Type Systems for Incorrectness: Vision and Approach

Steven Ramsay (PL), University of Bristol

1 Summary of the Proposal

Type-safe programming languages provide a guarantee: programs that pass the typechecker are sure to be free from certain kinds of type violations at runtime: they will not crash trying to, e.g. call an integer literal $3()$ as if it were a function, or pass a string to a method that expects only an integer.

However, the type system provides no guarantees about those programs that fail typechecking. For each program that doesn't typecheck, there are two possibilities: (i) it could be defective - it will crash at runtime due to a type violation, or (ii) it could be completely bug-free, but the typechecker is unable to *prove* it. For example, the Java conditional expression true ? 3 : "foo" has no danger of crashing at runtime, but is rejected by the Java typechecker.

The fact that type systems cannot distinguish real bugs is due to an inherent limitation in the underlying theory. In the traditional approach to type systems for programming languages, only the well-typed programs can be assigned types, and well-typed programs don't go wrong. Thus, types can say nothing about *program incorrectness* - programs that *do* go wrong. However, the recent explosion of work on *incorrectness* program logics for first-order imperative programs, popularised by O'Hearn's Incorrectness Logic (IL) [6], has clearly demonstrated that reasoning about programs that do go wrong is tremendously useful in practice[1].

This proposal describes a programme of work to develop the ***theory and practice of type systems for reasoning about incorrectness***.

These are type systems that would be able to prove that programs do go wrong, e.g. to prove that a given program will crash at runtime with a memory fault, or that a program is certain to reveal some private data. A type error in such a system *guarantees that there is an issue in the code which will manifest at runtime* and, moreover, the content of the proof can be used to pinpoint the exact sequence of commands which will provoke the error. Thus, not only need no programmer effort be wasted in diagnosing whether or not the type error corresponds to a genuine bug, but the effort required to *de*bug is significantly reduced.

Underpinning the proposed research is a new foundation for type systems, called *two-sided typing*, which was introduced at POPL earlier this year by the PL and his student [7]. Two-sided type systems are a smooth generalisation of traditional type systems that incorporate the ability to *refute* type assignments. As a very simple example, in a two-sided type system with a "top" type, Ok, describing the set of all possible values, one can refute the type assignment $\mathsf{head}\,[]\;:\;\mathsf{Ok}$. Such a refutation amounts to a proof that taking the head of an empty list will never reach a value - a *guarantee* that the program must necessarily either crash or diverge.

Our aim is to demonstrate that two-sided typing can be a rich foundation that enables impactful applications in incorrectness reasoning for functional programs. To achieve this, there are significant theoretical and practical challenges to be overcome, such as the characterisation of sound principles for refutation of type assignments in the presence of complex datatypes and computational effects, and the design of efficient and scalable type inference for two-sided systems.

To give a sharper focus to this foundational work, the project culminates in the development of a type-based static analysis tool to automatically detect data-flow violations in large Erlang codebases. Such a tool would, for example, pinpoint paths in the code where private data will reach a sink, such as writing two phone numbers to the same log entry, which constitutes a violation of user privacy.

Erlang, a functional programming language for highly-available concurrent and distributed systems is an ideal testing ground to evaluate the applicability of the work. Erlang programmers already embrace a limited form of incorrectness reasoning in the widely used Dialyzer tool. Moreover, many prominent Erlang systems, such as those represented by the project partners, have stringent privacy requirements, and there is a pressing need for tooling to help with regulatory obligations[2].

[1] E.g. Pulse-X, an IL-based analyser, recently found a slew of previously undiscovered bugs in OpenSSL [4].

[2] Such as the forthcoming EU ePrivacy Directive

2 Background to the Problem

In a traditional type system, when a program is typable, then there is a guarantee that the program will be free of certain class of runtime errors. But, when a program is untypable, there are no guarantees, it could be that the program really is incorrect: it will go wrong if someone abuses it in just the right way at runtime, but it could also be that the program is correct, it's just that the type system is not expressive enough to prove the correctness property. And when a program gets rejected for being untypable, it's up the programmer to invest effort into finding out which of these two cases it is.

For familiar, mainstream type systems, this is perhaps not too onerous (though proponents of untyped programming may forcefully disagree). By and large, programmers understand enough of those type systems to be able to predict when they will succeed, or at least quickly diagnose and program around their limitations. However, every year, the research community develops new, highly sophisticated type systems that are designed for *verifying* stronger, more behavioural properties of program expressions. These systems have *tremendous potential*, but they also face the same difficulties as general program verification, namely that proofs are very difficult to construct automatically. Moreover, failed proof attempts (that is: type errors) are difficult to diagnose because the type systems are typically highly complex, and the success of automatic type inference can be bound up with the efficacy of *ad hoc* heuristics. Consequently, very few of these type systems come to have any impact in programming practice.

