: Example of why effect inference is needed

This snippet makes use of a variable `name` which is defined above, and a function `f`. There are three possibilities:

1. `f` assigns to the variable `name`. In this case, the second `check` cannot be moved over the call to `f`, as it might have changed the type of `name`.
2. `f` assigns to the variable `name`, but preserves its type (`string`).
3. `f` does not assign to `name`. In both cases, the second `check` can safely be moved over the call to `f` and merged with the first `check`.

With effect inference we can determine which of the two cases applies and correctly move (or not move) the `check` upwards. Figure 8 describes the type and effect inference rules for Scheme$_\beta$. Judgments are of the form $\Gamma \vdash_E \mathtt{e} : \tau \,!\, \bar{\mathtt{x}}$, meaning that — according to environment $\Gamma$ — expression `e` has type $\tau$ and assigns to variables in the set $\bar{\mathtt{x}}$. We use $\star$ to denote all variables that are assigned to in the program, which can be known by traversing the AST. Type-level functions are denoted with $\overset{\bar{\mathtt{x}}}{\Pi}\, \alpha_{1\ldots n}.\tau$, where $\bar{\mathtt{x}}$ is the set of variables mutated by the function, or $\star$ if this is unknown.

The type- and effect inference rules are as follows:

- There are two rules for looking up the types of variables: TE-VAR-MUT assigns variables type $\star$ if they are a member of $\star$. Otherwise, TE-VAR performs type environment lookup as usual.
- TE-CONST is unmodified with respect to the paper.
- TE-IF returns an or-type consisting of the types of the two branches, and its effect is the union of both branches' effects.
- TE-LET again uses `Chain` to chain the types of the expression and body together. The effect is the union of the effects of both subexpressions, but without the bound variable `x`.
- The type returned by TE-LAMBDA is a type-level function with given parameters and body type. This function type is annotated with the effects of its body. The `lambda` expression itself has no effects.

$\boxed{\Gamma \vdash_E \mathsf{e} : \tau \,!\, \overline{\mathsf{x}}}$

$$\frac{\mathsf{x} \in \star}{\Gamma \vdash_E \mathsf{x} : \star \,!\, \emptyset} \quad \text{(TE-VAR-MUT)} \qquad \frac{\Gamma(\mathsf{x}) = \tau}{\Gamma \vdash_E \mathsf{x} : \tau \,!\, \emptyset} \quad \text{(TE-VAR)}$$

$$\frac{}{\Gamma \vdash_E \mathsf{c} : \text{Typeof}(\mathsf{c}) \,!\, \emptyset} \quad \text{(TE-CONST)}$$

$$\frac{\Gamma \vdash_E \mathsf{e}_1 : \tau_1 \,!\, \overline{\mathsf{x}}_1 \qquad \Gamma \vdash_E \mathsf{e}_2 : \tau_2 \,!\, \overline{\mathsf{x}}_2}{\Gamma \vdash_E (\mathsf{if\ s\ e}_1\ \mathsf{e}_2) : \tau_1 \vee \tau_2 \,!\, \overline{\mathsf{x}}_1 \cup \overline{\mathsf{x}}_2} \quad \text{(TE-IF)}$$

$$\frac{\Gamma \vdash_E \mathsf{e}_\mathsf{x} : \tau_1 \,!\, \overline{\mathsf{x}}_1 \qquad \Gamma, \mathsf{x} : \tau_L \vdash_E \mathsf{e} : \tau \,!\, \overline{\mathsf{x}} \qquad \tau_L = \text{Leaves}(\tau_1)}{\Gamma \vdash_E (\mathsf{let}\ (\mathsf{x}\ \mathsf{e}_\mathsf{x})\ \mathsf{e}) : \text{Chain}(\tau_1, \tau) \,!\, \overline{\mathsf{x}}_1 \cup \overline{\mathsf{x}} \setminus \{\mathsf{x}\}} \quad \text{(TE-LET)}$$

$$\frac{\Gamma, \mathsf{x}_1 : \alpha_1, \ldots, \mathsf{x}_n : \alpha_n \vdash_E \mathsf{e} : \tau \,!\, \overline{\mathsf{x}} \qquad \alpha_1, \ldots, \alpha_n \text{ fresh}}{\Gamma \vdash_E (\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e}) : \overset{\overline{\mathsf{x}}}{\Pi}\ \alpha_{1\ldots n}.\tau \,!\, \emptyset} \quad \text{(TE-LAMBDA)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_E \mathsf{s}_i : \tau_i \,!\, \emptyset \quad \forall i \in [0 \ldots n] \\ \ell_f = \text{Label}((\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n))\end{array}}{\Gamma \vdash_E (\mathsf{s}_0\ \mathsf{s}_1 \ldots \mathsf{s}_n) : \text{Apply}(\tau_0, \ell_f, \tau_1 \ldots \tau_n) \,!\, \mathsf{FunEffect}(\tau_0)} \quad \text{(TE-APPLY)}$$

$$\frac{}{\Gamma \vdash_E (\mathsf{set!\ x\ s}) : \mathsf{void} \,!\, \{\mathsf{x}\}} \quad \text{(TE-SET)}$$

Fig. 8: Inference of side effects

- TE-APPLY constructs its return type using the same Apply function as the paper. The effects of a function application entirely depend on the output of the FunEffect function, shown in fig. 9. This function analyzes the type $\tau_0$ of the value being applied and returns an overestimation of the variables being assigned to. Applying a known type function results in the same effect as the function, while applying a type variable results in the $\star$ effect.
- Finally, TE-SET has type void and the intended effect.

$$
\begin{array}{ll}
\mathsf{FunEffect}(\overset{\bar{\mathsf{x}}}{\Pi}\ \alpha_{1\ldots n}.\tau) & = \bar{\mathsf{x}} \\
\mathsf{FunEffect}(\tau_1 \vee \tau_2) & = \mathsf{FunEffect}(\tau_1) \cup \mathsf{FunEffect}(\tau_2) \\
\mathsf{FunEffect}((\tau_t\ ?{=}\ \tau_1)^{\ell_c}_{\ell_b} \to \tau) & = \mathsf{FunEffect}(\tau) \\
\mathsf{FunEffect}(\alpha) & = \star \\
\mathsf{FunEffect}(\tau) & = \emptyset
\end{array}
$$

Fig. 9: Auxiliary functions for type- and effect inference for Scheme$_\beta$

The type $\star$ is inspired from gradual typing [3], where it means "any type". In this work types are only used to statically eliminate type tests however, so primitive operations involving variables in $\star$ will always perform a type test. The idea is that after check mobility and simplification, most of the type tests on such variables are merged.

It should be possible to improve the accuracy of type inference for mutated variables. Such improvements come at the cost of increased complexity and transformation time. The difference would be most noticeable in code that only ever assigns values of a single type to variables, as shown in Listing 1.3. It defines a simple `counter` function that returns a higher number every time it is called. Because `count` is assigned to on line 4, it has type $\star$ and therefore the `counter` function also has type $\star$. All uses of the function will require a type test even though `counter` will never return anything but a number.

```
1 (define count 1)
2
3 (define (inc)
4   (set! count (+ count 1))
5   count)
```

Listing 1.3: Example of type-preserving mutation

check introduction proceeds as in the paper: checks are inserted around function application expressions, according to their type. The inferred effects will be used in the next step, namely to restrict check mobility.

### 2.3 Blame Prediction with Mutation

After having defined the semantics of a mutation-enabled Scheme$_\beta$ and shown how to perform type- and effect inference, we now turn to check mobility. As

mentioned in the paper, it is very important that check mobility does not change the semantics of the program. There were two invariants:

1. checks on variables may not escape the let expressions that bind them, and
2. checks may not escape lambda expressions.

We now add a third, namely

3. checks on mutable variables may not skip over expressions that potentially assign to them.

To illustrate this third point, consider the code in Listing 1.4.

```
1 (define (foo x)                          ; type of x is
    unknown
2   [number? x] (set! x (+ x 1))           ; type of x becomes
    number
3   (set! x (to-string x)                  ; type of x becomes
    string
4   [string? x] (string-append "Hello, " x))
```

Listing 1.4: Restrictions on check mobility

The string? test on line 4 may not skip over the set! on line 3, as the type of x is changed there. Before line 3, the type of x will be number.

We present an updated set of rules for check mobility in fig. 10. Most of the rules are the same as in the paper, but the new and modified rules require some explanation:

- The FE-SET rule needs to process a potential lambda expression passed as argument.
- FE-LET is similar to the rule in the paper: only the checks in the body that do not involve x are propagated. However, instead of using the mask function, it now uses the Push function. This function takes a list of variables that cannot be used in propagated checks and a precondition; it returns a precondition that can be propagated upwards ($\mathsf{p}^{\uparrow}$) and a precondition that must remain ($\mathsf{p}^{\downarrow}$). Applying this to the let expression shows that $\mathsf{p}^{\downarrow}$ consists of preconditions which mention the bound variable x or any of the mutated variables $\overline{\mathsf{x}}$, while $\mathsf{p}^{\uparrow}$ consists of the precondition which do not.

### 2.4 Check Simplification with Mutation

The check simplification rules proposed in the paper still apply, with one important caveat: check–check simplification on a variable in the $\overline{\mathsf{x}}$ set may not jump over an expression that possibly modifies the variable. Consider the program in Listing 1.5. It first uses the variable x in an addition, then sets it to the result of applying f to x and 5, and finally tries to use it as a number again. In this case the function f is unknown, so it might return a number or it might not. In the second case, eliminating the check on line 4 would result in a wrong prediction.

$$\frac{\mathsf{e} \to \mathsf{e}' \uparrow \mathsf{p}}{\mathsf{e} \to_P [\mathsf{p}]\mathsf{e}'} \quad (\text{FE-PROGRAM}) \qquad \mathsf{c} \to \mathsf{c} \uparrow \#\mathsf{t} \quad (\text{FE-CONST})$$

$$\frac{\mathsf{s} \to \mathsf{s}' \uparrow \#\mathsf{t}}{(\mathsf{set!}\ \mathsf{x}\ \mathsf{s}) \to (\mathsf{set!}\ \mathsf{x}\ \mathsf{s}') \uparrow \#\mathsf{t}} \quad (\text{FE-SET}) \qquad \mathsf{x} \to \mathsf{x} \uparrow \#\mathsf{t} \quad (\text{FE-VAR})$$

$$\frac{\mathsf{s}_i \to \mathsf{s}_i' \uparrow \#\mathsf{t} \qquad \forall i \in 1 \ldots n}{[\mathsf{p}](\mathsf{s}_1 \ldots \mathsf{s}_n) \to (\mathsf{s}_1' \ldots \mathsf{s}_n') \uparrow \mathsf{p}} \quad (\text{FE-APPLY})$$

$$\frac{\mathsf{s} \to \mathsf{s}' \uparrow \#\mathsf{t} \qquad \mathsf{e}_1 \to \mathsf{e}_1' \uparrow \mathsf{p}_1 \qquad \mathsf{e}_2 \to \mathsf{e}_2' \uparrow \mathsf{p}_2}{(\mathsf{if}\ \mathsf{s}\ \mathsf{e}_1\ \mathsf{e}_2) \to (\mathsf{if}\ \mathsf{s}\ [\mathsf{p}_1]\mathsf{e}_1'\ [\mathsf{p}_2]\mathsf{e}_2') \uparrow \mathsf{p}_1 \vee \mathsf{p}_2} \quad (\text{FE-IF})$$

$$\frac{\mathsf{e}_\mathsf{x} \to \mathsf{e}_\mathsf{x}' \uparrow \mathsf{p}_\mathsf{x} \qquad \mathsf{e} \to \mathsf{e}' \uparrow \mathsf{p} \qquad \overline{\mathsf{x}} = \mathsf{Effects}(\mathsf{e}_\mathsf{x}) \qquad \langle \mathsf{p}^\uparrow, \mathsf{p}^\downarrow \rangle = \mathsf{Push}(\mathsf{x} \cup \overline{\mathsf{x}}, \mathsf{p})}{(\mathsf{let}\ (\mathsf{x}\ \mathsf{e}_\mathsf{x})\ \mathsf{e}) \to (\mathsf{let}\ (\mathsf{x}\ \mathsf{e}_\mathsf{x}')\ [\mathsf{p}^\downarrow]\mathsf{e}') \uparrow \mathsf{p}_\mathsf{x} \wedge \mathsf{p}^\uparrow} \quad (\text{FE-LET})$$

$$\frac{\mathsf{e} \to \mathsf{e}' \uparrow \mathsf{p}}{(\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ \mathsf{e}) \to (\mathsf{lambda}\ (\mathsf{x}_1 \ldots \mathsf{x}_n)\ [\mathsf{p}]\mathsf{e}') \uparrow \#\mathsf{t}} \quad (\text{FE-LAMBDA})$$

Fig. 10: Check mobility in the presence of side effects

$$\begin{array}{lll}
\mathsf{Push}(\overline{\mathsf{x}}, (\tau?\ \mathsf{c})^{\ell_b}) & = \langle (\tau?\ \mathsf{c})^{\ell_b}, (\tau?\ \mathsf{c})^{\ell_b} \rangle & \\
\mathsf{Push}(\overline{\mathsf{x}}, (\tau?\ \mathsf{x})^{\ell_b}) & = \langle (\tau?\ \mathsf{x})^{\ell_b}, (\tau?\ \mathsf{x})^{\ell_b} \rangle & \text{if } \mathsf{x} \notin \overline{\mathsf{x}} \\
 & = \langle \#\mathsf{t}, (\tau?\ \mathsf{x})^{\ell_b} \rangle & \text{if } \mathsf{x} \in \overline{\mathsf{x}} \\
\mathsf{Push}(\overline{\mathsf{x}}, \mathsf{p}_1 \wedge \mathsf{p}_2) & = \langle \mathsf{p}_1^\uparrow \wedge \mathsf{p}_2^\uparrow, \mathsf{p}_1^\downarrow \wedge \mathsf{p}_2^\downarrow \rangle & \\
\mathsf{Push}(\overline{\mathsf{x}}, \mathsf{p}_1 \vee \mathsf{p}_2) & = \langle \mathsf{p}_1^\uparrow \vee \mathsf{p}_2^\uparrow, \mathsf{p}_1^\downarrow \vee \mathsf{p}_2^\downarrow \rangle & \\
\multicolumn{3}{l}{\quad \text{where } \langle \mathsf{p}_1^\uparrow, \mathsf{p}_1^\downarrow \rangle = \mathsf{Push}(\overline{\mathsf{x}}, \mathsf{p}_1), \langle \mathsf{p}_2^\uparrow, \mathsf{p}_2^\downarrow \rangle = \mathsf{Push}(\overline{\mathsf{x}}, \mathsf{p}_2)}
\end{array}$$

Fig. 11: Auxiliary functions for check mobility

```
1  (check (number? x)
2    (let (y (+ x 1))
3      (set! x (f x 5))
4      (check (number? x)
5        (+ x 1))))
```

Listing 1.5: Example where check-check simplification is not appropriate

### 2.5 Mutation Patterns

Having shown how blame prediction needs to be adapted to deal with mutation, in this section we present how blame prediction applies to three commonly used idioms involving mutation.

*Idiom 1: Caching* Consider the code in Listing 1.6. The goal of this code is to cache the result of an expensive computation. The variable `cached-value` holds a cached value[1], while the variable `cached?` is used to remember whether the expensive computation has taken place.

```
1  (define cached-value #f)
2  (define cached?      #f)
3
4  (define (get-or-compute)
5    (if cached?
6      cached-value
7      (begin (set! cached? #t)
8             (set! cached-value (expensive-computation))
9             cached-value)))
```

Listing 1.6: Caching expensive computations

At the start of the program, both `cached?` and `cached-value` contain a value of type `boolean`. After invoking `get-or-compute` once, `cached-value` will contain the value computed by `expensive-computation`. According to the type inference above, however, `cached-value` has type $\star$ rather than the specific type computed by `expensive-computation`.

*Idiom 2: Wrong initial types* Listing 1.7 is taken from the `fft` program in the Gabriel benchmarks [4]. The program calculates the Fast Fourier Transform (FFT) of two given vectors `areal` and `aimag`. The objective of the benchmark is to measure the speed of arithmetic computation, so in order to minimize creation of garbage, all variables are defined up front in a big `let` expression and later overwritten using `set!` to intermediate or final values. However, this causes the type to change; for example the `set!` expressions on lines 6 and 7 change the types of the values in `ar` and `ai` from `number` to `vector`.

[1] This caching mechanism could also be implemented using a combination of **`delay`** and **`force`**, two Scheme primitives for delayed evaluation. We avoid this method because few other languages support these primitives.

```
1  (define (fft areal aimag)
2    (let ((ar 0) (ai 0)
3          (i  1) (j  1)
4          (k  0) (n  0)
5          (tr 0) (ti 0))
6      (set! ar areal)
7      (set! ai aimag)
8      (set! n (- (vector-length ar) 1))
9      ...
10     (let l3 ()
11       (cond ((< i j)
12              (set! tr (vector-ref ar j))
13              (set! ti (vector-ref ai j))
14              (vector-set! ar j (vector-ref ar i))
15              (vector-set! ai j (vector-ref ai i))
16              (vector-set! ar i tr)
17              (vector-set! ai i ti)))
18       ...
19       (set! j (+ j k))
20       (set! i (+ i 1))
21       (if (< i n) (l3) 'done))
22     ...
23     #t))
```

Listing 1.7: Example of wrongly typed initial values

Let us consider what this means for blame prediction. The `vector-ref` and `vector-set!` operations on lines 12–17 require `ar` and `ai` to be of type vector. These checks will move upwards to the beginning of the function at line 12. They cannot be eliminated further because of the condition on line 11. However, `ar` and `ai` are declared as numbers on line 2, so an implementation of blame prediction that is not aware of type-changing assignments will predict an error at the top of the let, even though there is none.

*Idiom 3: Type changes at a distance* Listing 1.8 defines a common helper function to perform an arbitrary operation on a file and write back the results. The operation to be performed is defined in the `process` function, which is passed in by the programmer. The `process` function should return a string, otherwise the `write-file` built-in will raise a type error. The implementation of blame prediction as described in the paper will generate a check for (string? `input`) around `write-file` and immediately float it up below the `read-file`. At run-time, this check will always succeed, as `read-file` indeed returns a string. Instead, the check should remain right before the call to `write-file`, as `process` might return something other than a string.

```
1  (define (with-file-do filename process)
2    (let ((input (read-file filename)))
3      (set! input (process input))
4      (write-file filename input)))
```

Listing 1.8: Example of changes at a distance

### 2.6 Proof of Safety

In this section we show that the extensions made to the blame prediction transformation do not affect program correctness.

**Correctness of Effect Inference** Central to this proof is the effect inference from section 2.2: for any given expression $\mathsf{e}$, it must correctly identify the variables $\overline{\mathsf{x}}$ it modifies. We start by proving that the type and effect inference *over-approximates* this set of variables, ie. that the expression *might* assign to any of the variables in $\overline{\mathsf{x}}$, but *may not* assign to any variables outside of $\overline{\mathsf{x}}$.

As a simple example, consider the snippet below:

```
1 (if (even? x)
2   (set! even (+ even 1))
3   (set! odd (+ odd 1)))
```

It modifies either `even` or `odd`, but which variable exactly depends on the value of `x`. Therefore we over-approximate the set of variables it modifies as $\{\mathsf{even}, \mathsf{odd}\}$.

**Lemma 1.** *Let* $\mathsf{e}$ *be a* Scheme$_\beta$ *expression; then the judgment* $\Gamma \vdash \mathsf{e} : \tau \,!\, \overline{\mathsf{x}}$ *over-approximates the set of variables mutated by* $\mathsf{e}$ *as* $\overline{\mathsf{x}}$*. We define "over-approximation" as follows: for all sets of mutated variables* $\overline{\mathsf{x}_o}$ *observed from program execution,* $\overline{\mathsf{x}_o} \subseteq \overline{\mathsf{x}}$*.*

*Proof.* By case analysis on the type- and effect inference rules:

- TE-VAR, TE-VAR-MUT and TE-CONST analyze variable references and constant expressions, which can have no side effects.
- TE-IF over-approximates the set of mutated variables by taking the union of the mutated variables of both branches.
- TE-LET similarly over-approximates by taking the union of variables mutated both in the expression and the body.
- TE-LAMBDA correctly states that a lambda expression by itself does not mutate any variables. The type returned contains an over-approximation of all variables set in the body, which is used later.
- TE-APPLY asserts that both the function expression and its arguments cannot mutate any variables, as they are simple expressions. The only variables mutated by a function application are those as returned by the FunEffect helper function. For the most part it simply traverses the given type, except for the following kinds of types
  - Function types, as stated above, contain an over-approximation of the types modified by their body; they are simply returned.
  - In a function application, function types could be bound to *any* possible function. FunEffect over-approximates by assuming all variables are mutated ($\star$). This is an over-approximation because it also includes variables local to the current function, which cannot possibly be affected by external functions.

  - Applying other concrete type will result in an error, therefore no variables can be set.
- TE-SET states that a set! expression mutates its variable, which is exactly the intended effect.

### Program Properties after Blame Prediction

**Lemma 2.** *The check mobility and simplification stages preserve the following properties:*

1. check *expressions never refer to unbound variables;*
2. check *expressions verify the same values as the preconditions they guard.*

We begin the proof of lemma 2 with check mobility.

*Proof.* By induction over the mobility rules.

- FE-PROGRAM simply captures any preconditions that escape the program and reinserts them.
- FE-CONST, FE-SET and FE-VAR do not propagate up any preconditions.
- FE-APPLY simply propagates any preconditions attached to it. This is the base case, as any preconditions start from the function application expressions.
- FE-IF lifts the preconditions $\mathsf{p}_1$ and $\mathsf{p}_2$ out of its branches. Property 1 is preserved because an if expression does not introduce new bindings, and property 2 is preserved because the condition s cannot contain any set! expressions.
- FE-LET is the most important rule in regards to the two properties. Regarding property 1, it must ensure that the preconditions to be propagated do not mention x, as that would introduce a reference to an unbound variable. Regarding property 2, preconditions on variables mutated by the expression $\mathsf{e}_\mathsf{x}$ may not be lifted out of the body. Both are satisfied by the Push function: it separates p into $\mathsf{p}^\uparrow$ and $\mathsf{p}^\downarrow$, where $\mathsf{p}^\uparrow$ is guaranteed not to contain any references to x or any of the variables $\overline{\mathsf{x}}$ mutated by $\mathsf{e}_\mathsf{x}$. Moreover, lemma 1 states that $\mathsf{e}_\mathsf{x}$ mutates *at most* the variables returned by the Effects function.
- FE-LAMBDA, as before, prevents any preconditions from floating out of the lambda expression.

*Proof.* For the check simplification stage, we simply enumerate the simplifications:

- **Or–true simplification** removes preconditions of the form $\#\mathsf{t} \vee \mathsf{p}$, where p would never be evaluated.
- **And–check simplification** merges multiple identical preconditions on the same variable in an aggregate precondition. From the previous stage we know that these preconditions would not reference unbound variables or check values different from the blame label, so this is fine.

– **Check–literal simplification** removes checks on literals when they can statically be proven true. It does not touch preconditions on variables.
– **Check–check simplification**, finally, has the potential to violate the second property. The first property cannot be altered: we remove an innermost check expression which must already satisfy the first property, and we merge it with the outermost check expression, which must also already satisfy it. For expressions of the form (check ($\tau$? x) (CONTEXT (check ($\tau$? x) ...))), the innermost check may only be merged with the outermost check if CONTEXT is known not to modify x (ie. where x remains the same throughout CONTEXT). By relying on the over-approximation of lemma 1, we only perform this simplification when this is the case.

**Traces and Trace Semantics** Before we can prove the three properties described in section 5, we need to extend traces to record the changing types of variables. To obtain traces, we execute the program in a modified version of the semantics presented in section 2.1. The modified semantics build up a trace while evaluating the program. Figure 12 shows the most important modified rules, the rest simply thread the trace through the evaluation.

$$
\begin{array}{lll}
\mathsf{T} & ::= & \mathsf{s} \cdot \mathsf{T} \mid \epsilon \mid \mathsf{err\text{-}blame(p)} \mid \mathsf{err\text{-}use(p)} \\
\mathsf{s} & ::= & \mathsf{use(p)} \qquad\qquad \text{use step, as before} \\
& \mid & \mathsf{check(p)} \qquad\qquad \text{check step, as before} \\
& \mid & \ell \leftarrow \tau \qquad\qquad \text{Update memory at } \ell \text{ to type } \tau
\end{array}
$$

$$
\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{e_x}, \mathsf{T} \rightsquigarrow \mathcal{H}_1, \mathsf{v_x}, \mathsf{T}_1 \\ \ell \text{ fresh} \qquad \mathcal{E}' = \mathcal{E}[\mathsf{x} \to \ell] \qquad \mathcal{H}_2 = \mathcal{H}_1[\ell \to \mathsf{v_x}] \\ \tau = \mathsf{typeof(v_x)} \qquad \mathsf{T}_2 = \mathsf{T}_1 \cdot (\ell \leftarrow \tau) \text{ if } \mathsf{x} \in \star, \quad \text{otherwise } \mathsf{T}_2 = \mathsf{T}_1 \\ \mathcal{H}_2, \mathcal{E}', \mathsf{e}, \mathsf{T}_2 \rightsquigarrow \mathcal{H}_3, \mathsf{v}, \mathsf{T}_3\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{let\ (x\ e_x)\ e}), \mathsf{T} \rightsquigarrow \mathcal{H}_3, \mathsf{v}, \mathsf{T}_3} \quad \text{(ET-LET)}
$$

$$
\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{p}, \mathsf{T} \rightsquigarrow \mathcal{H}, \mathsf{v_p}, \mathsf{T} \\ \text{If } \mathsf{v_p} = \#\mathsf{f}, \text{raise a } \mathsf{err\text{-}blame(p)} \text{ error.} \\ \mathcal{H}, \mathcal{E}, \mathsf{e}, \mathsf{T} \cdot \mathsf{check}(\mathcal{E}[\mathsf{p}]) \rightsquigarrow \mathcal{H}_1, \mathsf{v}, \mathsf{T}_1\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{check\ p\ e}), \mathsf{T} \rightsquigarrow \mathcal{H}_1, \mathsf{v}, \mathsf{T}_1} \quad \text{(ET-CHECK)}
$$

$$
\frac{\begin{array}{c}\mathcal{H}, \mathcal{E}, \mathsf{s}, \mathsf{T} \rightsquigarrow \mathcal{H}, \mathsf{v}, \mathsf{T} \qquad \mathcal{H}_1 = \mathcal{H}[\mathcal{E}[\mathsf{x}] \to \mathsf{v}] \\ \tau = \mathsf{Typeof(v)} \qquad \mathsf{T}_1 = \mathsf{T} \cdot (\mathcal{E}[\mathsf{x}] \leftarrow \tau)\end{array}}{\mathcal{H}, \mathcal{E}, (\mathsf{set!\ x\ s}), \mathsf{T} \rightsquigarrow \mathcal{H}_1, \mathsf{void}, \mathsf{T}_1} \quad \text{(ET-SET)}
$$

Fig. 12: Modified semantics for generating traces

In these tracing semantics, rules take the form $\mathcal{H}, \mathcal{E}, \mathsf{e}, \mathsf{T} \rightsquigarrow \mathcal{H}', \mathsf{v}, \mathsf{T}'$, meaning that evaluating expression e with heap $\mathcal{H}$ and environment $\mathcal{E}$ returns a value v and new heap $\mathcal{H}'$, while trace T is extended to $\mathsf{T}'$. Traces can consist of a

$\mathsf{use}(\mathsf{p})$ or $\mathsf{check}(\mathsf{p})$ step from the paper, as well as a "set type" step $\ell \leftarrow \tau$. This step indicates that a memory location belonging to a mutable variable has been set, potentially changing its type to $\tau$. We track memory locations rather than variable names, as a recursive function invocation can cause multiple copies of the same variable name to appear in a trace. Appending a step $\mathsf{s}$ to a trace $\mathsf{T}$ is denoted as $\mathsf{T} \cdot \mathsf{s}$.

The modified rules are as follows:

- ET-LET evaluates the expression $\mathsf{e_x}$ and updates the heap as before. If the variable being bound is a member of $\star$, the trace is amended to note that the newly allocated memory location now contains a value of type $\tau$. If the variable being bound is never assigned to, it is not recorded in the trace.
- ET-CHECK records the fact that the precondition `p` has been tested. We use the syntax $\mathcal{E}[\mathsf{p}]$ to denote "replace all variable references in `p` with their location in the heap".
- ET-SET updates the trace to note that the variable `x` has been updated.

To illustrate the tracing semantics, let us examine the trace produced by the program Listing 1.9.

```
1  (let* ([counter 0]
2         [add! (lambda (x)
3           (let ((new-counter (+ counter x)))
4             (set! counter new-counter)))])
5    (display counter)
6    (add! 5)
7    (display counter)
8    (add! 2)
9    (display counter))
```

Listing 1.9: Example program for tracing semantics

The trace is then:

$$\mathsf{T} = [\underbrace{\ell_c \leftarrow \mathsf{int}}_{\text{line 1}}, \underbrace{\mathsf{check}(\mathsf{number?}\ \ell_c), \ell_c \leftarrow \mathsf{int}}_{\text{line 6}}, \underbrace{\mathsf{check}(\mathsf{number?}\ \ell_c), \ell_c \leftarrow \mathsf{int}}_{\text{line 8}}]$$

The first part of the trace shows how the `let` expression assigns type `int` to the memory location $\ell_c$, which it allocated for the variable `counter`. The memory location of `add!` is not recorded, as it is not a member of $\star$. The second part of the trace is the invocation of `add!` on line 6, which adds 1 to `counter` (the `check`), then stores the result back (the $\leftarrow$ part). The third part of the trace is identical to the second part.

This trace exhibits two important properties:

1. First, variables (or rather the memory locations they point to) are always initialized with a type. It is an error to refer to a variable before it is bound, similar to the first lemma of the paper.

2. Second, every check refers to the type of the value that was *last assigned* to a particular memory location. We could therefore rearrange checks in a trace, as long as they do not skip over an assignment of the variable(s) they inspect.

**Proof of equivalences** Having defined traces for Scheme$_\beta$ with mutation, we now turn to proving the three equivalences from section 5. They are reproduced below for easy reference.

**Value Preservation**
Iff P runs to completion and produces a value $v$, P′ produces the same value $v$.
Formally: P $\rightsquigarrow v \Leftrightarrow$ P′ $\rightsquigarrow v$.
**Use–Blame Equivalence**
If P raises an error, the blame predicted program P′ must predict blame.
Formally: P $\rightsquigarrow$ err-$\omega \Rightarrow$ P′ $\rightsquigarrow$ err-blame(p).
**Blame–Use Equivalence**
If P′ predicts blame, the original program P must raise an error.
Formally: P′ $\rightsquigarrow$ err-blame(p) $\Rightarrow$ P $\rightsquigarrow$ err-$\omega$.

**Lemma 3.** *Check introduction guards all type tests that could fail in primitive operations with a* check *expression.*

*Proof.* The proof for this is analogous to a proof in the paper. Mutable variables always have type $\star$, as per rule TE-VAR-MUT. Therefore, any use of these variables will generate a check expression, trivially satisfying the property.

*Proof of Value Preservation*

*Proof.* We prove each direction separately.
$\Rightarrow$: Given that P $\rightsquigarrow v$, prove that P′ $\rightsquigarrow v$.
Analogously to the proof in the paper, we start by comparing the trace T of P with T′ from the blame predicted program P′. T returned a value, so every use(p) succeeded. check introduction will insert a check expression to guard every primitive operation that might fail (see previous lemma). Further, the mobility and simplification stages will move these check expressions up, as long as 1) no unbound variables are referenced, and 2) the preconditions verify the same values as the use expressions. For every check in T′, there is a use in T′ (and thus T) that requires the same predicate to be true. Every check and use expression therefore succeeds, thus T must not raise an error and produce a value.

$\Leftarrow$: Given that P′ $\rightsquigarrow v$, prove that P $\rightsquigarrow v$.
The trace T′ of program P′ is equivalent modulo check to the trace T of program P. In other words, removing the checks from T′ yields T. T′ succeeds, so T must also succeed, and thus program P must produce the same value as program P′.

*Proof of Use-Blame Equivalence*

*Proof.* Evaluation of the program P raises an error on a $\mathsf{use}(\mathsf{p})$. We discriminate on the form of $\mathsf{p}$:

1. $\mathsf{p} = (\tau?\ \mathsf{c})$: a check expression would be generated and propagated as high up as possible, the proof is analogous to that of the paper.
2. $\mathsf{p} = (\tau?\ \mathsf{x})$, with $\mathsf{x} \notin \star$: the check experssion for this precondition is propagated as high as possible, up to the binding of $\mathsf{x}$ or the nearest enclosing function. The proof is also analogous to that of the paper.
3. $\mathsf{p} = (\tau?\ \mathsf{x})$, with $\mathsf{x} \in \star$: in this case, the check expression is propagated up to the most recent assignment of $\mathsf{x}$. Lemma 2 ensures that both the check and the use expressions verify the type of the same value. Therefore, at the very latest the check expression will predict blame for the use expression.

*Proof of Blame-Use Equivalence*

*Proof.* $\mathrm{P}'$ evaluates to an $\mathsf{err\text{-}blame}(\mathsf{p})$ error, which must be raised by a $\mathsf{check}(\mathsf{p}_k)$ expression in $\mathrm{P}'$. The proof is analogous to that in the paper, the only difference is the presence of $\ell \leftarrow \tau$ steps in the trace, but lemma 2 guarantees that these do not affect the values tested by check expressions.

## 3 Conclusion

In this technical report we developed support for recurstion and mutation in Scheme$_\beta$.

For recursion the key problem is the infinite types produced by the check inference phase: functions may refer to themselves with arbitrary conditional types beforehand. Our solution consisted of splitting every infinite type into *paths* and expanding paths until a fixpoint is reached. This fixpoint then contains both paths that end in a concrete return type, and paths that end in an (already visited) recursive invocation. The paths with a concrete return type are reassembled into finite types.

For mutation, we started by defining a modified semantics of Scheme$_\beta$ that allowed for variable assignment. This semantics featured an explicit environment that mapped variables to heap locations, and a mutable heap that was threaded through the evaluation. Next, we defined type- and effect inference for Scheme$_\beta$. This allowed us to approximate the variables changed by every expression in the program. Finally, we restricted the check mobility and simplification phases to make sure check expressions are only simplified or moved upwards in the program if the contents of the variable they examine has not changed in the meantime.

## References

1. Peyton Jones, S.: The Implementation of Functional Programming Languages. Prentice-Hall, Inc. (May 1987)

2. Waddell, O., Sarkar, D., Dybvig, R.K.: Fixing Letrec: A Faithful Yet Efficient Implementation of Scheme's Recursive Binding Construct. Higher-Order and Symbolic Computation **18**(3-4) (December 2005)
3. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: SFP '06: Proceedings of the 2006 Workshop on Scheme and Functional Programming. (2006) 81–92
4. Gabriel, R.P., Masinter, L.M.: Performance of Lisp systems. In: LFP '82: Proceedings of the 1982 ACM symposium on LISP and Functional Programming. (1982) 123–142