Blame Prediction

Dries Harnie*, Christophe Scholliers, and Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel, Belgium,
{dharnie,cfscholl,wdmeuter}@vub.ac.be

Abstract. Static type systems are usually conservative. Therefore, many interesting programs are rejected by the type system, even though they often execute without errors. Dynamic type systems would allow such ill-typed programs to run. However, the cause of runtime errors is often far removed from the place where the type errors are raised, making the program hard to debug. We present a hybrid typing discipline called *blame prediction* which transforms programs in order to detect runtime type errors as soon as they are guaranteed to happen. These type errors relate the future type error with its cause, aiding in debugging. As a proof of concept, we have applied blame prediction to a functional Scheme-like language.

Keywords: type systems, dynamic typing, blame prediction

1 Introduction

In recent years there has been a surge in the use of dynamically typed programming languages. Developers are building large systems for all kinds of purposes, taking advantage of the fast prototyping and short edit-run-debug cycles offered by these languages. These advantages arise because developers do not need to get their program accepted by the type system. The interpreters for these languages only report errors at runtime when they are detected. It is up to the programmer to discover the expressions that caused these errors.

In the meantime, researchers have developed expressive type systems that attempt to ascribe types to ever-increasing subsets of dynamically typed programming languages. This research is spearheaded by the ongoing research into type systems and developer tools for Ruby and JavaScript [1,2]. This approach guarantees that programs accepted by the type system do not exhibit runtime errors. However, these type systems and tools work on a subset of these dynamic languages and are necessarily conservative, which means that large classes of real-world programs cannot be typed. For example, they often assume array elements have the same type, that both branches of an if-expression return the same type,...

Static and dynamic approaches aim to achieve different goals: static typing approaches are conservative and report errors up front, but they *only* allow programs to run if it can be proven that they never produce errors. By contrast,

* Funded by the Prospective Research For Brussels program of Innoviris, Brussels.

dynamic typing approaches allow every program to run, but only report type errors when primitive operations are applied. This means that statically identifiable type errors will only be reported long after they *could* have been reported.

In this paper we present *blame prediction*: a hybrid technique that makes primitive type tests explicit and performs them as early as possible. The key insight of blame prediction is that primitive operations can be decoupled from the type tests they need to perform, which allows these type tests to be performed *much* earlier. When a type test fails and *all* code paths after the test require it to succeed, a blame prediction error is raised. This error references both the faulty expression and the primitive operations that depend on the type test. Using these blame prediction errors, programmers can more easily debug their code. We have implemented a proof-of-concept blame prediction transformation for a functional core of Scheme. We are convinced blame prediction can be also applied to a whole class of dynamic programming languages and type systems.

The rest of this paper is structured as follows: we describe the motivation and the idea behind blame prediction in sections 2 and 3. Section 4 defines the blame prediction transformation. Related work is described in section 5 and we describe planned work in section 6.

2 Motivation

In this section we explain the tension between dynamic and static typing by exploring a small example: a number guessing game, a typical assignment given to first-year computer science students. This game consists of the user trying to guess a hidden target number, for every guess the program gives feedback like "too high" or "too low". If the user guesses the number, the game is over. Consider the following Scheme implementation of the number guessing game:

```
(define (guess target)
  (let ((input (read)))
    (cond ((eq? input 'quit) 'quit)
          ((< input target)
            (display "Too low!\n") (guess target))
          ((> input target)
            (display "Too high!\n") (guess target))
          (else
            (display "You guessed correctly.\n") 'done))))
```

Listing 1.1. Simple number guessing game in Scheme

This program uses the `read` primitive function to read input from the user and parse it as a Scheme value. If this input is the symbol `'quit`, the game is immediately stopped. Otherwise the input is assumed to be a guess, so it is compared to the `target` number. If the guess was correct, the game ends, otherwise a feedback message is printed and the game continues.

If we try to apply a static type discipline to the program, the following steps will occur: first of all, because the `read` primitive can return a value of *any* type, the type of `input` is inferred as $\top$ (which represents all Scheme types), . In the

first branch of the **cond**, the `eq?` check narrows the type of `input` to symbols, as the `eq?` primitive requires two input expressions of the same type. Then, in the second branch, `input` is numerically compared to `target`, which needs two numeric arguments. As this requires `input` to have a numeric type, a type error is signaled and the program is rejected. Since the program is rejected, we cannot run the program.

A dynamic typing discipline has the opposite result: the program will run normally, but type tests are performed at runtime and errors are raised if they fail. In this case, after the `read` primitive returns a value, it is compared to the symbol `quit`. This does not require a runtime type test, only a pointer equality test. The calls to `<`, `>` and `=` all verify that both `input` and `target` are numbers before they execute the actual comparison code. If these type tests fail, an error is signaled and evaluation is aborted.