This problem of numerous, difficult to diagnose, false positives – that is where a program is labelled as potentially defective whilst in reality being bug-free – has recently received a lot of attention in the automated program verification community. Starting with O'Hearn's work at Facebook [6], there has been a push towards to formal systems capable reasoning about the *incorrectness* of first-order, imperative programs, that is, for proving the presence of bugs rather than their absence. These systems combine the strengths of program logics, such as compositional reasoning, abstraction and automated proof search techniques, with the utility of industry-standard approaches like testing and symbolic execution.

Unfortunately, the question of what incorrectness reasoning might look like for *higher-order programs* (e.g. functional programs) has received little attention. This is a serious problem because higher-order functions introduce a fundamental additional layer of complexity, namely that one can no longer know statically what code will actually get executed at a specific point in the program text. Two notable attempts are Zhou, Mishra, Delaware and Jagannathan's *Coverage Types* [9], which embed "must-style", underapproximate reasoning in a type system designed for checking the coverage of generators in property-based testing; and Lindahl and Sagonas' Success Types [5], which is a type-based program analysis for Erlang. However, both are rather bespoke, and lack a convincing general foundation.

In the case of [9], coverage types introduce a notion of underapproximate types, but because they do not interact well with standard function types, to every such is also introduced a dual, overapproximate version. Thus, the meaning of a typing formula $M : A$ now depends on whether the A is understood as under- or overapproximate. In the case of [5], although the program analysis is presented using constrained typing rules, it is not the case that every solution to the constraints gives rise to a valid typing and, rather, this notion is bound up with the particular strategy used by the solving algorithm to find solutions. Moreover, since it is the absence of a type assignment that implies the presence of a bug (or divergence), tracing the defect in the code can be difficult.

Consequently, how to distil and then transfer the key ideas of these systems in order to reason about other incorrectness properties, how to also incorporate *correctness* guarantees or how to extend to new programming features is unclear.

3 Basis of our Approach

In [7], the PL argued that a more satisfying answer to the above question is to extend the fundamental notion of type system to allow for the refutation as well as the affirmation of type assignments. This gives rise to the idea of *two-sided type systems*, which form the basis of our approach.

It is natural to think of a type system as a kind of proof system, whose purpose is reasoning about the behaviour of program expressions. In this view, the atomic formulas of the system are typings $M : A$, where M is a term and A a type and, in building a proof of a judgement $\Gamma \vdash M : A$, the aim is to conclude an atomic formula $M : A$ under some assumptions Γ on the types of its free variables. Therefore, built into traditional type systems, but absent from most general proof systems, is a fundamental asymmetry: although we can conclude arbitrary formulas $M : A$, we may only make assumptions on variable typing formulas $x : B$.

Two-sided type systems remove this asymmetry, allowing for assumptions on arbitrary typing formulas $N : A$. In a call-by-value setting, the meaning of a typing judgement:

$$N_1 : A_1, \ldots, N_k : A_k \vdash M : B$$

is "if N_1 evaluates to a value of type A_1 and ... and N_k evaluates to a value of type A_k, then M either diverges or evaluates to a value of type B." For example, in a CBV PCF-like language we can state that, whenever $(\lambda x.\, x)\, y$ evaluates to a numeral, then y must have been a numeral:

$$(\lambda x.\, x)\, y : \mathsf{Nat} \vdash y : \mathsf{Nat}$$

By allowing for an empty right-hand side of the judgement, two-sided typing can express that some combination of typings is inconsistent. For example, the meaning of the following judgement is "if x is a function on numerals, and taking the head of the singleton list containing x evaluates to numeral, then false":

$$x : \mathsf{Nat} \to \mathsf{Nat},\ \mathsf{head}\, [x] : \mathsf{Nat} \vdash$$

By including a type of all values (i.e. a top element of the subtype order), here called Ok, we can express that assuming that a given program expression evaluates (to any value) is inconsistent. For example, we can refute that the following program evaluates:

$$\mathsf{head}\, [] : \mathsf{Ok} \vdash$$

Whenever a judgement of shape $M : \mathsf{Ok} \vdash$ is provable, a two-sided type system guarantees that M will not reach a value at runtime - i.e. it *must* either crash or diverge. Such terms M are said to be "ill-typed" ("untypable" is reserved for those terms for which no typing is possible), and this is the basis for incorrectness reasoning.

The rules of a two-sided type system are organised as a sequent calculus whose only formulas are typings. There are *right rules*, which are the usual rules, familiar from traditional type systems, for affirming type assignments to terms. There are also *left rules*, which do not exist in traditional type systems, but can be thought of as explaining how to refute typings (although they are commonly used for extracting information from assumptions). For example, the following left rule explains how one can refute a typing for a conditional:

$$\frac{\Gamma,\, P : A \vdash \Delta \quad \Gamma,\, Q : A \vdash \Delta}{\Gamma,\, \mathsf{ifz}\, M\, \mathsf{then}\, P\, \mathsf{else}\, Q : A \vdash \Delta}$$

It explains that one way to refute that a conditional evaluates to a value of type A is to *both* refute that the 'then'-branch will evaluate to an A and refute that the 'else'-branch will evaluate to an A.

A key ingredient is a new function type $A \rightarrowtail B$ pronounced "A *only to* B". Its right rule has a premise symmetrical to that of the usual arrow:

$$\frac{\Gamma,\, M : B \vdash x : A}{\Gamma \vdash \lambda x.\, M : A \rightarrowtail B}$$

This function type internalises the notion that some property (type) of a term depends *necessarily*[3] on some property (type) of its free variable. For example, suppose we have a type $\mathsf{NEList(a)}$ of non-empty lists whose elements are uniformly of type a. Then, to obtain an a value from the head function, it is *necessary* to supply a non-empty list of a as input, i.e. $\vdash \mathsf{head} : \mathsf{NEList(a)} \rightarrowtail \mathsf{a}$. On the other hand, it is not necessary to supply a numeral to the constantly 42 function in order to obtain a numeral in return: $\nvdash \lambda x.\, 42 : \mathsf{Nat} \rightarrowtail \mathsf{Nat}$.

With this new function type, it can be explained how to refute a typing for an arbitrary function application MN:

$$\frac{\Gamma \vdash M : B \rightarrowtail A \quad \Gamma,\, N : B \vdash \Delta}{\Gamma,\, MN : A \vdash \Delta}$$

Namely, by (i) *affirming* that the operator M is a function that, to produce an A, *necessarily* requires that its formal parameter is a B, and (ii) *refuting* that the actual parameter N evaluates to a B. Using this rule in combination with function type discussed above for head, the earlier example can be justified:

$$\frac{\dfrac{}{\vdash \mathsf{head} : \mathsf{NEList(Ok)} \rightarrowtail \mathsf{Ok}} \quad \dfrac{}{[] : \mathsf{NEList(Ok)} \vdash}}{\mathsf{head}\, [] : \mathsf{Ok} \vdash}$$

In other words, to refute that head [] evaluates (i.e. reaches a value of type Ok), one can observe that for head to return a value, it is necessary that its argument is a non-empty list of values, but it is straightforward to refute this for the empty list.

4 Hypothesis and Objectives

Two-sided type systems provide a mechanism for refutation of type assignments, and through this, incorrectness reasoning. However, the theory described in the preliminary work only extends to reasoning about the (in)correctness of pure (i.e. no

[3] By contrast, the more traditional arrow type describes a *sufficient* dependency, e.g. we have $\vdash \lambda x.\, 42 : \mathsf{Nat} \to \mathsf{Nat}$.

effects beyond divergence and crashing), call-by-value functional programs with respect to a narrow spectrum of pattern-matching exhaustiveness properties [7]. We believe that a fully developed theory of two-sided type systems can act as an effective foundation more generally.

Hypothesis: Two-sided type systems can provide a rich, theoretical foundation that enables impactful applications in the automated verification of (in)correctness for higher-order programs with effects.

There are two parts to this hypothesis, foundations and application.

Foundations For the former, we mean *richness* as the extent to which the theory enjoys strong properties that show its robustness and generality. To test it, we formulate the following objective:

(O1) Development of a robust, general theory of two-sided typing with effects.

Success of this objective is measured by the strength and applicability of the theoretical results we are able to obtain.

- For the theory to be *robust*, there should be consistency theorems; natural, alternative characterisations of expressive power; and it should be precisely situated in the larger landscape of logics for incorrectness. Key markers are the development of syntactic and semantic soundness results covering both true positives and true negatives; an alternative presentation in terms of one-sided sequents and type complement; and an understanding of the relative expressive power compared with Incorrectness Logic [6], and Success Types [5].

- For the theory to be *general*, it should be able to express sound reasoning apparatus for a wide range of types and effects which are largely independent of operational details of the programming language, such as evaluation strategy. Key markers are the characterisation of refutation principles for the central type theoretic constructions such as products, sums and their dependent analogues; the support for effects including non-termination, crashing, and, in anticipation of (O2), unbounded process creation and message passing.

Applications To test the second part of the hypothesis, we work with our partners to design and implement tools based on two-sided type theory.

(O2) The design and implementation of accurate and scalable type-based static analysis tools for the detection of data-flow violations in Erlang.

The success of this objective is measured by an empirical evaluation of the artefacts produced.

- By *accurate*, we mean that data-flow violations (such as taint leak) can be identified in small but complex program fragments. We will cultivate a suite of Erlang programs that illustrate complex interactions of features which are typical of practice.

  For example, the following is a test case from Facebook's Infer tool, which has a recent extension for proving properties of Erlang programs. This program spawns a new process which propagates any tainted message it receives back to the sender (this is the content of tito_process(), code omitted), sends the new process a tainted message source(), and then waits to receive a message X back before sinking it.

  ```
  fnl_test_send6_Bad() ->
      Pid = spawn(fun() -> tito_process() end),
      Pid ! {self(), source()},
      receive X -> sink(X) end.
  ```

  Since the program ultimately sinks data from a tainted source, it represents a violation of a data-flow property. However, the Infer tool is incapable of detecting it. To discover it, as well as accurately tracking data-flow through standard, sequential control flow and function calls, a tool must also be able to reason about higher-order functions and message passing.

- By *scalable*, we mean the ability to remain efficient even when processing large input programs. To evaluate their scalability, we will adopt two approaches. During development, we will create families of synthetic benchmarks that test the efficiency of the tools according to specific parameters, such as the size of the input program, the complexity of the types that are allowed, the size of function summaries, and so on. Once satisfactory results are obtained, we will work with our partners to conduct case studies that measure efficiency when running over their own codebases and the those in the open-source community.

5 Beneficiaries and Dissemination

Realising the objectives will directly benefit (i) the programming languages research community and

(ii) industrial users of Erlang with strong privacy and security requirements.

Type systems are a large part of academic programming languages research, and two-sided type systems add a new dimension to the common understanding of what constitutes a type system. Every year the community creates new, more sophisticated type systems that extend the notion of type safety to capture more comprehensive correctness properties, but many do not have any impact beyond publication. Having a general theory of two-sided typing would allow these systems to be extended so that they can also reason about incorrectness and, in doing so, form the basis of automated tools for bug catching, which are potentially more attractive for programmers. The results generated will be published at academic conferences.

Erlang's place as one of the best languages for developing scalable soft-real time applications with high availability (e.g. telecoms, Internet communications, finance and e-commerce) means that privacy and security are important concerns. A suite of tools for detecting data-flow violations (such as taint leaks) or to prove their absence, would allow Erlang developers to gain confidence in the properties of their code and generate evidence to give to auditors. To ensure that these tools remain actively developed beyond the lifetime of the project, we will work closely with our partners and the developers (e.g. through developer-focussed conferences such as Code BEAM and Lambda Days) to build a community of contributors.

Although our focus on applications is on Erlang, we expect that the core ideas behind the program analysis tools, their inference algorithms and user experience can be easily transported to other programming languages. Our strategy will be to engage initially with the Elixir language community, which shares a lot of the Erlang ecosystem and with which we will naturally be in proximity since they attend a number of the same venues.

6 Work Programme

The project is structured into seven work packages (WP), each of which contributes to one or more of the objectives (O1) and (O2). The following should be read in conjunction with the Work Plan, which contains the milestones and deliverables.

(WP1) Two-Sided Typing and Effects

The standard approach to reasoning about effects in a type (and effect) system augments the typing judgement with an additional component, which describes a conservative approximation of the effects that would occur when evaluating the subject. However, in the general form of a two-sided system, judgements can contain more than one subject. Moreover, it is unclear what the effects associated with evaluating terms on the left-side of the judgement should mean.

The aim of the WP is to define a framework to express effects in the two-sided setting. Three applications will be used to guide the design. One is to be able to express sound principles for reasoning with call-by-name (CBN) evaluation. The two-sided systems of [7] are not generally sound for CBN, because of the implicit treatment of the non-termination effect. The second is to obtain a system supporting sound left and right rules for the complement type operator, whilst also validating a form of "well-typed programs don't go wrong'. This is not possible in *loc cit* due to the coarse treatment of the going-wrong effect. The third is to express an instance of the framework that supports the description of effects such as process creation and the sending and receiving of messages using terms from a small process calculus.

(WP2) The Role of η Rules and Extensionality

In a two-sided type system, η-rules, by which we mean the expression of extensionality principles for datatypes, such as the equivalence $x \equiv (\mathsf{fst}(x), \mathsf{snd}(x))$ for x of product type, seem to take on increased importance. For example, in [7], it is not possible to prove the following judgement, since the η-equivalence above is not available:

$$\mathsf{fst}(x) : \mathsf{Nat},\ \mathsf{snd}(x) : \mathsf{Nat} \vdash x : \mathsf{Nat} \times \mathsf{Nat}$$

The aim of this work package is to propose a standard approach to reasoning with extensionality in two-sided type systems. Of course, it is easy to add extensionality rules, but the most convenient form for these rules is not clear. Of especial interest is the interaction with type inference – the most obvious formulation will violate the principle that rule premises contain only subterms of the conclusion.

(WP3) Dependent Types and Qualifiers

Dependent types play an essential roles in many of the most successful existing systems for higher-order program verification, with Rondon, Kawaguchi and Jhala's *Liquid Types* being the most prominent example in mainstream programming [8]. Moreover, for our proposed application identifying data-flow violations in Erlang, we expect to need a degree of dependence in order to reason accurately about certain primitive datatypes, like integers and strings.

The aim of this WP is to understand how refutation interacts with dependent types by designing a two-sided version the Liquid Types system. The main challenges are to characterise a dependent analogue of the necessity function arrow, and to design the extension in such a way that the key algorithmic properties of Liquid Types, such as the reduction of typability to satisfiability modulo theories, are preserved.

(WP4) A Two-Sided Hoare Logic

The explosion of work has almost entirely been devoted to Hoare-style program logics for first-order imperative programs. We would like to understand the relationship between these logics and two-sided typing, and thereafter tap into the growing body of theoretical results and practical techniques. The aim of this WP is to define a two-sided Hoare logic.

The idea is to define the logic, in its most general form, as a sequent calculus whose formulas are Hoare triples. Following the principles as two-sided typing, *total* correctness triples are refuted on the left side, and *partial* correctness triples are affirmed on the right side. This should bring the logic into close connection with Cousot's necessary preconditions [2], the algorithmics of which were extensively studied in [1].

(WP5) A Success Type System for Core Erlang

The aim of this WP is to combine the results of (WP1)–(WP4) in a Success-style type system for Core Erlang with automatic type inference. The challenge is to find a careful balance between the expressive power (incorporating a small process calculus to denote concurrency effects (WP1) – see also (WP6), η-rules (WP2) and a degree of dependence for reasoning about primitive datatypes and pids (WP3)) and efficiency of automation (using insights from (WP3)).

Following the PL's work on highly scalable type inference for pattern-match safety in Haskell [3], we will design a compositional algorithm, in which the size of summaries (typings for top-level functions) is tightly bounded to ensure an acceptable overall time complexity. The key is to determine a bound that, to as small a degree as possible, does not exclude the possibility of proving the incorrectness of practical examples. Hence, the repository of example code cultivated in collaboration with our partners will be crucial.

(WP6) Model Checking Concurrency Effects

The system of (WP5) is intended to support the description of effects using the terms of a small process calculus appropriate to Erlang-style concurrency. However, the rules of this system are not intended to be able to reason about the terms of this calculus, merely accumulate them: the effect assigned by the system to a program is a process term which describes the creation of actors and their potential interactions when sending and receiving messages.

The aim of this WP is to design and implement algorithms for detecting possible executions of a term of this process calculus that result in a data-flow violation in the original program. Since our motivation is produce a tool that is free of false positives, our approach will be based on (context-) bounded model checking, in which all possible executions are exhaustively explored up to a threshold.

(WP7) Static Analysis Tool for Erlang Developers

The aim of this WP is to integrate the type inference engine from (WP5) and model checker of (WP6) as the backend of a static analysis tool for detecting data-flow violations (such as taint leaks) and, with our partners, to conduct case studies on its usability, accuracy and efficiency. Compared to the existing tool Dialyzer, our analyser will have sophisticated support for reasoning about data-flow, even in the presence of complex concurrency, and the basis in two-sided typing will allow it to provide detailed information about the code location of issues.

This is a large work package which will run throughout the project, and act as the basis for discussion and collaboration with the project partners. We will distinguish three milestones:

(i) Implementation of type inference for a version of the basic system from [7] for Core Erlang. This is used to establish a baseline, and to conduct performance experiments that will inform (WP5).

(ii) Extension to exactly capture inference for the system of (WP5). Evaluation of this implementation provides evidence towards the success of (WP5).

(iii) Integration into a static analysis tool for data-flow. The inference engine essentially becomes the "back-end" of a tool for data-flow queries on Erlang programs.

References

[1] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 128–148, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[2] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In R. Jhala and D. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 150–168, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] E. Jones and S. