Big Types in Little Runtime

Open-World Soundness and Collaborative Blame for Gradual Type Systems

Michael M. Vitousek Cameron Swords Jeremy G. Siek

Indiana University, USA

{mvitouse,cswords,jsiek}@indiana.edu

Abstract

Gradual typing combines static and dynamic typing in the same language, offering programmers the error detection and strong guarantees of static types and the rapid prototyping and flexible programming idioms of dynamic types. Many gradually typed languages are implemented by translation into an untyped target language (e.g., Typed Clojure, TypeScript, Gradualtalk, and Reticulated Python). For such languages, it is desirable to support arbitrary interaction between translated code and legacy code in the untyped language while maintaining the type soundness of the translated code. In this paper we formalize this goal in the form of the *open-world soundness* criterion. We discuss why it is challenging to achieve open-world soundness using the traditional proxy-based approach for higher-order casts. We then show how an alternative, the transient design, satisfies open-world soundness by presenting a formal semantics for the transient design and proving that the semantics satisfies open-world soundness. In this paper we also solve a challenging problem for the transient design: how to provide *blame tracking* without proxies. We define a semantics for blame and prove the Blame Theorem. We also prove that the Gradual Guarantee holds for this system, ensuring that programs can be evolved freely between static and dynamic typing. Finally, we demonstrate that the runtime overhead of the transient approach is low in the context of Reticulated Python, an implementation of gradual typing for Python.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords gradual typing, type systems, semantics, interoperability

1. Introduction

Gradual typing [33, 44] enables the safe interaction of statically typed and dynamically typed code. In recent years, gradual typing has been of interest to the research community [2, 3, 28, 30, 37, 39, 41] and to industry developers: numerous new languages have arrived on the scene with elements of gradual typing, including Dart [20], Hack [16], and TypeScript [27].

Gradually typed languages use the dynamic type $\star$ and a *consistency relation* on types to govern how statically typed and untyped code interacts: the consistency relation plays the role that type equality usually does in the type system. Types are consistent if they are equal up to the presence of $\star$.

Most existing gradually typed languages operate by translating a surface language program into an underlying target language, which is then executed. For many gradually-typed systems such as Typed Racket and TypeScript, the target language is a dynamically typed programming language, and gradually-typed programs are expected to seamlessly interact with legacy code in the dynamic language that has not been translated. In this paper, we present a formal treatment of this property, which we refer to as open-world soundness.

Sound gradual typing requires runtime type checks. While most dynamically typed target languages are "sound" in the sense that they will not reach stuck configurations or elicit undefined behavior such as memory corruption, a translated gradually typed language may be *unsound with respect to its type annotations* if values of the wrong type can inhabit variables in statically typed regions of code. This can cause unexpected and difficult-to-debug uncaught type errors. Runtime checks can prevent this, and the traditional way to implement them is to translate the source program into a target language with explicit casts [33, 44].

Not all gradually typed languages are sound with respect to their static types. Languages like TypeScript and Hack do not enforce types at runtime when statically typed and dynamically typed code interact. In these languages, the translation can simply erase the types and no responsibility is placed on target-language programs to supply the translated code with values of the correct type at runtime. This design allows translated programs to straightforwardly interact with target-language programs. However, interoperability is more challenging to achieve for sound gradually typed languages, where runtime casts are inserted to ensure that values flowing into statically typed regions of code satisfy their expected static types. Soundness in these languages is usually shown in a closed world, where the only programs considered are ones that originate in the surface language and are translated and then executed. Open-world soundness extends soundness to translated programs that are embedded in arbitrary target-language code.

We discuss open-world soundness in the context of different designs for runtime type enforcement, demonstrating that the traditional guarded approach to gradual typing fails open-world soundness when the target language is a spartan host (i.e., a language which lacks native runtime support for gradual typing) and that the transient design proposed by Vitousek et al. [48] satisfies it.

Moreover, we explore open-world soundness in the context of the *gradual guarantee* [36], which states that weakening or removing type annotations from a program does not introduce new errors. Languages which support the gradual guarantee allow programmers to evolve their code from untyped scripts to precisely typed programs, and languages that combine open-world soundness and the gradual guarantee let programmers transition projects from dynamic to static at any granularity without needing to change the untyped sections of their code.

In this paper, we identify the open-world soundness property and prove, for the first time, that it holds for a calculus. However, we hypothesize that open-world soundness holds for some existing gradually-typed languages, including Typed Racket [41, 45], TS* [39] and StrongScript [31]. These designs support some degree of open interaction with dynamically typed code in the target language without error. Unfortunately, many of these systems achieve this interaction by significantly limiting which implicit type conversions are allowed, violating the gradual guaranteeor example, a TS* function of the type $\star \to \star$ cannot be passed into a context that expects a bool $\to$ bool function, and therefore the gradual guarantee does not hold.

Typed Racket is aided by Racket's built-in contract support. When functions are exported from Typed Racket into untyped code, they are wrapped with a contract monitor that ensures that interactions between typed and untyped code are sound. Racket is therefore not a spartan host; its features enable Typed Racket to have open-world soundness with a guarded approach.

Contributions. In this work, we explore the design of open-world gradual typing with the gradual guarantee as a guiding principle. Our contributions are:

- We identify and formalize open-world soundness as an important property for integrating gradual typing into existing languages (Section 2.1).
- We demonstrate the difficulties of supporting open-world soundness with the guarded design when translating to a language with limited support for proxies, and show how the transient approach recovers it (Section 2.2).
- We discuss the challenge of defining blame tracking for transient casts and present a solution (Section 3).
- We define the first formal semantics for the transient design as a variant of the gradually typed lambda calculus $\lambda^{\star}_{\to}$, including the new blame tracking strategy (Section 4).
- We prove that $\lambda^{\star}_{\to}$ satisfies the Blame Theorem (Section 5.1), Open-World Soundness (Section 5.2), and the Gradual Guarantee (Section 5.3).
- We present experimental results showing that the performance overhead for transient is "usable" à la Takikawa et al. [42] and avoids the worst-case slowdowns found in Typed Racket (Section 6).

2. Open-World Soundness and Gradual Typing

In this section, we introduce *open-world soundness* for gradual typing, presenting the general idea and then its formalization and proof technique, and discuss how it benefits users. We then contrast approaches to runtime verification for gradual typing in the context of open-world soundness, and show that the guarded design does not support open-world soundness when the target language of the translation is a spartan host.

2.1 Open-World Soundness in a Nutshell

Many gradually typed languages [9, 27, 46, 48] translate programs written in the *source language* into dynamically typed programs in the *target language*. This translation process does not necessarily produce programs that may safely interoperate with untranslated code in the target language without losing the usual type soundness guarantees of gradual typing. This lack of soundness inhibits the implementation and distribution of typed-and-translated libraries, prevents gradually typed programs from using existing, target-language libraries, and misses out on much of the utility of type annotations.

In this work, we propose an additional property, *open-world soundness*: if a program is well-typed and translated from a gradually-typed surface language into an untyped target, it may interoperate with arbitrary untyped code without producing uncaught type errors. If a translation process fulfills open-world soundness, then programmers are free to write their programs in the gradually-typed source language, using untyped third-party libraries and distributing their own code to untyped clients.

2.1.1 Formalizing Open-World Soundness

To formalize open-world soundness, we must differentiate between terms based on their *origin*: terms e that originate from translated portions of the program are marked $\lozenge$, while program contexts $\mathcal{C}$, which represent target-language code, are marked $\blacklozenge$. With this distinction in place, we define open-world soundness:

Definition (Open World Soundness). *Suppose e_s is a source expression of type T that translates to term e of the target language. The system fulfills* ***open-world soundness*** *if, for any program context $\mathcal{C}$ in the target language, either:*

- $\mathcal{C}[e]$ *reduces to v in zero or more steps for some value v, or*
- $\mathcal{C}[e]$ *diverges, or*
- $\mathcal{C}[e]$ *reduces to a runtime cast error, or*
- $\mathcal{C}[e]$ *reduces to a type error while evaluating in $\mathcal{C}$.*

This definition has the flavor of type soundness with one fundamental difference: instead of prohibiting programs $\mathcal{C}[e]$ in the target language from misbehaving, we ensure that any incorrect behavior is due to the program context $\mathcal{C}$, which may represent a client program interacting with e or the source program e interacting with an untyped library.

2.1.2 Open-World Soundness Helps Programmers

Open-world soundness goes beyond traditional type safety: programmers using gradually typed languages that satisfy open-world soundness can safely write programs that interact with arbitrary programs in the underlying dynamic language. Moreover, programmers may distribute their translated programs as libraries to users of the underlying language, who will benefit from the improved error detection without having to use—or even know about—the gradually typed source language.

Likewise, open-world soundness benefits users who write their own code with static types to ensure correctness, but nevertheless wish to take advantage of external libraries written in the target language. Open-world soundness guarantees that using such libraries will not cause the typed code to misbehave or produce errors, even in the presence of two-way communication between translated code and the target-language library (e.g. via callbacks).

2.2 Gradual Typing Strategies and Open-World Soundness

We now inspect two different approaches to providing runtime type checking for gradual typing and discuss each in the context of open-world soundness.

2.2.1 Guarded Inhibits Open-World Soundness

In the guarded semantics, the traditional approach to gradual typing [23, 33], the programmer annotates part of the program with types and the compiler translates the entire program into a target language. During translation, the types are erased and casts are inserted at every implicit conversion site. At runtime, these casts ensure that values adhere to the specified types, raising runtime errors when they detect type violations. Consider the following example and the translation of its invocation:

```
fun filter (f:int→bool, l:list int)→list int.
  if f(head l)
  then cons (head l) ( filter f (tail l))
  else filter f (tail l)
filter (fun _ (x:⋆)→⋆ . x % 2 = 0) [1,2,3,4]
```

In guarded, the compiler inserts a cast on the anonymous procedure argument to *filter* from $\star \to \star$ to int $\to$ bool during translation.

```
# Guarded Call  Translation
filter ((fun _ (x:⋆)→⋆. x%2 = 0) :: ⋆→⋆⇒ℓ0 (int→bool))
        [1,2,3,4]
# ⟶* [2,4]
```

While casts on first-order values are checked immediately when evaluated at runtime, casts on functions and objects in guarded install *proxies*: in the example above, the cast from $\star \to \star$ to int $\to$ bool installs a wrapper on the original procedure that, when called, casts the input from int to $\star$, calls the underlying procedure, and then casts the result from $\star$ to bool [18].

Unfortunately, these proxies interfere with open-world soundness on spartan hosts by inhibiting sound module interactions and sound foreign-function calls.

Guarded *inhibits sound module interaction.* Without modifying the target-language runtime to reason about proxied values, translated programs cannot soundly interact with programs written in the target language. Guarded dictates that *callers*, not *callees*, are responsible for type safety when interacting with typed code. For example, consider the following function definition:

```
isEven ≜ fun isEven (n:int)→bool. (n % 2) = 0
```

After guarded translation, this program contains no casts; the translation process proved that the output will always be a bool, and so the program is provided without additional annotation:

```
## Guarded Translation
isEven ≜ fun isEven n. (n % 2) = 0
```

Unfortunately, while the caller is responsible for invoking *isEven* with an integer, nothing enforces this responsibility. As a result, a program in the target language may incorrectly invoke *isEven*:

```
# Plain Code (Target Language)
isEven("Hi")
# ⟶* Error in (n % 2) = 0:
#        n is a string, expected an integer
```

Under guarded, the callee fails its responsibility and thus the program yields an error from the internals of the translated program. This error would occur in an untyped version of this program, but is now more difficult to debug: the programmer may incorrectly think that they can trust their type annotations.