Both approaches fall short: static typing simply rejects the program, while dynamic typing accepts the program but it will crash if given the wrong input. We require a hybrid approach: one that applies static typing for the most part, while shielding expressions that might raise errors with type tests. One way of making this program safer is by inserting dynamic type checks in the right places. For example, the code assumes that the `input` variable is either a number or a symbol, as evidenced by the `eq?` and `<` tests. The programmer could therefore insert a type test into the code, right below the **let**:

```
(define (guess target)
  (let ((input (read)))
    (if (not (or (symbol? input) (number? input)))
      (error "Invalid type for input")
      (cond ...))))
```

Listing 1.2. Guessing game with an extra type test

If the user now enters an unexpected kind of input, an error is raised immediately. This program has properties of both the static and dynamic approaches: it allows the program to run as long as no type errors are raised, but an error is raised as soon as possible if errors can be predicted. We discuss how to mechanize this transformation in the next section.

3 Blame Prediction

In this section we specify blame prediction for a functional subset of Scheme called Scheme$_\beta$. Its syntax and semantics are described in Figure 1. Both are standard, but extra attention is paid to errors that can be raised during evaluation.

The most important evaluation rule here is E-Apply, which is responsible for applying both primitive operations and user-defined functions. Evaluation of an application happens as follows: both the function and its arguments are evaluated and then passed to the δ function. If the function being applied is a primitive operation like `+`, the types of the arguments are tested and a err-not-int

error is raised if one of the arguments is not an integer. A second case is when a user-defined function is invoked, in which case the number of arguments must match the arity of the function. Otherwise an err-args-λ error is raised. Finally, application of a non-function value results in a err-not-λ error. Note that — just like in Scheme — these errors are raised at the time of application and propagate outwards, halting the evaluation process.

$$
\begin{array}{rcll}
\mathrm{e} \in \mathrm{Exp} & ::= & x & \text{variables} \\
 & & \#f \mid \#t \mid \mathsf{n} & \text{constants and literals} \\
 & & (e\ e_1 \ldots e_n) & \text{application} \\
 & & (\mathsf{if}\ e\ e\ e) & \text{conditional} \\
 & & (\mathsf{let}\ (x\ e)\ e) & \text{let} \\
 & & (\mathsf{lambda}\ (x_1 \ldots x_n)\ e) & \text{lambda} \\
 & & & \\
v & ::= & \#f \mid \#t \mid \mathsf{n} \mid \lambda x_{1\ldots n}.e & \text{Runtime values} \\
E & ::= & \square \mid (\mathsf{if}\ E\ e\ e) \mid (\mathsf{let}\ (x\ E)\ e) & \text{Evaluation contexts} \\
 & & \mid (E\ e \ldots) \mid (v \ldots E\ e \ldots) &
\end{array}
$$

$$
\begin{array}{rlll}
\text{(E-If-False)} & E\langle(\mathsf{if}\ \#f\ e_1\ e_2)\rangle & \rightarrow E\langle e_2\rangle & \\
\text{(E-If-True)} & E\langle(\mathsf{if}\ v\ e_1\ e_2)\rangle & \rightarrow E\langle e_1\rangle & \text{if } v \neq \#f \\
\text{(E-Let)} & E\langle(\mathsf{let}\ (x\ v)\ e)\rangle & \rightarrow E\langle e[x/v]\rangle & \\
\text{(E-Lambda)} & E\langle(\mathsf{lambda}\ (x_1 \ldots x_n)\ e)\rangle & \rightarrow E\langle\lambda x_{1\ldots n}.e\rangle & \\
\text{(E-Apply)} & E\langle(v_f\ v_1 \ldots v_n)\rangle & \rightarrow E\langle\delta(v_f, v_1, \ldots, v_n)\rangle & \\
\text{(E-Error)} & E\langle\text{err-}\omega(v)\rangle & \rightarrow \text{err-}\omega(v) &
\end{array}
$$

$$
\begin{array}{rcll}
\delta(+, v_1, v_2) & = & v_1 + v_2 & \text{if } \mathsf{int?}(v_1) \wedge \mathsf{int?}(v_2) \\
\delta(+, v_1, v_2) & = & \text{err-not-int}(v_1) & \text{if } \neg\mathsf{int?}(v_1) \\
\delta(+, v_1, v_2) & = & \text{err-not-int}(v_2) & \text{if } \neg\mathsf{int?}(v_2) \\
\delta(\lambda x_{1\ldots m}.e, v_1, \ldots, v_n) & = & e[x_1 \ldots x_m / v_1 \ldots v_n] & \text{if } m = n \\
\delta(\lambda x_{1\ldots m}.e, v_1, \ldots, v_n) & = & \text{err-args-}\lambda(\lambda x_{1\ldots m}.e) & \text{if } m \neq n \\
\delta(v, \ldots) & = & \text{err-not-}\lambda(v) & \text{if } \neg\mathsf{function?}(v)
\end{array}
$$

Fig. 1. Evaluation rules of Scheme$_\beta$

Throughout this and the next section we will use a synthetic example to demonstrate blame prediction. This function switches its operation (and thus its return type) according to the truth value of the `mode` parameter: it either adds one to a number, or prepends a string with "Hello":

```
(define (inc-or-greet mode y)
  (if mode
    (+ y 1)
    (string-append "Hello, " y)))
```

Listing 1.3. Running example

Consider the expression `(inc-or-greet #t (read))`. The `read` primitive reads input from the user and passes it to the `inc-or-greet` function, along

with the boolean value `#t`. Inside this function, the true branch is taken and the `+` primitive is applied to y. If y is not a number, the interpreter raises an error along the lines of "non-numeric value passed to `+`: y".

One observation we can make is that despite the branch inside `inc-or-greet`, the type of the `y` parameter must be `int` or `string`. If this is not case, the invocation of `inc-or-greet` *always* produces an error. Since the type tests for `+` and `string-append` only happen right before they are invoked, users of `inc-or-greet` receive a type error too late. We can therefore perform a type test on `y` upon entering the function and error out if the test returns false:

```
(define (inc-or-greet mode y)
  (check (or (string? y) (number? y))
    (if mode
      (+ y 1)
      (string-append "Hello, " y))))
```

Listing 1.4. `inc-or-greet`, transformed

This `check` macro raises an error if its first argument is $\#f$, otherwise it is equivalent to its second parameter. Note that there is no type test on `mode`, as the if-expression in Scheme$_\beta$ accepts any type of value.

A second observation is that the return type of `inc-or-greet` depends on the path taken through the function (based on the runtime value of `mode`), and the type of the `y` parameter. While we cannot predict the first, we can assert that *if* `y` is of type `int` or of type `string` *and* the evaluation of the body does not result in an error, its return type is the same as that of `y`. We can thus say that `inc-or-greet` is of type

$$((\mathsf{boolean} \times \mathsf{int}) \to \mathsf{int}) \vee ((\mathsf{boolean} \times \mathsf{string}) \to \mathsf{string})$$

We can then use this type to add type checking to code that invokes `inc-or-greet`.

```
(let ((input (read)))
  (check (or (string? input) (number? input))
    (inc-or-greet #t input)))
```

Listing 1.5. transformed application of `inc-or-greet`

4 The Blame Prediction Transformation

In this section we describe the transformation that enables blame prediction. This transformation makes the primitive type tests performed by the runtime system explicit, and aims to perform them as soon as possible. This transformation has two important properties:

Blame may only be predicted if all possible paths result in an error
This property forms the main distinction between blame prediction and a type system: a type system performs all its type tests at verification time and rejects (parts of) programs if they *might* cause a runtime error. By contrast,

blame prediction allows programs to run up until the point where *all* execution paths result in an error. For example, upon entering `(inc-or-greet #t "hi")`, some paths can still succeed so no blame is predicted. Once execution reaches the true branch of the if-expression, all possible paths (namely `(+ y 1)`) will result in an error. This property ensures that blame-predicted programs only raise errors if their original versions do.

Blame prediction may resolve type tests statically
Blame predicted programs are allowed to elide type tests when the inferred type of variables is exactly the requested type or when earlier type tests have already satisfied a type test. Likewise, blame prediction is allowed to replace expressions with type tests that will *always* fail by a static error message. For example, `(inc-or-greet #t #t)` will always fail with a type error, so the function does not need to be entered.

The blame prediction transformation has three stages: type inference (subsection 4.1), insertion of type tests (subsection 4.2), and moving type tests upwards (subsection 4.3). The rest of this section assumes that the input is free of variable shadowing. Additionally, the input program needs to be in A-normal form [3]: this makes the evaluation order explicit, simplifying the logic for moving and insertion of type tests.

4.1 Type system

While blame prediction happens primarily at run-time, the transformation needs to analyze the program beforehand and associate each subtree with a type. Such a type encodes a primitive type like other type systems do but also records all type tests that lead up to yielding a value of this type. The types returned by this analysis are listed in Figure 2. The most unconventional part of this type system are the conditional types $(\tau_1 \sim \tau_2) \cdot \tau_t$: they represent a value that only exists if τ_1 is equivalent[1] to τ_2. τ_1 is always a ground type like int or string, or a function type. Union types combine the types returned by the branches of if-expressions.

There are also function types that represent the different paths through a function along with the type tests they make and their return types. Their arguments are represented by type variables, which get embedded in type tests and return types. Applying these functions yields a type-level application, which substitutes argument types for type variables. If the function type being applied is a type variable, the type-level application is left unresolved until the function type is known. We will point out how each kind of type can be generated by the type rules, shown in Figure 3.

- Rules T-CONST and T-VAR are defined as usual.
- Rule T-IF enables if-expressions to combine the types of both branches using a union type.

[1] The type equivalence operator $\sim$ performs logical equivalence, instead of structural, so $(\mathsf{string} \vee \mathsf{int}) \sim (\mathsf{int} \vee \mathsf{string})$

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{int} \mid \mathsf{string} & \text{ground types} \\
 & \mid & \tau \vee \tau & \text{union types} \\
 & \mid & \alpha & \text{type variable} \\
 & \mid & \Pi\alpha_{1\ldots n}.\tau & \text{function type} \\
 & \mid & (\alpha\ \tau_1 \ldots \tau_n) & \text{function type application} \\
 & \mid & (\tau \sim \tau) \cdot \tau & \text{conditional type} \\
\text{TVar} & \supseteq & \alpha, \beta &
\end{array}
$$

Fig. 2. Types

- Rule T-LET is different from normal type systems because it needs to ensure that the type of the whole let-expression is guarded by the type tests made by e_x. For example, the type of `(let (x (+ y 1)) #t)` should record the fact that `y` is used as a number, even though it is not used in the let body. In order to record this, this type rule decomposes the type of e_x into the types at the leaves (τ_L) and the paths that lead to them. The rationale behind this decomposition is that all type tests in e_x have happened by the time the body is evaluated. The let body is inferred with the type of the let variable bound to the type leaves τ_L, yielding a body type τ. Finally, the type of the let-expression is made by appending the type τ to each path in the type of e_x.
- Rule T-LAMBDA infers the type of the body with the arguments bound to type variables $\alpha_1 \ldots \alpha_n$. This type is then wrapped in a type function with these type variables as arguments. Any type tests performed on the arguments are recorded in the type of the body as well. When this type function is eventually applied (see rule T-APPLY below), the type tests are propagated to the application site and can be eliminated or moved up.
- Rule T-APPLY analyzes function application. This rule infers the type of the called function and its arguments and tries to apply the type function to the arguments using the Apply type function discussed in the next section. The type of the function application is finally made by prepending a type test on the function to the type returned by Apply.
  For bookkeeping purposes, each type test generated by rule T-APPLY is associated with the application node being scrutinized. This enables the later steps of the transformation to assign blame to the correct part of the program in case the type test fails.

The Apply function In the simplest case, the Apply function immediately jumps to the case where τ_f is statically known to be a function. The return type is constructed by substituting the actual types for the argument type variables in the function type's body. Below are some examples of Apply:

$$
\begin{aligned}
\mathrm{Apply}(\Pi\alpha.((\mathsf{string} \sim \alpha) \cdot \mathsf{string}), \{\mathsf{string}\}) &= ((\mathsf{string} \sim \mathsf{string}) \cdot \mathsf{string}) && (1)\\
\mathrm{Apply}(\Pi\alpha, \beta.(\mathsf{int} \sim \alpha) \cdot (\mathsf{int} \sim \beta) \cdot \mathsf{int}, \{\mathsf{int}, \mathsf{int}\}) &= (\mathsf{int} \sim \mathsf{int}) \cdot (\mathsf{int} \sim \mathsf{int}) \cdot \mathsf{int} && (2)\\
\mathrm{Apply}(\mathsf{string} \vee \Pi\alpha.\alpha, \{\mathsf{int}\}) &= \mathsf{error} \vee \mathsf{int} && (3)\\
\mathrm{Apply}(\alpha, \{\mathsf{boolean}, (\mathsf{string} \vee \mathsf{int})\}) &= (\alpha\ \mathsf{boolean}\ (\mathsf{string} \vee \mathsf{int})) && (4)
\end{aligned}
$$

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

$$\frac{c \in \{\#t, \#f, \mathsf{n}\}}{\Gamma \vdash c : \mathrm{Typeof}(c)} \quad \text{(T-CONST)} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{if}\ e\ e_1\ e_2) : \tau_1 \vee \tau_2} \quad \text{(T-IF)}$$

$$\frac{\Gamma \vdash e_x : \tau_1 \qquad \Gamma, x : \tau_L \vdash e : \tau \qquad \tau_L = \mathrm{Leaves}(\tau_1)}{\Gamma \vdash (\mathsf{let}\ (x\ e_x)\ e) : \mathrm{Chain}(\tau_1, \tau)} \quad \text{(T-LET)}$$

$$\frac{\Gamma, x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e : \tau}{\Gamma \vdash (\mathsf{lambda}\ (x_1 \ldots x_n)\ e) : \Pi\alpha_{1\ldots n}.\tau} \quad \text{(T-LAMBDA)}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad \forall i \in [1 \ldots n] \qquad \Gamma \vdash e_f : \tau_f}{\Gamma \vdash (e_f\ e_1 \ldots e_n) : ((\Pi\alpha_{1\ldots n}.\_) \sim \tau_f) \cdot \mathrm{Apply}(\tau_f, \{\tau_{1\ldots n}\})} \quad \text{(T-APPLY)}$$

$\boxed{\text{Auxiliary definitions}}$

$$\begin{array}{lll}
\mathrm{Apply}(\tau_\alpha \vee \tau_\beta, \{\tau_{1\ldots n}\}) & = \mathrm{Apply}(\tau_\alpha, \{\tau_{1\ldots n}\}) \vee \mathrm{Apply}(\tau_\beta, \{\tau_{1\ldots n}\}) & \\
\mathrm{Apply}((\tau_t \sim \tau_\alpha) \cdot \tau, \{\tau_{1\ldots n}\}) & = (\tau_t \sim \tau_\alpha) \cdot \mathrm{Apply}(\tau, \{\tau_{1\ldots n}\}) & \\
\mathrm{Apply}(\Pi\alpha_{1\ldots m}.\tau_f, \{\tau_{1\ldots n}\}) & = \tau_f[\alpha_{1\ldots m}/\tau_{1\ldots n}] & \text{if } m = n \\
\mathrm{Apply}(\Pi\alpha_{1\ldots m}.\tau_f, \{\tau_{1\ldots n}\}) & = \mathsf{error} & \text{if } m \neq n \\
\mathrm{Apply}(\alpha, \{\tau_{1\ldots n}\}) & = (\alpha\ \tau_1 \ldots \tau_n) & \\
\mathrm{Apply}(\tau, \{\tau_{1\ldots n}\}) & = \mathsf{error} & \\
 & & \\
\mathrm{Chain}(\tau_1 \vee \tau_2, \tau_c) & = \mathrm{Chain}(\tau_1, \tau_c) \vee \mathrm{Chain}(\tau_2, \tau_c) & \\
\mathrm{Chain}((\tau_t \sim \tau_1) \cdot \tau, \tau_c) & = (\tau_t \sim \tau_1) \cdot \mathrm{Chain}(\tau, \tau_c) & \\
\mathrm{Chain}(\tau, \tau_c) & = \tau_c & \\
 & & \\
\mathrm{Leaves}(\tau_1 \vee \tau_2) & = \mathrm{Leaves}(\tau_1) \vee \mathrm{Leaves}(\tau_2) & \\
\mathrm{Leaves}((\tau_t \sim \tau) \cdot \tau_1) & = \mathrm{Leaves}(\tau_1) & \\
\mathrm{Leaves}(\tau) & = \tau &
\end{array}$$

Fig. 3. Blame prediction: type inference rules and auxiliary definitions

If the first argument to Apply is a conditional type or a union type, Apply is called recursively while preserving the original structure of the type. This can result in invalid function applications, for example in example 3 above. These applications are replaced with error tokens. After invoking Apply, only the non-error parts of union types are retained. If all parts of a union type result in error, the entire type is error. This is allowed because rule T-APPLY prepends a function test to the type of the application, ensuring only function values are applied. Normal type systems would reject such programs immediately, but blame prediction must continue as such an expression might be buried in a function that is never called or only on some paths. Expressions that are assigned the type error are guarded by type tests, so they are never reached.

Finally, a type variable might be the type function being applied to several type arguments. This can happen as a result of the user creating higher-order functions, which are common in functional languages. The type of such an application is a *type-level* function application, as in example 4 above. Type-level function applications remain in the type until their type variable is replaced by a concrete type, at which point Apply is called anew. This mechanism enables blame prediction to deal with higher-order functions.

After assigning types to all expressions of the program, a round of simplification is performed on the types. This includes:

- eliminating trivially true type tests such as $\mathsf{string} \sim \mathsf{string}$ in example 1,
- pruning type branches that will never succeed ($((\mathsf{int} \sim \mathsf{string}) \cdot \mathsf{int}) \vee \mathsf{string}$ becomes string), and
- merging union types with equivalent branches ($\mathsf{int} \vee \mathsf{int}$ becomes just int).

4.2 Introducing type tests

After inferring types for the program, blame prediction then makes type tests explicit by transforming the program. Type tests are introduced at application sites of both primitive operations and user-defined functions. Every type test is annotated with two labels: a *blame label* that references the function whose preconditions are being checked, and a *cause label* that points to the expression being tested. For example, in the expression `(f x)`, the blame label points to the definition of `f` and the cause is the expression `x`. These labels correspond to source positions in the user's program.

A type test is of the form `[check (`τ`? cause) blame]`. We denote it using square brackets as it is *attached* to nodes in the program, rather than being reified in the code. The next step in the process will move these nodes up and reify them once they reach their final place.

At each application site, the resulting type is converted to a tree structure of `check`s by the type-test function below:

$$
\begin{aligned}
&\text{type-test}((\tau_t \sim \tau) \cdot \tau_f) &&= \texttt{[and [check } (\tau_t\texttt{?}\ e_c)\ l_b\texttt{]}\ \text{type-test}(\tau_f)\texttt{]} \\
&\text{type-test}(\tau_1 \vee \tau_2) &&= \texttt{[or}\ \text{type-test}(\tau_1)\ \text{type-test}(\tau_2)\texttt{]} \\
&\text{type-test}(\_) &&= \#t
\end{aligned}
$$

Remember from subsection 4.1 that each type test is associated with a cause label l_c and a blame label l_b, both referring to source locations. To generate the actual `check`, the blame prediction transformation first looks up the variable or constant e_c at label l_c and inserts it into a primitive type test, annotated with the blame label l_b.

This tree structure is then simplified according to the normal logic formulas for **and** and **or**: $\#t$ is removed from **and** conditions and **and** conditions with a single element are replaced by that element. If the type of an application contains no type tests, it is just $\#t$. For example, the type $(\mathsf{int} \sim (\mathsf{int} \vee \mathsf{string})) \cdot \mathsf{int}$ with $l_b = +$ and $l_c = x$ becomes `[check (int? x) +]` after simplification.
After type test introduction, the `inc-or-greet` example looks like Listing 1.6. Note that each `[check]` expression is an annotation on the expression after it.

```
(define (inc-or-greet mode y)
  (if mode
    [check (int? y) +] (+ y 1)
    [check (string? y) string-append]
      (string-append "Hello, " y)))
```

Listing 1.6. `inc-or-greet` example after type test insertion

4.3 Moving type tests upwards

The last step of the blame prediction transformation attempts to move the `check` nodes as far up the evaluation tree as possible. This process (called "flotation") is applied to the expression tree in a bottom-up fashion; it is subject to a few simple rules. Because the program is in ANF, there are only a few cases to consider, listed in Figure 4. Flotation rules are of the form $e \mapsto e' \uparrow \mathsf{C}$, meaning that expression e can be rewritten to expression e', floating checks C upwards. The expressions may also contain check nodes C of their own.

- Rules F-CONST and F-VAR are for completeness: as we only introduce `check` nodes at function application sites, constants and variables will never give rise to a type test.
- The F-APPLY rule simply floats its checks upwards.
- Rule F-IF floats the disjunction of the checks performed by both nodes, while still performing these checks in the branches themselves. This enables the `inc-or-greet` example to predict blame if `y` variable contains something not of type `int` or `string`.
- Rule F-LET ensures that let-expressions only float checks upwards which do not involve the bound variable. Checks that do are replaced by #t.
- Rule F-LAMBDA stops checks from floating past lambda-expressions, as these type tests would only be performed when the function is actually applied.

Remember from subsection 4.1 that trivially true type tests (eg. int? 5) are eliminated as part of the simplification. However, type tests that *always* fail remain present in generated types; they are introduced and floated upwards

together with the other constraints. Such type tests are not stopped by let-expressions, so they have the potential to float all the way to the top of the program. When the program is then executed, these type tests immediately predict blame and stop, much like a static type system prevents an ill-typed program from running.

$$c \mapsto c \uparrow \#t \quad \text{(F-CONST)} \qquad\qquad x \mapsto x \uparrow \#t \quad \text{(F-VAR)}$$

$$[\mathsf{C}](e\ e_1 \ldots e_n) \mapsto (e\ e_1 \ldots e_n) \uparrow \mathsf{C} \quad \text{(F-APPLY)}$$

$$(\mathsf{if}\ e\ [\mathsf{C}_1]e_1\ [\mathsf{C}_2]e_2) \mapsto (\mathsf{if}\ e\ [\mathsf{C}_1]e_1\ [\mathsf{C}_2]e_2) \uparrow \texttt{[or C}_1\ \texttt{C}_2\texttt{]} \quad \text{(F-IF)}$$

$$\frac{\mathsf{C}' = \mathsf{C} \text{ with all } \texttt{check}\text{s involving } x \text{ replaced by } \#t}{(\mathsf{let}\ (x\ [\mathsf{C}_x]e_x)\ [\mathsf{C}]e) \mapsto (\mathsf{let}\ (x\ e_x)\ [\mathsf{C}]e) \uparrow \texttt{[and C}_x\ \texttt{C}'\texttt{]}} \quad \text{(F-LET)}$$

$$(\mathsf{lambda}\ (x_1 \ldots x_n)\ [\mathsf{C}]e) \mapsto (\mathsf{lambda}\ (x_1 \ldots x_n)\ [\mathsf{C}]e) \uparrow \#t \quad \text{(F-LAMBDA)}$$

Fig. 4. Rules for floating checks up the evaluation tree

Floating type tests in the `inc-or-greet` example finally yields the following program, which is what we wanted to accomplish in section 3.

```
(define (inc-or-greet mode y)
  [or [check (int? y) +] [check (string? y) string-append]]
  (if mode
    [check (int? y) +] (+ y 1)
    [check (string? y) string-append]
      (string-append "Hello, " y)))
```

Listing 1.7. `inc-or-greet` example after floating type tests up

We have also applied blame prediction to the `guess` example from the introduction, the resulting program is shown in section A.

5 Related work

As mentioned in the introduction, there exists a huge body of work [1,2] on ever more expressive type systems for dynamically typed programming languages, with the ultimate goal of being able to statically type all dynamic programs. These approaches limit themselves to a subset of the language they are studying, resulting in an inability to type whole classes of programs that will never raise a runtime error. The outcome of this body of work can be used to improve the static capabilities of blame prediction, at least for programs that can successfully be typed.

Soft typing [4] was among the first attempts at inserting type tests into dynamically typed programs. Their type system featured guarded primitives like `CHECK-car` and also marked subexpressions as "will always fail". The primary motivation for inserting these type tests is to allow type inference to proceed, but errors will only be reported if faulty expressions are evaluated. Blame prediction also floats these type tests up as much as possible, reporting errors earlier.

Recently, gradual typing [5] has acknowledged that programmers may want to gradually convert their programs to static typing. In the cases where normal type systems cannot reason over the entire program, gradual typing can at least be used to statically type check parts of the program. Parts that cannot be typechecked are assigned the unknown type '**?**' and interactions with statically typed code is guarded by a type conversion such as $\langle x \Leftarrow \tau \rangle$. Gradual typing was a big inspiration for this research, with the observation that the type tests performed by a gradually typed program can be performed earlier in the control flow.

Aside from primitive operations, explicit type tests can also be used to deduce type information in a dynamically typed language. The work in [1] sidesteps the "one variable, one type" restriction that is present in many type systems, instead opting to make types flow-sensitive. We plan to incorporate manual type tests in later versions of blame prediction to better estimate types, thus eliminating more type tests and floating tests up even further.

A recent addition to the Glasgow Haskell Compiler has lead to deferred type errors [6]. Rather than halting compilation when a type error is discovered, the program is compiled as normal but the wrongly-typed expressions are replaced by "holes". When such a hole is accessed by the runtime system, an error is raised as before. These errors are only raised as they are accessed however, even if the compiler can predict that the hole must be entered.

6 Discussion & Future Work

As blame prediction is still research in progress, in this section we discuss some of the areas that are still being worked on. Our implementation can be found at `https://github.com/botje/crystal`.

Recursive functions The mechanism for inferring types for functions currently cannot cope with recursive functions, as substitution would produce infinite type terms. For example, consider the following function:

```
(define (count n)
  (if (= n 0)
    "done"
    (count (- n 1))))
```

Listing 1.8. Recursive counting function

The inferred type would be count $= \Pi n.(\mathsf{int} \sim n) \cdot (\mathsf{string} \vee (\mathrm{count}\ n))$.
In the implementation of blame prediction we infer recursive functions in two

steps: first we infer the type with the recursive function name bound to a dummy function type. This function type accepts the same number of arguments, but only has a fresh type variable as body. After this inference step, we analyze the leaves of the function's type to determine which types are produced by non-recursive paths through the body. A second round of type inference is then done, with the recursive function's name bound to a function that returns this new return type. If *all* leaves are equal to the dummy type variable, the function never returns anything, so its type is left as "any". If only some leaves match this dummy type variable, the other (concrete) types in the union type are the return type. Finally, if none of the leaves match the dummy type variable, we use the return type as is.

For the `count` function above, the derivation goes as follows, where count_1 is the type of count after the first inference, and count_2 the type after the second inference.

$$\begin{aligned} \text{count}_1 &= \Pi n.(\mathsf{int} \sim n) \cdot (\mathsf{string} \vee \alpha_{dummy}) \\ \text{Leaves}(\text{count}_1) &= \mathsf{string} \vee \alpha_{dummy} \\ \text{count}_2 &= \Pi n.(\mathsf{int} \sim n) \cdot (\mathsf{string} \vee \mathsf{string}) \end{aligned}$$

This process can be performed analogously for groups of mutually recursive functions.

Redundant tests The algorithm as-is generates a lot of redundant tests: for example, if a variable is used as a number multiple times, each use gives rise to a type test with a different blame label. These tests will float to the top of the function they are in, clustering at the top. If a type test can refer to multiple blame labels, instead of only referring to one blame label, these clusters can be avoided.

Applying blame prediction to other type systems In this paper we used blame prediction to perform type tests in advance in order to improve early error detection. However, blame prediction can be used for other purposes as well. For example, languages sometimes offer "unsafe" variants of operators that promise a performance increase. Programmers are allowed to use these unsafe operators if they verify certain conditions up front. Using a mechanism similar to blame prediction, safe operators can automatically be split into unsafe operators plus tests to verify these conditions. These tests can then be floated upwards and combined with other tests, yielding a faster program without sacrificing safety.

Another technique that can be ported to blame prediction is tainting [7], where input from the user or the filesystem is considered as "tainted". Applying a function to a tainted value produces a value that is itself again tainted. Some primitive functions, like `system` or `open` raise a fatal error if they are invoked with tainted input. This flow of taint is easily described using type rules; a "taint predicted" program can point out areas where the developer overlooked tainting.

7 Conclusion

In this paper, we presented a novel typing discipline called blame prediction. The key insight of this research is that dynamically typed programs perform type tests on expressions only at the call site of primitive operations, while these tests could be performed considerably earlier. We have formalised this insight into a type system that makes type tests an integral part of expressions' types. These types are used to steer a program transformation that makes type tests explicit in code. The end result is a program that performs dynamic type tests well before they are needed, with pointers to the failing expression and the code that needs it. This in turn helps developers debug their applications faster and structure their program better.

References

1. Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing Local Control and State Using Flow Analysis. In: Proceedings of the 20th European Symposium on Programming. (2011) 256–275
2. Furr, M., An, J., Foster, J., Hicks, M.: Static type inference for Ruby. Proceedings of the ACM Symposium on Applied Computing (2009) 1859–1866
3. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. (1993) 237–247
4. Wright, A.K., Cartwright, R., Wright, A.K., Cartwright, R.: A practical soft type system for Scheme. ACM (1994)
5. Siek, J., Taha, W.: Gradual typing for functional languages. Workshop on Scheme and Functional Programming (2006)
6. Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Equality proofs and deferred type errors: A compiler pearl. Proceedings of the International Conference on Functional Programming (2012) 1–12
7. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the International Symposium on Software testing and Analysis. (2007)

A Blame prediction applied to `guess`

This section showcases how the `guess` example from the introduction is transformed by our current blame prediction implementation. It is not part of the main article because `guess` is recursive, which is still work in progress. Bracketed code denotes a type test annotation for the code after it. To keep the example brief, type tests are written as `(test cause blame)`.
The type of `guess` is $\Pi\alpha.\mathsf{symbol} \vee ((\mathsf{int} \sim \alpha) \cdot \mathsf{symbol})$.

```
(letrec ((guess (lambda (target)
  [or #t [and (number? target <) [or #t (number? target >)]]]
  (let ((input (read)))
    [or #t
        [and (number? input <) (number? target <)
             [or #t
                 [and (number? input >) (number? target >)]]]]
    (if (eq? input 'quit)
       'quit
       [and (number? input <) (number? target <)
            [or #t [and (number? input >) (number? target >)]]]
       (if (< input target)
          (begin (display "Too low!\n") (guess target))
          [and (number? input >) (number? target >)]
          (if (> input target)
             (begin (display "Too high!\n") (guess target))
             (begin (display "You win!\n") 'done)))))))
    (guess 4))
```

Listing 1.9. `guess` after blame prediction