Guarded *inhibits foreign function calls.* Programs translated with guarded semantics may also misbehave when using foreign functions, including built-in functions provided by the language runtime. For example, consider a program that uses built-in `sum` and `length` functions from the spartan target language, which both expect a list (and are assumed to be type $\star \to \star$ by the translation process) and which do not respect or accept proxies.

```
avg      ≜ fun avg (l: list float) → float. (sum l) / (length l)
readFile ≜ fun readFile (name : ⋆) → ⋆. openAndParse name

# main
avg(readFile "arr")
```

After guarded translation, the program is:

```
# Guarded Translation
avg      ≜ fun avg l.
              ((sum (l : list float⇒ℓ1 ⋆)) /
               (length (l : list float⇒ℓ2 ⋆))) :: ⋆ ⇒ℓ3 float
readFile ≜ fun readFile name. openAndParse name

# main
avg ((readFile "arr") :: ⋆ ⇒ℓ0 list float)
# ⟶* Error in sum: l is not a list
```

This program yields an error complaining that l is not a list: when the argument l in *avg* is passed to `sum`, the argument is cast from list float to $\star$ (because `sum` is a built-in function assumed to be type $\star \to \star$.), wrapping l in a proxy. The built-in `sum`, however, expects an unproxied list. When it receives a (structurally different) proxied value at runtime, it produces an error. Worse, the programmer may be misled during debugging: *avg* is typed, so they may assume *readFile* (and not the invocation of `sum` itself) is to blame.

This problem can be further exacerbated by objects and classes. For example, in Reticulated Python (which compiles to Python, a spartan host), a list proxy is a subtype of a list (to maintain the behavior of the commonly-used `isinstance` function), causing further misbehavior. The proxied list is a subtype of list , which means it contains a private, internal list structure, and when the CPython built-in functions operate over a proxied list, they directly manipulate this internal value (instead of the proxied list, as intended). The result is that calls appear to have no effect, and programmers may be at a loss to explain this behavior.

2.2.2 Transient Semantics Support Open-World Soundness

Recent work by Vitousek et al. [48] introduces an alternative semantics, transient, that eschews proxies in favor of shallow runtime *type checks*—lightweight queries that inspect values' "type tags" [6]—rather than proxy-building casts. During translation, the transient design inserts these checks into function bodies (to defensively ensure its inputs have correct type) and at function call sites (to ensure their results have the expected type).

This type-and-check approach allows transient to elide proxies, which solves the pointer-identity problems with guarded semantics [48] and recovers open-world soundness.

Transient *supports sound module interaction.* Under transient, functions are responsible for checking the types of their arguments, function calls are responsible for checking the type of return values, and dereference sites are responsible for checking the type of dereferenced values. As a result, plain programs written in the target language (i.e., Python for Reticulated Python, JavaScript for TypeScript, and Clojure for Typed Clojure) are not responsible for performing any checks themselves, and may invoke their typed-and-translated counterparts and rely on checks within the typed modules to detect and directly report type mismatches to the user. Consider the transient translation of *isEven*, which contains a type check (written n⇓ ⟨ int , *isEven* , ARG⟩) to ensure that its input n is an integer before executing the function's body:

```
## Transient Translation
isEven ≜ fun isEven n. n⇓ ⟨ int , isEven , ARG⟩; (n % 2) = 0
```

In the case that n is not an int, the check contains the information to report the error to the user (in this case, n was an argument in *isEven*, as indicated by ARG):

```
# Plain Code (Target Language)
isEven("Hi")
# ⟶* Error in isEven:
#        isEven was called with "Hi", which is not an int
```

Under transient semantics, the error correctly indicates that the argument to *isEven* was mistyped; the user is now guaranteed that once they've added type annotations to a piece of code, it will not misbehave.

Transient *supports foreign function calls.* Eschewing proxies in transient enables safe interaction with foreign function calls. Consider the transient translation of the *avg*-*readFile* example:

```
# Transient  Translation
avg       ≜ fun avg l.   l⇓ ⟨ list , avg, ARG⟩;
                         (sum (l  :  list float⇒^{ℓ1}⋆)) /
                         (length (l  :  list float⇒^{ℓ2}⋆))

readFile ≜ fun readFile  name . openAndParse name

(avg  (( readFile  "arr")  :: ⋆ ⇒^{ℓ0} list float))⇓ ⟨ float,  avg,  RES⟩
#  ⟶* 47.6
```

Where guarded installed a proxy around the list l before passing it to `sum`, the transient cast at the same place are *only* used to inform blame. As a result, the unproxied list l is passed to `sum`, and it behaves correctly. In lieu of the proxy, the *type checks* at the beginning of *avg* and the call site (written $l \Downarrow \langle$ list ,*avg*,ARG$\rangle$ and ... $\Downarrow \langle$ float,*avg*,RES$\rangle$) inspect the top-level type (or type tag) of each value, ensuring safe interaction and providing the expected type safety to programmers. Moreover, if a built-in or foreign function mutates its input in an ill-typed way, these checks will detect and report the ill-typed value when used.

3. Collaborative Blame

While the transient approach recovers pointer identity and open-world soundness, it does so by removing proxies. In previous work, proxies served as the mechanism for tracking and propagating blame information [18, 45, 50]. The runtime system uses this information when a runtime type error is encountered to report the source of the error—not just the location where the error was discovered and the error was raised, but also which implicit conversion site was violated. This information helps the programmer debug the issue more efficiently.

Traditionally, blame information is propagated through programs at runtime by being included in proxies so that when cast errors occur, the information can be included in errors. Consider the following example, in which *isEven* is cast to $\star \to \star$ and then invoked on a string (all within a gradually typed module):

```
## Guarded  Translation
isEven ≜ fun isEven n.  (n % 2) = 0

let dyFunc = (isEven  ::  int→bool ⇒^{ℓ0}⋆→⋆)
in dyFunc ("Hi"  :: str   ⇒^{ℓ1}⋆)
```

When the cast on *isEven* is applied at runtime, the result is a proxy around *isEven* which includes the information that the proxy was created by a cast with the label ℓ_0. When this proxy is applied to the string "Hi" (which has been casted to $\star$), it casts the argument to `int`. This cast fails and the resulting error message blames ℓ_0, indicating that the cast int$\to$bool$\Rightarrow^{\ell_0}\star\to\star$ is at fault.

Because transient lacks proxies, it is initially unclear that implementing blame tracking is possible in our system. However, without blame information, the programmer may not have enough details to properly diagnose errors and, as such, we develop an alternative mechanism to maintain and propagate this information and report blame errors.

3.1 Runtime Blame Management

We solve this problem by tracking blame information in a global *blame map*, updating the relevant blame information at every implicit conversion. We track when values are passed between different static types by statically inserting casts into the program as in guarded; rather than serving as a type enforement mechanism, these casts only update the blame map—checks are still the main mechanism for detecting type errors. When a check fails, this blame map is used in conjunction with the type information at the failure site to construct the full error account to the programmer. Furthermore, this construction process provides a *blame history*, indicating the conflicting assumptions that different pieces of the program made to produce this error.

Consider the previous example with *isEven*, now using the transient translation:

```
## Transient  Translation
isEven ≜ fun isEven n.  n⇓ ⟨ int ,  isEven ,  ARG⟩; (n % 2) = 0

let dyFunc = (isEven  ::  int→bool ⇒^{ℓ0}⋆→⋆)
in dyFunc ("Hi"  :: str   ⇒^{ℓ1}⋆)
```

When the cast ℓ_0 is applied at runtime to *isEven*, the blame map records the cast. Then, when the check at the beginning of *isEven*'s function body detects that n is not an integer, the runtime attempts to determine which cast (if any) was responsible by looking up the address of the *isEven* function in the blame map, where it will find the cast int$\to$bool$\Rightarrow^{\ell_0}\star\to\star$. Next, the runtime determines if this cast was potentially responsible for the error: the ARG *context tag* at the failed check indicates that it was checking a function argument, and that the cast int$\to$bool$\Rightarrow^{\ell_0}\star\to\star$ is unsafe in its argument positions (due to contravariance). Finally, the runtime finds that the actual argument to the function call is `str`, which conflicts with the domain of the function type `int`, and therefore indicates that the int$\to$bool$\Rightarrow^{\ell_0}\star\to\star$ cast is at fault.

It is not always possible, however, to go directly from a check failure to an incompatible cast. In the following program, *makeEqChecker* takes a string and returns a function that checks its argument against the string. The *makeEqChecker* function is then cast to str $\to \star \to \star$, applied to a string, and then the resulting function is applied to an integer.

```
# Transient  translation
fun makeEqChecker v.
  v⇓ ⟨ str ,  makeEqChecker, ARG⟩;
  fun eqChecker w.
    w⇓ ⟨ str ,  eqChecker,  ARG⟩;
    v  = w

let castFunc=(makeEqChecker ::str→str→bool ⇒^{ℓ0} str→⋆→⋆)
in ((castFunc  "Hi")⇓ ⟨ →,  castFunc,  RES⟩) (42  ::  int⇒^{ℓ1}⋆)
```

At runtime, a check inside *eqChecker* will detect that 42 is not a string. At this point, the runtime will look up *eqChecker* in the blame map in an attempt to find the responsible cast, but *eqChecker* never passed through a cast; it was implicitly cast as a result of the cast on *makeEqChecker*.

However, there is enough information in the inserted casts and checks to tie the check failure with the cast on *makeEqChecker*: when *makeEqChecker* is applied, a check ensures that the result corresponds with the type tag $\to$. This check updates the blame map before returning the result, adding an internal pointer from the result of the function call (the address of this particular instance of *eqChecker*) to the value that returned it (here *makeEqChecker*). This blame map appears in Figure 1.

When the check fails, the runtime must construct blame information. To do so, it traverses the pointer within the blame map from *eqChecker* to *makeEqChecker*, including the context tag RES that indicates *eqChecker* is the result of a call to *makeEqChecker*. The runtime uses this data in collaboration with the cast on *makeEqChecker* to

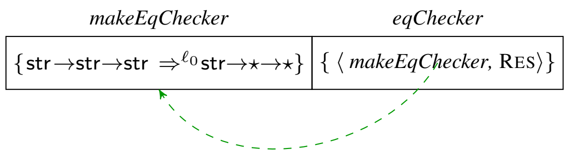

Figure 1. The blame map for *eqChecker* and *makeEqChecker*.

discern that the cast is potentially responsible for the check failure, and ultimately blame it.

3.2 Transient Blame is not Guarded Blame

Through use of the blame map, casts and checks collaborate to reconstruct the chains of responsibility that guarded proxies provide. Even so, the transient blame behavior differs from guarded: the algorithm may blame multiple casts if each of them is reachable in the blame map and *may* be responsible for the check failure occurring (similar to the behavior of the monotonic approach [37]), and if a sequence of incompatible casts are applied to a value but the value is never used (e.g., a function that is never applied), then no error occurs. Even so, this approach preserves the blame-subtyping theorem [50] (as we show in Section 5.1).

4. The Transient Gradual Lambda Calculus $\lambda^{\star}_{\rightarrow}$

In this section, we present the first formal semantics for transient, including a source-to-target translation, runtime semantics, and a blame system.

We begin with the source language $\lambda^{\star}_{\rightarrow}$ with expressions e_s in Figure 2, which includes variables, recursive functions, mutable references, numbers, and addition. $\lambda^{\star}_{\rightarrow}$ also has types T which range over function types $T_1 \to T_2$, reference types $\mathsf{ref}\ T$, integer types int, and the dynamic type $\star$.

Following previous approaches to gradual typing [33, 35, 43, 45], the semantics of $\lambda^{\star}_{\rightarrow}$ are defined by translation into a target language $\lambda^{\Downarrow}_{\ell}$ (as expressions e in Figure 2) which contains type checks as well as the usual type casts. We then define a single-step reduction relation over the $\lambda^{\Downarrow}_{\ell}$ language. Unlike the targets of cast insertion from previous work [23, 33, 50], $\lambda^{\Downarrow}_{\ell}$ is a dynamically typed calculus. Types appear syntactically in the casts of $\lambda^{\Downarrow}_{\ell}$, and shallow type tags S [6] are used in its checks.

This section proceeds as follows: we present the transient translation process (§4.1); the runtime semantics for the target language, including blame machinery for transient (§4.2); and finally present the formal transient blame assignment algorithm (§4.3).

4.1 Translating $\lambda^{\star}_{\rightarrow}$ to $\lambda^{\Downarrow}_{\ell}$

The translation relation, given in Figure 3, converts a type environment Γ and source term e_s into a target term e at type T, inserting type casts $e{::}T \Rightarrow^{\ell} T'$ and type checks $e_1{\Downarrow}\langle S; e_2; r\rangle$.

The translation proceeds as follows:

- Numbers n and variables x are translated to themselves; numbers have type int and the types of variables are given by Γ.
- Addition translates its operands e_{s1} and e_{s2} into e_1 and e_2 and constructs an output expression that casts e_1 and e_2 to integers with new blame labels ℓ_1 and ℓ_2 before performing addition.
- The translation of a function $\mathtt{fun}\ f\ (x{:}T) \to T.\ e_s$ is a $\lambda^{\Downarrow}_{\ell}$ function without type annotations. To ensure that the function argument corresponds to the static type T_1 of the original function input, the translation inserts the type check $x{\Downarrow}\langle T_1; f; \textsc{Arg}\rangle$ before the translated function body. If this check fails, f indicates

$$
\begin{array}{llll}
\lambda^{\star}_{\rightarrow}\ \text{exprs} & e_s & ::= & x \mid \mathtt{fun}\ f\ (x{:}T) \to T.\ e_s \mid e_s\ e_s \\
 & & \mid & \mathtt{ref}\ e_s \mid\ !e_s \mid e_s{:=}e_s \\
 & & \mid & n \mid e_s\ +\ e_s \\
 & & & \\
\text{types} & T & ::= & \mathsf{int} \mid T \to T \mid \star \mid \mathsf{ref}\ T \\
 & & & \\
\lambda^{\Downarrow}_{\ell}\ \text{exprs} & e & ::= & x \mid v \mid \mathtt{fun}\ f\ x.\ e \\
 & & \mid & e\ e \mid e\ +\ e \\
 & & \mid & \mathtt{ref}\ e \mid\ !e \mid e{:=}e \\
 & & \mid & e{::}T \Rightarrow^{\ell} T \mid e{\Downarrow}\langle S; e; r\rangle \\
\text{tags} & r & ::= & \textsc{Res} \mid \textsc{Arg} \mid \textsc{Deref} \\
 & & & \\
\text{type tags} & S & ::= & \mathsf{int} \mid\!\to\!\mid \mathsf{ref} \mid \star \\
 & & & \\
\text{addresses} & a & \in & \text{addresses} \\
\text{labels} & \ell & \in & \text{blame labels}
\end{array}
$$

Figure 2. Syntax of $\lambda^{\star}_{\rightarrow}$ and $\lambda^{\Downarrow}_{\ell}$.

$\boxed{\Gamma \vdash e_s \rightsquigarrow e : T}$

$$
\frac{}{\Gamma \vdash n \rightsquigarrow n : \mathsf{int}}
\qquad
\frac{\Gamma(x) = T}{\Gamma \vdash x \rightsquigarrow x : T}
$$

$$
\frac{\begin{array}{lll}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T_1 & T_1 \sim \mathsf{int} & \text{fresh}(\ell_1) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_2 & T_2 \sim \mathsf{int} & \text{fresh}(\ell_2)\end{array}}{\Gamma \vdash e_{s1}\ +\ e_{s2} \rightsquigarrow (e_1{::}T_1 \Rightarrow^{\ell_1} \mathsf{int}) + (e_2{::}T_2 \Rightarrow^{\ell_2} \mathsf{int}) : \mathsf{int}}
$$

$$
\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash e_s \rightsquigarrow e' : T_2' \qquad T_2 \sim T_2'}{\begin{array}{c}\Gamma \vdash \mathtt{fun}\ f\ (x{:}T_1) \to T_2.\ e_s \rightsquigarrow \\ \mathtt{fun}\ f\ x.\ (\mathtt{let}\ x = x{\Downarrow}\langle \lfloor T_1 \rfloor; f; \textsc{Arg}\rangle\ \mathtt{in}\ e') : T_1 \to T_2\end{array}}
$$

$$
\frac{\begin{array}{lll}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T & T \rhd T_1 \to T_2 & \text{fresh}(f) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_1' & T_1 \sim T_1' & \text{fresh}(\ell)\end{array}}{\begin{array}{c}\Gamma \vdash e_{s1}\ e_{s2} \rightsquigarrow \mathtt{let}\ f = e_1{::}T \Rightarrow^{\ell} T_1 \to T_2\ \mathtt{in} \\ (f\ (e_2{::}T_1' \Rightarrow^{\ell} T_1)){\Downarrow}\langle \lfloor T_2 \rfloor; f; \textsc{Res}\rangle : T_2\end{array}}
$$

$$
\frac{\Gamma \vdash e_s \rightsquigarrow e : T}{\Gamma \vdash \mathtt{ref}\ e_s \rightsquigarrow \mathtt{ref}\ e : \mathsf{ref}\ T}
$$

$$
\frac{\Gamma \vdash e_s \rightsquigarrow e : T \qquad T \rhd \mathsf{ref}\ T_1 \qquad \text{fresh}(x) \qquad \text{fresh}(\ell)}{\Gamma \vdash !e_s \rightsquigarrow \mathtt{let}\ x = e{::}T \Rightarrow^{\ell} \mathsf{ref}\ T_1\ \mathtt{in}\ !x{\Downarrow}\langle \lfloor T_1 \rfloor; x; \textsc{Deref}\rangle : T_1}
$$

$$
\frac{\begin{array}{lll}\Gamma \vdash e_{s1} \rightsquigarrow e_1 : T & T \rhd \mathsf{ref}\ T_1 & \text{fresh}(\ell_1) \\ \Gamma \vdash e_{s2} \rightsquigarrow e_2 : T_1' & T_1 \sim T_1' & \text{fresh}(\ell_2)\end{array}}{\Gamma \vdash e_{s1}{:=}e_{s2} \rightsquigarrow (e_1{::}T \Rightarrow^{\ell_1} \mathsf{ref}\ T_1){:=}(e_2{::}T_1' \Rightarrow^{\ell_2} T_1) : \mathsf{int}}
$$

$\boxed{\lfloor T \rfloor = S}$

$$
\begin{array}{lcl}
\lfloor \star \rfloor = \star & \qquad & \lfloor \mathsf{int} \rfloor = \mathsf{int} \\
\lfloor T_1 \to T_2 \rfloor = \to & & \lfloor \mathsf{ref}\ T \rfloor = \mathsf{ref}
\end{array}
$$

$\boxed{T \rhd T}$

$$
\begin{array}{lcl}
\mathsf{ref}\ T \rhd \mathsf{ref}\ T & \qquad & \star \rhd \mathsf{ref}\ \star \\
T_1 \to T_2 \rhd T_1 \to T_2 & & \star \rhd \star \to \star
\end{array}
$$

$\boxed{T \sim T}$

$$
\mathsf{int} \sim \mathsf{int} \qquad \star \sim T \qquad T \sim \star
$$

$$
\frac{T_1 \sim T_2}{\mathsf{ref}\ T_1 \sim \mathsf{ref}\ T_2}
\qquad
\frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}
$$

Figure 3. Translation from $\lambda^{\star}_{\rightarrow}$ to $\lambda^{\Downarrow}_{\ell}$.

the function being eliminated by the call and ARG indicates that the failure was the result of an ill-typed argument.

- Application translates its arguments e_{s1} and e_{s2} into e_1 and e_2, ensures that e_{s1} has type T, and uses the $\triangleright$ relation ("matching") to ensure that T is either dynamic or a function type, which allows it to be broken down into a source type T_1 and a target type T_2. It also ensures that e_{s2} has type T_1' consistent with T_1, and it casts e_1 to type $T_1 \to T_2$ and e_2 to type T_1. The cast of e_1 to $T_1 \to T_2$ does not ensure that the result of the application has type T_2, however, and thus the translation process wraps the application in a check to ensure that the result has the appropriate type. This check includes the information that e_1 (which we let-bind to f to prevent duplicate evaluation) was the function that was applied and that the result RES of the application is being checked.
- References `ref` e_s translate e_s to e and yield `ref` e.
- Reference mutation translates both sides of the assignment. Similar to function application, casts are inserted to ensure that the left hand side is a reference and that the right hand side matches the reference type.
- Dereferences $!e_s$ are translated by translating e_s to e, using a cast to ensure that its type is consistent with type ref T_1, and, finally, placing a check around the dereference which ensures that the reference being eliminated is at type T_1 (where the eliminated value is the reference itself and the context tag DEREF indicates dereference).

4.2 Reduction Semantics for $\lambda_\ell^{\Downarrow}$

The $\lambda_\ell^{\Downarrow}$ reduction relation, defined in Figure 4, is a single-step reduction that works over configurations of the form

$$\langle e, \sigma, \mathcal{B}\rangle$$

where e is an expression, σ is a heap, and $\mathcal{B}$ is the runtime *blame map*, which associates heap addresses with cast information. We update this blame map $\mathcal{B}$ at cast and check sites (using the utility definition ϱ), associating new blame information with heap addresses a. When a cast or check fails, we use this blame map to assign blame.

The reduction relation $\longrightarrow$ has a number of unusual features:

- Functions are not values in this calculus. Evaluating a function yields a fresh heap address pointing to the function's code (with self-references substituted away). Functions are stored in the heap so that every function has a unique address, which is used in blame tracking.
- Function applications look up the callee's code from the heap and then perform β-reduction as usual.
- Cast expressions $v{::}T_1 \Rightarrow^\ell T_2$ check if the value v corresponds to the tag $\lfloor T_2 \rfloor$ (which is the shallow tag corresponding to the type T_2, as defined in Figure 3) using the *hastype* relation (bottom of Figure 4). Evaluation proceeds as follows, based on *hastype*'s result and v's shape:
  - If *hastype*$(\sigma, v, \lfloor T_2 \rfloor)$ and v is not a heap address, v is returned immediately.
  - If *hastype*$(\sigma, v, \lfloor T_2 \rfloor)$ and v *is* a heap address, the blame map $\mathcal{B}$ is updated to record the cast, and v is returned.
  - If the value and the tag do not match, then the result is an error blaming ℓ.

Casts alter $\mathcal{B}$ by extending the blame information associated with a particular address with a new *labeled type* L (compiled from the casts as shown in Figure 5) that annotates each element in the type structure with an optional label indicating whether

$$\begin{array}{llll}
\text{eval. contexts} & E & ::= & \Box \mid E\,e \mid v\,E \mid E + e \mid v + E \\
 & & \mid & \texttt{ref}\ E \mid !E \mid E{:=}e \mid E{::}T \Rightarrow^\ell T \\
 & & \mid & E{\Downarrow}\langle S; e; r\rangle \mid v{\Downarrow}\langle S; E; r\rangle \\
\\
\text{machine states} & \varsigma & ::= & \langle e, \sigma, \mathcal{B}\rangle \mid \text{BLAME}(\mathcal{L}) \\
\\
\text{values} & v & ::= & a \mid n \\
\text{heaps} & \sigma & ::= & \cdot \mid a \mapsto h; \sigma \\
\text{heap values} & h & ::= & (\lambda x.e) \mid v \\
\text{blame sets} & \mathcal{B} & ::= & \cdot \mid a \mapsto \overline{b}; \mathcal{B} \\
\text{blame elems.} & b & ::= & \langle a, r\rangle \mid L \\
\text{labeled types} & L & ::= & \mathsf{int}^q \mid L \to^q L \mid \mathsf{ref}^q\ L \mid \star \mid \bot^\ell \\
\\
\text{label sets} & \mathcal{L} & \subset & \text{blame labels} \\
\text{optional labels} & q & ::= & \ell \mid \epsilon
\end{array}$$

$\boxed{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \varsigma}$

$$\begin{array}{rl}
\langle \texttt{fun}\ f\ x.\ e, \sigma, \mathcal{B}\rangle & \longrightarrow \langle a, \sigma[a \mapsto (\lambda x.e[a/f])], \mathcal{B}\rangle \\
 & \quad \text{where fresh}(a) \\
\langle a\ v, \sigma, \mathcal{B}\rangle & \longrightarrow \langle e[v/x], \sigma, \mathcal{B}\rangle \\
 & \quad \text{where } \sigma(a) = (\lambda x.e) \\
\\
\langle \texttt{ref}\ v, \sigma, \mathcal{B}\rangle & \longrightarrow \langle a, \sigma[a \mapsto v], \mathcal{B}\rangle \\
 & \quad \text{where fresh}(a) \\
\langle !a, \sigma, \mathcal{B}\rangle & \longrightarrow \langle v, \sigma, \mathcal{B}\rangle \\
 & \quad \text{where } \sigma(a) = v \\
\langle a{:=}v, \sigma, \mathcal{B}\rangle & \longrightarrow \langle 0, \sigma[a \mapsto v], \mathcal{B}\rangle \\
 & \quad \text{where } \sigma(a) = v' \\
\\
\langle n_1 + n_2, \sigma, \mathcal{B}\rangle & \longrightarrow \langle n', \sigma, \mathcal{B}\rangle \\
 & \quad \text{where } n' = n_1 + n_2 \\
\\
\langle v{::}T_1 \Rightarrow^\ell T_2, \sigma, \mathcal{B}\rangle & \longrightarrow \langle v, \sigma, \varrho(\mathcal{B}, a, [\![T_1 \Rightarrow^\ell T_2]\!])\rangle \\
 & \quad \text{where } \mathit{hastype}(\sigma, v, \lfloor T_2 \rfloor), v = a \\
\langle v{::}T_1 \Rightarrow^\ell T_2, \sigma, \mathcal{B}\rangle & \longrightarrow \langle v, \sigma, \mathcal{B}\rangle \\
 & \quad \text{where } \mathit{hastype}(\sigma, v, \lfloor T_2 \rfloor), v \neq a \\
\langle v{::}T_1 \Rightarrow^\ell T_2, \sigma, \mathcal{B}\rangle & \longrightarrow \text{BLAME}(\{\ell\}) \\
 & \quad \text{where } \neg(\mathit{hastype}(\sigma, v, \lfloor T_2 \rfloor)) \\
\\
\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle & \longrightarrow \langle v, \sigma, \varrho(\mathcal{B}, a', \langle a, r\rangle)\rangle \\
 & \quad \text{where } \mathit{hastype}(\sigma, v, S), v = a' \\
\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle & \longrightarrow \langle v, \sigma, \mathcal{B}\rangle \\
 & \quad \text{where } \mathit{hastype}(\sigma, v, S), v \neq a' \\
\langle v{\Downarrow}\langle S; a; r\rangle, \sigma, \mathcal{B}\rangle & \longrightarrow \mathit{blame}(\sigma, v, a, r, \mathcal{B}) \\
 & \quad \text{where } \neg(\mathit{hastype}(\sigma, v, S))
\end{array}$$

$\boxed{\varrho(\mathcal{B}, a, b) = \mathcal{B}}$

$$\varrho(\mathcal{B}, a, b) = \mathcal{B}[a \mapsto \mathcal{B}(a) \cup \{b\}]$$

$\boxed{\langle e, \sigma, \mathcal{B}\rangle \longmapsto \varsigma}$

$$\frac{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \langle e', \sigma', \mathcal{B}'\rangle}{\langle E[e], \sigma, \mathcal{B}\rangle \longmapsto \langle E[e'], \sigma', \mathcal{B}'\rangle} \qquad \frac{\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \text{BLAME}(\mathcal{L})}{\langle E[e], \sigma, \mathcal{B}\rangle \longmapsto \text{BLAME}(\mathcal{L})}$$

$\boxed{\mathit{hastype}(\sigma, v, S)}$

$$\frac{}{\mathit{hastype}(\sigma, n, \mathsf{int})} \qquad \frac{}{\mathit{hastype}(\sigma, v, \star)}$$

$$\frac{\sigma(a) = (\lambda x.e)}{\mathit{hastype}(\sigma, a, \to)} \qquad \frac{\sigma(a) = v}{\mathit{hastype}(\sigma, a, \mathsf{ref})}$$

Figure 4. Reduction definitions and semantics for the machine configuration for e.

$$\boxed{[\![T \Rightarrow^{\ell} T]\!] = L}$$

$$\begin{array}{rcl}
[\![\star \Rightarrow^{\ell} \star]\!] & = & \star \\
[\![\mathsf{int} \Rightarrow^{\ell} \mathsf{int}]\!] & = & \mathsf{int}^{\epsilon} \\
[\![\mathsf{int} \Rightarrow^{\ell} \star]\!] & = & \mathsf{int}^{\epsilon} \\
[\![\star \Rightarrow^{\ell} \mathsf{int}]\!] & = & \mathsf{int}^{\ell} \\
[\![T_1 \to T_2 \Rightarrow^{\ell} T_3 \to T_4]\!] & = & [\![T_3 \Rightarrow^{\ell} T_1]\!] \to^{\epsilon} [\![T_2 \Rightarrow^{\ell} T_4]\!] \\
[\![T_1 \to T_2 \Rightarrow^{\ell} \star]\!] & = & [\![\star \Rightarrow^{\ell} T_1]\!] \to^{\epsilon} [\![T_2 \Rightarrow^{\ell} \star]\!] \\
[\![\star \Rightarrow^{\ell} T_1 \to T_2]\!] & = & [\![T_1 \Rightarrow^{\ell} \star]\!] \to^{\ell} [\![\star \Rightarrow^{\ell} T_2]\!] \\
[\![\mathsf{ref}\, T_1 \Rightarrow^{\ell} \mathsf{ref}\, T_2]\!] & = & \mathsf{ref}^{\epsilon}\ [\![T_2 \Leftrightarrow^{\ell} T_1]\!] \\
[\![\mathsf{ref}\, T_1 \Rightarrow^{\ell} \star]\!] & = & \mathsf{ref}^{\epsilon}\ [\![\star \Leftrightarrow^{\ell} T_1]\!] \\
[\![\star \Rightarrow^{\ell} \mathsf{ref}\, T_1]\!] & = & \mathsf{ref}^{\ell}\ [\![T_1 \Leftrightarrow^{\ell} \star]\!] \\
[\![T_1 \Rightarrow^{\ell} T_2]\!] & = & \bot^{\ell}\ \text{otherwise}
\end{array}$$

$$\boxed{[\![T \Leftrightarrow^{\ell} T]\!] = L}$$

$$\begin{array}{rcl}
[\![\star \Leftrightarrow^{\ell} \star]\!] & = & \star \\
[\![\mathsf{int} \Leftrightarrow^{\ell} \mathsf{int}]\!] & = & \mathsf{int}^{\epsilon} \\
[\![\mathsf{int} \Leftrightarrow^{\ell} \star]\!] & = & \mathsf{int}^{\ell} \\
[\![\star \Leftrightarrow^{\ell} \mathsf{int}]\!] & = & \mathsf{int}^{\ell} \\
[\![T_1 \to T_2 \Leftrightarrow^{\ell} T_3 \to T_4]\!] & = & [\![T_3 \Leftrightarrow^{\ell} T_1]\!] \to^{\epsilon} [\![T_2 \Leftrightarrow^{\ell} T_4]\!] \\
[\![T_1 \to T_2 \Leftrightarrow^{\ell} \star]\!] & = & [\![\star \Leftrightarrow^{\ell} T_1]\!] \to^{\ell} [\![T_2 \Leftrightarrow^{\ell} \star]\!] \\
[\![\star \Leftrightarrow^{\ell} T_1 \to T_2]\!] & = & [\![T_1 \Leftrightarrow^{\ell} \star]\!] \to^{\ell} [\![\star \Leftrightarrow^{\ell} T_2]\!] \\
[\![\mathsf{ref}\, T_1 \Leftrightarrow^{\ell} \mathsf{ref}\, T_2]\!] & = & \mathsf{ref}^{\epsilon}\ [\![T_2 \Leftrightarrow^{\ell} T_1]\!] \\
[\![\mathsf{ref}\, T_1 \Leftrightarrow^{\ell} \star]\!] & = & \mathsf{ref}^{\ell}\ [\![\star \Leftrightarrow^{\ell} T_1]\!] \\
[\![\star \Leftrightarrow^{\ell} \mathsf{ref}\, T_1]\!] & = & \mathsf{ref}^{\ell}\ [\![T_1 \Leftrightarrow^{\ell} \star]\!] \\
[\![T_1 \Leftrightarrow^{\ell} T_2]\!] & = & \bot^{\ell}\ \text{otherwise}
\end{array}$$

Figure 5. Compilation of casts to labeled types

or not the associated cast is responsible for introducing that portion of the type [34].

- Check expressions $v\Downarrow\langle S; a; r\rangle$ use the *hastype* relation to compare v and S. If the comparison succeeds, the checked value v is returned unmodified, and if v is higher-order then $\mathcal{B}$ is updated to record the check. Otherwise, the *blame* algorithm (§4.3) is invoked to assign blame. Successful checks on higher order values add blame pointers $\langle a', r\rangle$ to a blame map entry $\mathcal{B}(a)$, where a' is v, the checked value. Cast insertion ensures that a is the value responsible for this check—if the check is on a function call or argument, a is the address of the function, and if it is a check on a dereference, a is the reference. The blame pointer indicates that any casts applied to a' (or any value reachable from a' in $\mathcal{B}$), can potentially affect any future check on a, and may need to be considered when assigning blame. These pointer chains within the blame map allow the runtime system to blame specific casts when a check signals an error.

4.3 Blame Assignment

As previously stated, if the runtime system detects a type violation, it determines which boundary crossing caused the violation to occur and blames it. If a cast fails immediately, we blame the cast itself. If, however, we detect the violation through a check, determining the guilty cast is more complicated: the runtime system must determine which cast(s) are responsible for the failure, and blame each cast that could potentially be responsible for the error that occurred. For example, consider the following code, in which a string reference is casted and passed into two dynamically typed functions which both update it with an integer. The original $\lambda^{\star}_{\to}$ source is on the left and the $\lambda^{\Downarrow}_{\ell}$ translation is on the right.

$\boxed{\lambda^{\star}_{\to}}$

```
g  ≜ fun g (x : ⋆) . x := 42
h  ≜ fun h (y : ⋆) . y := 21
rf ≜ ref "hello"
g rf;
h rf;
! rf
```

$\boxed{\lambda^{\Downarrow}_{\ell}}$

```
g  ≜ fun g x . x := 42
h  ≜ fun h y . y := 21
rf ≜ ref "hello"
g (rf::ref str ⇒^ℓ0 ⋆);
h (rf::ref str ⇒^ℓ1 ⋆);
(! rf) ⇓ ⟨ str , rf, DEREF⟩
```

When the check on the dereference on the last line of the $\lambda^{\Downarrow}_{\ell}$ code occurs, the result is 21, and thus an error is raised. However, the blame assignment algorithm will report that both casts could potentially have led to the error.

Figure 6 shows the *blame* function (used to allocate blame when a check fails), which takes the following arguments:

- the current heap σ,
- the value v which triggered the check failure,
- the heap address a representing the eliminated value of the triggering check,
- a context tag r indicating what operation triggered the check,
- and the current blame set $\mathcal{B}$.

Given these inputs, the algorithm computes a set of labels ℓ which are collectively responsible for the failure, using helper functions *collectblame* and *resolve*.

The *collectblame* function takes a blame element b—either a labeled type L or an internal pointer—and a list of context tags $\overline{r}$, and proceeds based on the shape of b, collecting blame from the blame set $\mathcal{B}$ as follows:

- If the blame element is a pointer $\langle a, r\rangle$, then the pointer's context tag r is prepended to $\overline{r}$, and *collectblame* is recursively invoked on all the blame elements in $\mathcal{B}(a)$.
- If the blame element is a labeled type L, then the *extract* metafunction uses $\overline{r}$ as a "path" through L to extract some L', a subterm of L. For example,

  $$\mathit{extract}(\textsc{Arg}{:}\textsc{Res}{:}\textsc{Deref}, (\mathsf{int}^{\epsilon} \to^{\ell_2} \mathsf{ref}^{\epsilon}\ \mathsf{int}^{\ell_3}) \to^{\ell_1} \mathsf{int}^{\epsilon}) = \mathsf{int}^{\ell_3}$$

  according to the cast labeled ℓ_3. If L' does not have a label, it cannot be blame candidate: the cast that introduced the new type did not change this portion of the type. If a label is attached, however, then the cast could have introduced this error and thus L' is included in the "potential blame set" $\overline{L}$.

Once *collectblame* has constructed a set $\overline{L}$ of blame candidates, the *resolve* metafunction compares v—the actual value that triggered the error—with each $L \in \overline{L}$. If L is $\bot^{\ell}$ or if, after L is converted to type tag S as $\lfloor L\rfloor = S$, S is not related to v by *hastype*, the top-level label ℓ attached to L is one of the labels blamed.

For example, in the following program, the constant function *cnst42* is passed into *fn1* and *fn2*.

$\boxed{\lambda^{\star}_{\to}}$

```
fn1   ≜ fun g (x:int→int)→str . x
fn2   ≜ fun h (y:str→str)→str . y 21
cnst42 ≜ fun cnst42 (n : ⋆) → ⋆ . 42
fn1 cnst42;
fn2 cnst42
```

$\boxed{\lambda_\ell^{\Downarrow}}$

$$
\begin{array}{l}
\mathit{fn1} \triangleq \texttt{fun}\ g\ x\ .\ x \\
\mathit{fn2} \triangleq \texttt{fun}\ h\ y\ .\ (y\ 21){\Downarrow} \langle\, \mathsf{str},\ y,\ \textsc{Res}\rangle \\
\mathit{cnst42} \triangleq \texttt{fun}\ \mathit{cnst42}\ n\ .\ 42 \\
\mathit{fn1}\ \ (\mathit{cnst42}\ :: \star{\to}\star{\Rightarrow}^{\ell_0}\mathsf{int}{\to}\mathsf{int}); \\
\mathit{fn2}\ \ (\mathit{cnst42}\ :: \star{\to}\star{\Rightarrow}^{\ell_1}\mathsf{str}{\to}\mathsf{str})
\end{array}
$$

Casts are applied to *cnst42* when it is passed into both fns. Each cast adds an entry in $\mathcal{B}$ to the casts that have been applied to the address of *cnst42*: $\mathsf{int}^{\epsilon} \to^{\ell_0} \mathsf{int}^{\ell_0}$ for the cast labeled ℓ_0, and $\mathsf{str}^{\epsilon} \to^{\ell_1} \mathsf{str}^{\ell_1}$ for the cast labeled ℓ_1. Then, when *fn2* applies *cnst42* and expects to receive a str as the result, an error is raised because 42 is returned instead. The *collectblame* metafunction looks into the casts that have been applied to *cnst42* in $\mathcal{B}$, and since the check that detected the error was marked with RES, extracts the target type from each of the labeled function types. The result is $\{\mathsf{int}^{\ell_0}, \mathsf{str}^{\ell_1}\}$. The value that actually triggered the error, 42 is related to int by *hastype*, so ℓ_0 cannot be blamed. However, it is not related to str, so the result blames ℓ_1.

5. Blame, Soundness, and the Gradual Guarantee

With our languages, translation, and runtime systems in place, we now present theoretical results for the $\lambda^{\star}_{\to}/\lambda^{\Downarrow}_{\ell}$ formalism, including the blame theorem (§5.1), open-world soundness (§5.2), and the gradual guarantee (§5.3).

5.1 Blame Theorem

The algorithm that transient uses for blame tracking and assignment is dramatically different from the techniques in guarded systems. Nonetheless, it still obeys the *blame-subtyping* theorem [50]: programs whose implicit type conversions are safe will never be blamed for cast failures. Specifically, conversions which are upcasts with respect to the blame subtyping relation (Figure 6) will never be blamed.

In $\lambda^{\Downarrow}_{\ell}$ it is insufficient to reason solely about terms because blame information also appears in the blame map $\mathcal{B}$. Therefore we extend the standard safe relation [2, 34, 37, 50] to use $\mathcal{B}$:

Definition 5.1 (Blame Safety for Terms)**.** *A term e is safe with respect to label ℓ under a blame map $\mathcal{B}$, written $\mathcal{B} \vdash e$ safe ℓ, if there are no unsafe-for-ℓ casts are present in e* and *that no reachable unsafe casts have occurred and been stored in $\mathcal{B}$.*

We define this predicate in Figure A.6 of the supplemental material [49], along with similar predicates over L-types, blame elements b, and heaps σ.

To prove the blame theorem, we must also show that the *blame* algorithm presented above does not blame any cast that the program was safe for. We show this using variants of the standard progress and preservation lemmas, showing that safety is preserved by evaluation and that terms that are safe ℓ cannot blame ℓ when evaluated.

Lemma 5.2 (Blame safety progress)**.** *If $\mathcal{B} \vdash e$ safe ℓ and $\mathcal{B} \vdash \sigma$ safe ℓ and $\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \varsigma$, then $\varsigma \neq \textsc{Blame}(\mathcal{L})$ with $\ell \in \mathcal{L}$.*

Lemma 5.3 (Blame safety preservation)**.** *If $\mathcal{B} \vdash e$ safe ℓ and $\mathcal{B} \vdash \sigma$ safe ℓ and $\langle e, \sigma, \mathcal{B}\rangle \longrightarrow \langle e', \sigma', \mathcal{B}'\rangle$, then $\mathcal{B}' \vdash e'$ safe ℓ and $\mathcal{B}' \vdash \sigma'$ safe ℓ.*

Complete proofs of these lemmas are given in Appendix B.2 of the supplemental material [49]. Finally, we state the blame theorem:

Theorem 5.4 (The Blame Theorem)**.** *For any e_s and T, if*

- $\emptyset \vdash e_s \rightsquigarrow e : T$,

$\boxed{\mathit{extract}(\overline{r}, L) = L}$

$$
\begin{array}{rcl}
\mathit{extract}(\cdot, L) & = & L \\
\mathit{extract}((\textsc{Res} : \overline{r}), L_1 \to^q L_2) & = & \mathit{extract}(\overline{r}, L_2) \\
\mathit{extract}((\textsc{Arg} : \overline{r}), L_1 \to^q L_2) & = & \mathit{extract}(\overline{r}, L_1) \\
\mathit{extract}((\textsc{Deref} : \overline{r}), \mathsf{ref}^q\ L) & = & \mathit{extract}(\overline{r}, L) \\
\mathit{extract}((r : \overline{r}), \star) & = & \star
\end{array}
$$

$\boxed{\mathit{label}(L) = q}$

$$
\begin{array}{rcl}
\mathit{label}(\star) & = & \epsilon \\
\mathit{label}(\mathsf{int}^q) & = & q \\
\mathit{label}(L_1 \to^q L_2) & = & q \\
\mathit{label}(\mathsf{ref}^q\ L) & = & q \\
\mathit{label}(\perp^{\ell}) & = & \ell
\end{array}
$$

$\boxed{\lfloor L \rfloor = T}$

$$
\lfloor \star \rfloor = \star \qquad \frac{}{\lfloor \mathsf{int}^q \rfloor = \mathsf{int}}
$$

$$
\frac{\lfloor L_1 \rfloor = T_1 \qquad \lfloor L_2 \rfloor = T_2}{\lfloor L_1 \to^q L_2 \rfloor = T_1 \to T_2} \qquad \frac{\lfloor L \rfloor = T}{\lfloor \mathsf{ref}^q\ L \rfloor = \mathsf{ref}\ T}
$$

$\boxed{\mathit{collectblame}(\overline{r}, \mathcal{B}, b) = \overline{L}}$

$$
\frac{\mathit{extract}(\overline{r}, L){=}L' \quad \mathit{label}(L') = \ell}{\mathit{collectblame}(\overline{r}, \mathcal{B}, L) = \{L'\}} \qquad \frac{\mathit{extract}(\overline{r}, L){=}L' \quad \mathit{label}(L') = \epsilon}{\mathit{collectblame}(\overline{r}, \mathcal{B}, L) = \emptyset}
$$

$$
\frac{}{\mathit{collectblame}(\overline{r}, \mathcal{B}, \langle a, r\rangle) = \cup_{b \in \mathcal{B}(a)}\ \mathit{collectblame}((r;\overline{r}), \mathcal{B}, b)}
$$

$\boxed{\mathit{resolve}(\sigma, v, \overline{L}) = \mathcal{L}}$

$$
\begin{array}{l}
\mathit{resolve}(\sigma, v, (\perp^{\ell};\overline{L})) = \ell;\mathit{resolve}(\sigma, v, \overline{L}) \\
\mathit{resolve}(\sigma, v, (L;\overline{L})) = \begin{cases} \mathit{label}(L);\mathit{resolve}(\sigma, v, \overline{L}) \\ \quad \text{if } \neg \mathit{hastype}(\sigma, v, \lfloor\lfloor L \rfloor\rfloor) \\ \mathit{resolve}(\sigma, v, \overline{L}) \\ \quad \text{otherwise} \end{cases} \\
\mathit{resolve}(\sigma, v, \cdot) = \cdot
\end{array}
$$

$\boxed{\mathit{blame}(\sigma, v, a, r, \mathcal{B}) = \varsigma}$

$$
\frac{\overline{L} = \cup_{b \in \mathcal{B}(a)}\ \mathit{collectblame}(r, \mathcal{B}, b) \qquad \mathcal{L} = \mathit{resolve}(\sigma, v, \overline{L})}{\mathit{blame}(\sigma, v, a, r, \mathcal{B}) = \textsc{Blame}(\mathcal{L})}
$$

$\boxed{T <:_b T}$

$$
\frac{}{\mathsf{int} <:_b \star} \qquad \frac{T_1 \to T_2 <:_b \star \to \star}{T_1 \to T_2 <:_b \star} \qquad \frac{\mathsf{ref}\ T <:_b \mathsf{ref}\ \star}{\mathsf{ref}\ T <:_b \star}
$$

$$
\frac{}{\mathsf{int} <:_b \mathsf{int}} \qquad \frac{T_3 <:_b T_1 \qquad T_2 <:_b T_4}{T_1 \to T_2 <:_b T_3 \to T_4} \qquad \frac{}{\mathsf{ref}\ T <:_b \mathsf{ref}\ T}
$$

Figure 6. The transient blame-assignment algorithm

- *e contains a subterm e'::$T_1 \Rightarrow^{\ell} T_2$ containing the only occurrence of ℓ in e, and*
- *$T_1 <:_b T_2$,*

then $\langle e, \emptyset, \emptyset\rangle \not\longrightarrow^{*} \textsc{Blame}(\mathcal{L})$ with $\ell \in \mathcal{L}$.

The proof of this theorem is given in Appendix B.2 of the supplemental material [49].

5.2 Open-World Soundness

Next, we focus on *open-world soundness*, which states that a well-typed term in $\lambda^{\star}_{\to}$, translated into $\lambda^{\Downarrow}_{\ell}$, may safely interact with arbitrary $\lambda^{\Downarrow}_{\ell}$ code.

To prove this property we proceed as follows: first, we introduce *origin tracking* for terms, indicating if a term e is a translated $\lambda^{\star}_{\to}$ term or a native $\lambda^{\Downarrow}_{\ell}$ term; then we introduce a type system for $\lambda^{\Downarrow}_{\ell}$ that uses this origin tracking to ensure that translated terms include

$$\begin{array}{rcl} e & ::= & x \mid v \mid \texttt{fun}\, f\, x.\, e \mid (e\ e)^p \mid e\ +^p\ e \mid \texttt{ref}\ e \\ & \mid & !e^p \mid e{:=}^p e \mid e{::}T \Rightarrow^\ell T \mid e\Downarrow\langle T; e; r\rangle \\ p & ::= & \Diamond \mid \blacklozenge \\ \mathcal{C} & ::= & \Box \mid \texttt{fun}\, f\, x.\, \mathcal{C} \mid \mathcal{C}\ e^\blacklozenge \mid v\ \mathcal{C}^\blacklozenge \mid \ldots \end{array}$$

Figure 7. Syntax of $\lambda_\ell^\Downarrow$ with origin tracking.

$\boxed{\langle e, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p}$

$$\frac{}{\langle n\ v^p, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p} \qquad \frac{\sigma(a) = v'}{\langle a\ v^p, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p}$$

$$\frac{}{\langle a\ +^p\ v, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p} \qquad \frac{}{\langle n\ +^p\ a, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p}$$

$$\frac{}{\langle !n^p, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p} \qquad \frac{\sigma(a) = (\lambda x.e)}{\langle !a^p, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p}$$

$$\frac{}{\langle n{:=}^p v, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p} \qquad \frac{\sigma(a) = (\lambda x.e)}{\langle a{:=}^p v, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p}$$

$$\frac{\langle e, \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p}{\langle E[e], \sigma, \mathcal{B}\rangle\ \mathsf{stuck}\ p}$$

Figure 8. Stuck configurations

the appropriate casts and checks; next, we introduce expression contexts for embedding translated expressions into untyped code; and finally we state the open-world soundness theorem and discuss its proof. The entire proof is in the supplemental material [49].

Origin tracking for $\lambda_\ell^\Downarrow$. To prove that only untranslated code can reach a stuck configuration, we must distinguish between translated and untranslated code. We achieve this by introducing *origin tracking* for $\lambda_\ell^\Downarrow$ (similar to the ownership annotations of Dimoulas et al. [13]) and marking the elimination forms of the language (application, addition, dereference, and mutation) with owners. Figure 7 defines this revised version of $\lambda_\ell^\Downarrow$ with origin markers p, which ranges over $\Diamond$, for code translated from well-typed $\lambda^\star_\to$ terms, and $\blacklozenge$, for code that originates in $\lambda_\ell^\Downarrow$ and lacks static types.

With this modified target language, the translation shown in Figure 3 is modified to attach $\Diamond$ markers to translated terms (Figure A.1 of Appendix A in the supplemental material [49]). The reduction rules of Figure 4 are unchanged modulo the addition of markers.

Next, we define a stuck relation over machine configurations (Figure 8). Configurations are stuck if they cannot be reduced by any evaluation rule, similar to the *faulty expressions* of Wright and Felleisen [51]. The stuck relation also indicates whether a stuck state was caused by a $\Diamond$-marked term or a $\blacklozenge$-marked term. (We will prove below that no $\Diamond$ term is ever stuck.)

Origin-sensitive typing for $\lambda_\ell^\Downarrow$. In addition to indicating whether a dynamic type error occurred in typed or untyped code, origin markers also let us define a type system for $\lambda_\ell^\Downarrow$ that places restrictions on translated code. We use this type system to state and prove open-world soundness. This type system relates expressions to type tags S, as defined in Figure 2, which are repurposed as types.

The typing rules are shown in Figure 9. There are two rules for each marked expression, one for $\Diamond$ and one for $\blacklozenge$.

Each $\blacklozenge$ rule requires that all subexpressions be typed at $\star$ (which may subsume any other type via TSUBSUMP), and thus no programs may be ill-typed unless they contain $\Diamond$-marked expressions. The $\Diamond$ rules are more restrictive and allow us to prove that a $\Diamond$-marked expression is well-typed when it cannot become stuck, and thus $\Diamond$ rules require that subexpressions have the appropriate types to ensure $\longrightarrow$ reducibility.

$\boxed{\Gamma; \Sigma \vdash e : S}$

$$\text{(TVAR)}\ \frac{\Gamma(x) = S}{\Gamma; \Sigma \vdash x : S} \qquad \text{(TADDR)}\ \frac{\Sigma(a) = S}{\Gamma; \Sigma \vdash a : S} \qquad \text{(TINT)}\ \frac{}{\Gamma; \Sigma \vdash n : \mathsf{int}} \qquad \text{(TSUBSUMP)}\ \frac{\Gamma; \Sigma \vdash e : S}{\Gamma; \Sigma \vdash e : \star}$$

$$\text{(TCHECK)}\ \frac{\Gamma; \Sigma \vdash e_1 : \star \quad \Gamma; \Sigma \vdash e_2 : \mathit{tagtype}(r)}{\Gamma; \Sigma \vdash e_1\Downarrow\langle S; e_2; r\rangle : S} \qquad \text{(TCAST)}\ \frac{\Gamma; \Sigma \vdash e : \lfloor T_1 \rfloor \quad T_1 \sim T_2}{\Gamma; \Sigma \vdash e{::}T_1 \Rightarrow^\ell T_2 : \lfloor T_2 \rfloor}$$

$$\text{(TFUN)}\ \frac{\Gamma, x : \star, f : \to; \Sigma \vdash e : \star}{\Gamma; \Sigma \vdash \texttt{fun}\ f\ x.\ e : \to}$$

$$\text{(TLET)}\ \frac{\Gamma; \Sigma \vdash e_1 : S_1 \quad \Gamma, x : S; \Sigma \vdash e_2 : S_2}{\Gamma; \Sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : S_2}$$

$$\text{(TAPP)}\ \frac{\Gamma; \Sigma \vdash e_1 : \to \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1\ e_2^\Diamond : \star} \qquad \text{(TAPP-}\star\text{)}\ \frac{\Gamma; \Sigma \vdash e_1 : \star \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1\ e_2^\blacklozenge : \star}$$

$$\text{(TREF)}\ \frac{\Gamma; \Sigma \vdash e : \star}{\Gamma; \Sigma \vdash \texttt{ref}\ e : \mathsf{ref}} \qquad \text{(TDEREF)}\ \frac{\Gamma; \Sigma \vdash e : \mathsf{ref}}{\Gamma; \Sigma \vdash !e^\Diamond : \star} \qquad \text{(TDEREF-}\star\text{)}\ \frac{\Gamma; \Sigma \vdash e : \star}{\Gamma; \Sigma \vdash !e^\blacklozenge : \star}$$

$$\text{(TUPDTREF)}\ \frac{\Gamma; \Sigma \vdash e_1 : \mathsf{ref} \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1{:=}^\Diamond e_2 : \mathsf{int}} \qquad \text{(TUPDTREF-}\star\text{)}\ \frac{\Gamma; \Sigma \vdash e_1 : \star \quad \Gamma; \Sigma \vdash e_2 : \star}{\Gamma; \Sigma \vdash e_1{:=}^\blacklozenge e_2 : \mathsf{int}}$$

$\boxed{\mathit{tagtype}(r) = S}$

$$\mathit{tagtype}(\text{ARG}) = \to \qquad \mathit{tagtype}(\text{RES}) = \to \qquad \mathit{tagtype}(\text{DEREF}) = \mathsf{ref}$$

$\boxed{\Sigma \vdash \sigma}$

$$\text{(THEAP)}\ \frac{\forall a \in \mathit{dom}(\Sigma),\ \Sigma \vdash \sigma(a) : \Sigma(a)}{\Sigma \vdash \sigma}$$

$\boxed{\Sigma \vdash h : S}$

$$\text{(THREF)}\ \frac{\emptyset; \Sigma \vdash v : \star}{\Sigma \vdash v : \mathsf{ref}} \qquad \text{(THFUN)}\ \frac{\emptyset, x{:}\star; \Sigma \vdash e : \star}{\Sigma \vdash (\lambda x.e) : \to}$$

$\boxed{\Sigma \sqsubseteq \Sigma}$

$$\frac{\forall a \in \mathit{dom}(\Sigma_2),\ \Sigma_1(a) = \Sigma_2(a)}{\Sigma_1 \sqsubseteq \Sigma_2}$$

Figure 9. Typing rules for $\lambda_\ell^\Downarrow$ (excluding addition).

Finally, TAPP and TDEREF are judged to have type $\star$. If the result of such an expression is expected to have a more specific type like $\to$ or ref, then a check has to be inserted around the expression. Checks accept expressions of type $\star$ and return the type that the expression is checked against. This design ensures that any $\Diamond$-marked expression uses casts and checks in a defensive way.

Expression contexts for explicit embedding. To reason about code interactions, we use program contexts $\mathcal{C}$ [22], defined in Figure 7. These contexts indicate *how* embedding occurs for translated $\Diamond$ code into $\blacklozenge$ programs, and thus these contexts are all marked with $\blacklozenge$ origin. These contexts are typed in the usual way [22]:

$$\mathcal{C} : \Gamma; S_1 \Rightarrow \Gamma'; S_2$$

The context typing rules are in Figure A.5 of the supplement [49].

Open-world soundness. Equipped with origin markers and a type system for $\lambda_\ell^\Downarrow$, we now state open-world soundness (where $\lfloor \Gamma \rfloor$ is the result of applying $\lfloor T \rfloor$ from Figure 3 to all the types in Γ).

$$\boxed{T \sqsubseteq T}$$

$$T \sqsubseteq \star \qquad \mathsf{int} \sqsubseteq \mathsf{int} \qquad \frac{T_1 \sqsubseteq T_2}{\mathsf{ref}\ T_1 \sqsubseteq \mathsf{ref}\ T_2} \qquad \frac{T_{11} \sqsubseteq T_{21} \qquad T_{12} \sqsubseteq T_{22}}{T_{11} \to T_{12} \sqsubseteq T_{21} \to T_{22}}$$

Figure 10. Type precision

Theorem 5.5 (Open-world soundness)**.** *If* $\Gamma \vdash e_s \rightsquigarrow e : T$ *and* $\vdash \mathcal{C} : \lfloor \Gamma \rfloor; \lfloor T \rfloor \Rightarrow \emptyset; S$*, then* $\emptyset; \emptyset \vdash \mathcal{C}[e] : S$ *and either:*

- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \langle v, \sigma, \mathcal{B} \rangle$ *and* $\emptyset; \Sigma \vdash v : S$ *and* $\Sigma \vdash \sigma$*, or*
- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \textsc{Blame}(\mathcal{L})$*, or*
- $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \langle e', \sigma, \mathcal{B} \rangle$ *and* $\langle e', \sigma, \mathcal{B} \rangle$ stuck $\blacklozenge$*, or*
- *for all* ς *such that* $\langle \mathcal{C}[e], \emptyset, \emptyset \rangle \longrightarrow^* \varsigma$*, have that* $\varsigma = \langle e', \sigma, \mathcal{B} \rangle$ *and exists* ς' *such that* $\langle e', \sigma, \mathcal{B} \rangle \longrightarrow \varsigma'$*.*

The proof is given in Appendix B.1. The proof combines a progress and preservation type soundness proof (for $\lambda^{\Downarrow}_{\ell}$) with proofs that the translation relation is type preserving and that the composition of terms and contexts is well-typed. The progress lemma requires that a configuration with a well-typed term either steps to a new ς or is stuck via a $\blacklozenge$-marked term.

This theorem states that stuck configurations can never arise from evaluating a $\lozenge$-marked term. If $\lozenge$-marked terms could become stuck, it would indicate an uncaught type error in translated $\lambda^{\star}_{\to}$ code. Type soundness is an immediate corollary of this theorem:

Corollary 5.5.1 (Type soundness)**.** *If* $\emptyset \vdash e_s \rightsquigarrow e : T$ *then* $\emptyset; \emptyset \vdash e : \lfloor T \rfloor$ *and either:*

- $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \langle v, \sigma, \mathcal{B} \rangle$ *and* $\emptyset; \Sigma \vdash v : \lfloor T \rfloor$ *and* $\Sigma \vdash \sigma$*, or*
- $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \textsc{Blame}(\mathcal{L})$*, or*
- *for all* ς *such that* $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \varsigma$*, have that* $\varsigma = \langle e', \sigma, \mathcal{B} \rangle$ *and exists* ς' *such that* $\langle e', \sigma, \mathcal{B} \rangle \longrightarrow \varsigma'$*.*

Proof. By Theorem 5.5, taking $\mathcal{C}$ to be the empty context. □

Ramifications of open-world soundness. Because $\lambda^{\star}_{\to}$ admits open-world soundness, a program written in $\lambda^{\star}_{\to}$ can be used by native $\lambda^{\Downarrow}_{\ell}$ clients. For example, an $\lambda^{\star}_{\to}$ library may put type annotations on its API boundaries, preventing ill-typed terms from raising difficult-to-diagnose errors deep within the library—even if the library interacts with code which has no concept of static types.

Furthermore, the $\lambda^{\star}_{\to}$ code is protected from errors arising due to mutation: while foreign functions are not modeled directly in the $\lambda^{\star}_{\to}$ and $\lambda^{\Downarrow}_{\ell}$ calculi, the distinction between untranslated target-language programs and foreign, compiled C code is only relevant in guarded because of the presence of proxies. The transient design lacks proxies, and thus this distinction is irrelevant—foreign functions may be modeled as native $\lambda^{\Downarrow}_{\ell}$ code.

5.3 The Gradual Guarantee

Finally, we prove that the *gradual guarantee* holds for $\lambda^{\star}_{\to}$. The gradual guarantee ensures that changing the static type annotations in a program does not alter either the static or dynamic semantics of the program, except by raising a static type error or causing blame at runtime if the type annotations are strengthened [36]. This property allows programmers to be confident in gradually adding types to their program: they know that a program will never produce an entirely different result because of a change to the type annotations. They are also guaranteed that if a program raises a new error after a type annotation is added or strengthened, it is because the new annotation was "wrong": it did not correspond to other types in the program (if the error is static) or to the program's values at runtime (if it is a runtime blame error).

To prove the gradual guarantee, we use a precision relation for types, also referred to as *naïve subtyping* [34], which is defined in Figure 10. A type T_1 is said to be *more precise* than T_2, written $T_1 \sqsubseteq T_2$, if T_2 contains $\star$ in places where T_1 does not.

We also extend precision to expressions in $\lambda^{\star}_{\to}$. For any two expressions e_{s1}, e_{s2}, we have that $e_{s1} \sqsubseteq e_{s2}$ if the expressions are identical *up to* their type annotations, and if every type annotation in e_{s1} is more precise than the same type annotation in e_{s2}. The specification of expression precision is given in Figure A.3 of the supplemental material [49].

To state the gradual guarantee, we also extend precision to heaps and values. These extensions are straightforward and can be found in Figure A.4. With them, we prove the gradual guarantee for $\lambda^{\star}_{\to}$.

Theorem 5.6 (The gradual guarantee)**.** *If* $e_s \sqsubseteq e'_s$ *and* $\emptyset \vdash e_s \rightsquigarrow e : T$*, then*

1. $\emptyset \vdash e'_s \rightsquigarrow e' : T'$*, with* $T \sqsubseteq T'$*, and*
2. *if* $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \langle v, \sigma, \mathcal{B} \rangle$*, then* $\langle e', \emptyset, \emptyset \rangle \longrightarrow^* \langle v', \sigma', \mathcal{B}' \rangle$ *with* $v \sqsubseteq v'$ *and* $\sigma \sqsubseteq \sigma'$*, and*
3. *if* $\langle e, \emptyset, \emptyset \rangle$ *diverges, then* $\langle e', \emptyset, \emptyset \rangle$ *diverges, and*
4. *if* $\langle e', \emptyset, \emptyset \rangle \longrightarrow^* \langle v', \sigma', \mathcal{B}' \rangle$*, then either* $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \langle v, \sigma, \mathcal{B} \rangle$ *with* $v \sqsubseteq v'$ *and* $\sigma \sqsubseteq \sigma'$*, or* $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \textsc{Blame}(\mathcal{L})$*, and*
5. *if* $\langle e', \emptyset, \emptyset \rangle$ *diverges, then either* $\langle e, \emptyset, \emptyset \rangle$ *diverges or* $\langle e, \emptyset, \emptyset \rangle \longrightarrow^* \textsc{Blame}(\mathcal{L})$*.*

The proof is given in Appendix B.3 of the supplement [49]. Part 1 is proved by induction on $e_s \sqsubseteq e'_s$. Parts 2 and 3 are proved by first showing that $e \sqsubseteq e'$, and then proving a simulation between the evaluation of e and e'. Parts 4 and 5 are corollaries of parts 2 and 3.

Part 1 of this theorem indicates that a less precise expression will always typecheck successfully and be translated into a $\lambda^{\Downarrow}_{\ell}$ expression if a more precise one does: removing or weakening type annotations will never cause a program to behave worse. Parts 2 and 3 show that the weakening of type annotations from a program can never cause the program to behave differently: if the stronger program diverges, so will the weaker one, and if it returns a result, the weaker one will return a result that is weaker than it. Finally, parts 4 and 5 show that strengthening a program's type annotations can only cause it to behave differently than a weaker one by producing a blame error—if the weaker program diverges, then the stronger one will either go to blame or also diverge, and if the weaker program returns a result, then the stronger one will either go to blame or return a stronger result. Adding type annotations can never, for example, cause a program to diverge if it didn't before.

6. Implementation and Evaluation

In this section, we discuss our implementation of the transient system with blame in Reticulated Python, an implementation of gradual typing for the Python language. We discuss the importance of open-world soundness in this setting and then show experimental performance results.

Reticulated Python provides a static typechecker, a source-to-source translator from type-annotated Python-like programs to Python 3 programs with the appropriate casts and checks, and runtime libraries that implement these casts and checks. While Reticulated supports several gradual typing designs, we focus on extending the transient semantics with blame tracking as presented in the previous sections. During evaluation, casts are recorded and associated with the casted value in a global map and checks add internal pointers within the map and use the algorithm described in Section 4.3. When an error occurs, the runtime uses this map to identify the responsible parties.

6.1 Open-World Soundness and Reticulated Python

We conjecture that Reticulated Python is also open-world sound when using the transient semantics; it is certainly closer to open-world soundness than its guarded semantics, and in separate work we proved open-world soundness for Anthill Python, a calculus based on Reticulated and supporting many of Python's features [47]. Vitousek et al. [48] ran each of their case studies without issue under transient, but the guarded counterparts required substantial modification to avoid proxy identity problems and, in the case of the CherryPy web framework, the program was unable to run because proxied values could not correctly interact with Python's pickling library. Likewise, the benchmarks that we test with our modified version of Reticulated Python are able to interact with pure Python libraries without errors or incorrect results.

6.2 Performance of Transient Reticulated Python

The pervasive checks that the transient design uses to ensure open-world soundness come at a runtime cost, but our results indicate that the cost is relatively small, especially when blame-tracking features are disabled.

To analyze the runtime performance of transient, we applied the transient implementation of Reticulated Python to several benchmarks from the official Python benchmark suite.[1] We selected 13 benchmark programs that were compatible with Python 3 and which did not make extensive use of external libraries. (While Reticulated Python is designed to be open-world sound and therefore allow interaction with external libraries, doing so here would shed little light on the cost of runtime soundness since the bulk of execution time would occur in untranslated code.)

We modified each of these benchmarks, inserting static type annotations wherever possible. In some cases, there were parameters or object fields that were inherently dynamic, or which Reticulated Python's type system is unable to provide static types for. In these cases, we defaulted to the dynamic type. Additionally, in several examples we made minor changes to function bodies to avoid trivial dynamicity—for example, when it couldn't change the semantics of the program, changing a list that was initialized with `None` objects and then filled with `ints` (which could only be typed as `List(⋆)`), to a list initialized with numbers (soundly typed at `List(int)`). We then used Reticulated Python to translate the programs using transient semantics and executed the translated program with standard Python 3.4.

Our experiments consider two versions of Reticulated Python's transient implementation: one using the blame tracking technique described in Section 3 and one that does not track blame (merely reporting errors on check failure).

Figure 11 compares the runtime efficiency of the transient translations of the benchmarks with the original untyped code. The green bars show the relative performance of the typed-and-translated programs *without* blame compared to standard Python (the black bars, normalized to 1), while the purple bars show the relative performance of those same programs with blame tracking.

In our tests, the transient system without blame performs at best equally as fast as regular Python, and at worst 5.4x slower. The average slowdown was 2.5x. The test cases that use classes and lists heavily (and thus require checks when members are read from objects or elements read from lists) performed worse than those which primarily used functions and mathematical operators.

The performance degradation exhibited by transient Reticulated Python is significantly less than has been observed in Typed Racket, where in many cases slowdown of over 100x occurs [42]. Transient performs well within the 3/10-usable criterion suggested by Takikawa et al. [42], meaning that no benchmark incurs a more than 10x slowdown compared to the regular Python version.

While this slowdown is not acceptable for all classes of Python programs (where it may be necessary to disable runtime checks for distribution), many other applications are tolerant of this degree of overhead, and in such cases it would be practical to deploy applications with transient's runtime system in place.

Unsurprisingly, Reticulated Python is less efficient when performing blame tracking. As Figure 11 shows, however, this additional cost is never more than 4x compared to the blame-free version, and never more than 18x compared to the original, untyped code. While substantial, this overhead is not necessarily prohibitive: programmers may still use it for developmental debugging, and in most cases it falls within the 3/10-usable range and never incurs as much slowdown as Typed Racket sometimes does. We envision that a common approach, when blame tracking is too expensive, would be to run programs with blame disabled but then re-enable it when a check failure is detected.

7. Related Work

Open-world soundness. Open-world soundness is related to the assertion of Wadler and Findler [50] that "well-typed programs can't be blamed." We adapt this approach to support the dynamically typed target languages of modern, real-world gradual typing.

Allende et al. [4] develop a cast insertion scheme for Gradualtalk [3] intended to facilitate interaction between typed libraries and untyped clients without requiring client recompilation. Their approach, called *execution semantics*, uses callee-installed proxies on function arguments. The callee aspect is similar to the transient semantics, but it still uses proxies and so is vulnerable to problems of object identity and interactions with foreign functions.

Typed Racket [41, 45] includes first-class classes and strives for open-world interaction between Typed Racket modules and untyped Racket, utilizing Racket's software contract system. While Racket provides robust capabilities for module exports and proxied values, the Typed Racket implementation also faces some of the same problems we address in this paper. First, Racket's native proxies inhibit pointer-based equality checking via `eq?` (though Racket's deep equality operator `equal?` looks through proxies). Second, Racket's runtime system accounts for potential proxies when using built-in operations. As stated previously, this style of support would require modifying the Python runtime, impacting Reticulated Python's portability.

Dimoulas et al. [14] introduce *complete monitoring*, a correctness criterion for contract systems that ensures that each contract violation blaming party k is the result of evaluating a module boundary crossing, where the value crossing the boundary is owned by k. *Open-world soundness* similarly uses origin tracking (applied to reducible expressions, rather than values) to ensure that any stuck state is reached by evaluating untranslated code. Open-world soundness also guarantees that the system will detect any errors original to the gradually-typed program.

Gradual typing. Language designers have been interested in mixing static and dynamic typing in the same language for quite some time [1, 10, 11]. Gradual typing [5, 19, 33, 44], was preceded by quasi-static typing of Thatte [43], the Java extensions of Gray et al. [21], Bigloo [32], and Cecil [12]. Gradual typing is distinguished by its use of the consistency relation [5, 33] to govern where typed and untyped code can flow into each other, and where runtime checks need to be performed to ensure safety at runtime. See Siek et al. [36] for a detailed discussion of the core principles that gradual typing aims to satisfy. Blame assignment originated with Findler and Felleisen [18] and was extended to gradual typing by Tobin-Hochstadt and Felleisen [44] and Wadler and Findler [50].

[1] `https://hg.python.org/benchmarks/`

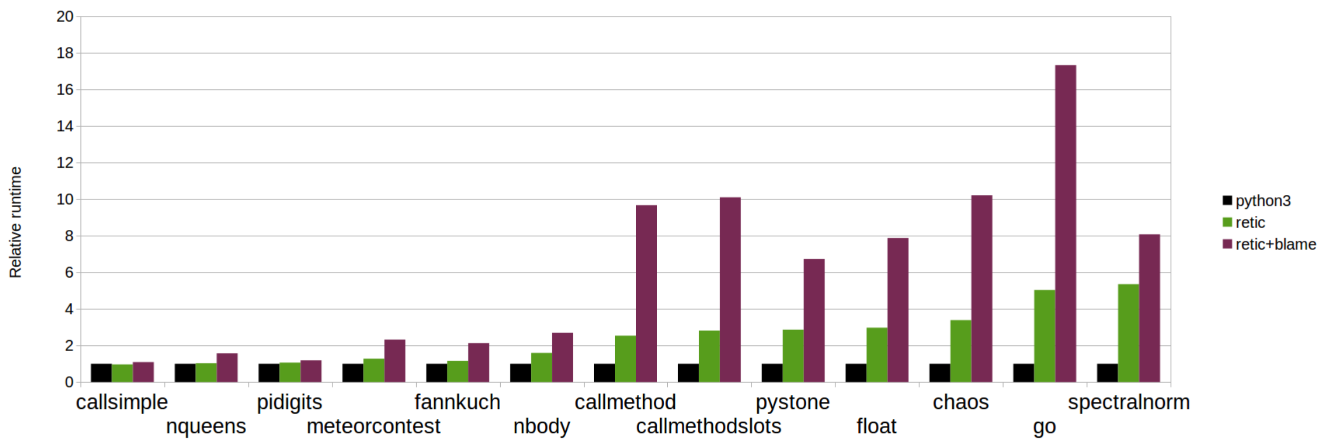

Figure 11. Runtime comparison of Reticulated Python to standard Python 3.4. Experiments were performed on an Ubuntu 14.04 laptop with a 2.8GHz Intel i7-3840QM CPU and 16GB memory.

Another relevant alternative design is the *like-types* approach [8, 52]. This approach avoids proxies by splitting static type annotations into *concrete types* (whose inhabitants are never proxied, and which cannot flow into dynamically typed code) and like types, which can freely interact with dynamic code. This approach was used in designing a sound variant of TypeScript called StrongScript, which obeys open-world soundness [31]. However, because of the incompatibility of concrete and like types, it is not straightforward to evolve a program in StrongScript from dynamic to static, as is frequently desirable. As a result, the *gradual guarantee* of Siek et al. [36] does not hold for these designs.

TS* [39] and Safe Typescript [29] are further variants of TypeScript that implement sound gradual typing. However, compared to the "classical," type consistency based approaches to gradual typing, these languages are restrictive in what kinds of programs are accepted. In these systems, implicit conversions are only allowed on upcasts (using a subtyping lattice similar to that shown in Figure 6). For example, a function of the type any $\rightarrow$ any cannot be cast to bool $\rightarrow$ bool (where any corresponds to $\star$). TS* additionally supports an additional dynamic type beyond any: the un type for unsafe dynamic code. There are no implicit conversions between typed values and un—the programmer must write explicit casts. The gradual guarantee therefore does not hold.

In recent years, gradual typing and ideas related to it have become popular among industrial language designers, with C# [7] adding a dynamic type, and Typescript [27], Dart [20], and Hack [16] offering static typechecking of optional type annotations.

Alternatives to transient semantics. Other approaches also tackle the problem of object identity: Keil and Thiemann [25] present a solution based on the idea of making proxies transparent with respect to identity and type tests. TypeScript [27], Dart [20], and other languages that compile to JavaScript without any runtime checks are trivially free of proxies but their type annotations are not enforced at runtime. Dart offers a *checked mode*, wherein function arguments are checked against optional type annotations, similar to the transient approach, but these checks do not cover all cases and uncaught runtime type errors are still possible [26].

Contracts. Eiffel [15] first popularized software contracts and the idea of writing programs with pervasive contract checking, and it has inspired a large body of research [2, 13, 14, 17, 18, 24]. This work typically relies on a hybridized guarded/transient approach to verification: functions and objects are wrapped in proxies as they move through contracts [38], but contracts are "defensive": callees are *always* responsible for ensuring their inputs and outputs pass their contracts, and callers are absolved of all *programming* responsibility. The transient approach is more defensive: each callee ensures that its inputs have the correct type, and every translated caller ensures that its outputs have the correct type.

Blame tracking was originally invented for software contracts and has been widely studied in that context [2, 13, 18, 24]. While our approach mirrors the overall blame approaches proposed in these works, gradual typing varies due to its need to enforce global type invariants. Moreover, our values do not carry blame information, and thus we introduce a side-channel, global communication model similar to approach described by Swords et al. [40], communicating cast expectations to the blame map during execution.

8. Conclusion

We have discussed an important problem in the implementation of sound gradually typed languages: ensuring soundness of typed programs in the presence of unmoderated interaction with untyped, untranslated code. We refer to this problem as *open-world soundness*. We showed that the traditional guarded design for gradual typing, when embedded in a spartan host, inhibits open-world soundness, but that it holds for the transient design of Vitousek et al. [48]. We developed a novel blame tracking technique that does not rely on proxies and is therefore compatible with the transient design, and we showed that the transient design obeys the gradual guarantee, allowing programmers to freely evolve their code between static and dynamic. By evaluating Reticulated Python's transient design and extending it with blame, we showed that the use of the transient design does not sacrifice usable efficiency. We provided the first formal treatment of transient with the $\lambda^{\star}_{\rightarrow}$ calculus (and its translation target, $\lambda^{\Downarrow}_{\ell}$), and proved open-world soundness, the blame theorem, and the gradual guarantee.

Acknowledgments

We are grateful to Sam Tobin-Hochstadt, Amr Sabry, Ambrose Bonnaire-Sergeant, Matteo Cimini, Andre Kuhlenschmidt, and Christian A. Wailes for their feedback during the development of this work, and to Jim Baker and Zeina Migeed for their assistance and feedback in designing Reticulated Python.

This research was funded by the National Science Foundation under grant number 1518844, *SHF: Large: Gradual Typing Across the Spectrum*, and grant number 1217454, *SHF: Small: Information Effects*.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL*, 1989.

[2] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *POPL*, 2011.

[3] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Markus Denker. Gradual typing for smalltalk. *Science of Computer Programming*, August 2013.

[4] Esteban Allende, Johan Fabry, and Éric Tanter. Cast insertion strategies for gradually-typed objects. In *DLS*, 2013.

[5] Christopher Anderson and Sophia Drossopoulou. BabyJ - from object based to class based programming via types. In *Workshop on Object Oriented Developments*, 2003.

[6] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[7] Gavin Bierman, Erik Meijer, and Mads Torgersen. In *ECOOP*. Springer-Verlag.

[8] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the jvm. In *OOPSLA*, 2009.

[9] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstad. Practical optional types for clojure. In *ESOP*, 2016.

[10] Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *OOPSLA*, New York, NY, USA, 1993. ACM.

[11] Robert Cartwright. User-defined data types as an aid to verifying LISP programs. In *ICALP*, 1976.

[12] Craig Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.

[13] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *POPL*, 2011.

[14] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, 2012.

[15] Eiffel. The Power of Design by Contract. URL `http://www.eiffel.com/developers/design\_by\_contract.html`.

[16] Facebook. Hack, 2013. URL `http://hacklang.org`.

[17] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *FLOPS*, 2006.

[18] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[19] Cormac Flanagan. Hybrid type checking. In *POPL*, Charleston, South Carolina, 2006.

[20] Google. Dart: structured web apps, 2011. URL `http://dartlang.org`.

[21] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*. ACM Press, 2005.

[22] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[23] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming*, 2007.

[24] Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In *ICFP*, 2015.

[25] Matthias Keil and Peter Thiemann. Transparent object proxies in JavaScript. In *ECOOP*, 2015.

[26] Gianluca Mezzetti, Anders Møller, and Fabio Strocco. Type unsoundness in practice: An empirical study of Dart. In *DLS*, 2016.

[27] Microsoft. Typescript, 2012. URL `http://www.typescriptlang.org/`.

[28] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *POPL*, 2012.

[29] Aseem Rastogi, Nikhil Swamy, Cedric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. Technical Report MSR-TR-2014-99, Microsoft Research, 2014.

[30] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The ruby type checker. In *Symposium on Applied Computing*, 2013.

[31] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *ECOOP*, 2015.

[32] Manuel Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.

[33] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[34] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *POPL*, 2010.

[35] Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *ESOP*, 2009.

[36] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL '15*, 2015.

[37] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *ESOP*, 2015.

[38] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*, 2012.

[39] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *POPL*, 2014.

[40] Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. Expressing contract monitors as patterns of communication. In *ICFP*, 2015.

[41] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *OOPSLA*, 2012.

[42] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *POPL*, 2016.

[43] Satish Thatte. Quasi-static typing. In *POPL*, 1990.

[44] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *DLS*, 2006.

[45] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, January 2008.

[46] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP*, 2010.

[47] Michael M. Vitousek and Jeremy G. Siek. Gradual typing in an open world. Technical Report TR729, Indiana University, 2016.

[48] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *DLS*, 2014.

[49] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Supplemental material, January 2017. URL `http://homes.soic.indiana.edu/mvitouse/popl17sup.pdf`.

[50] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, 2009.

[51] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. ISSN 0890-5401.

[52] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. In *POPL*, 2010.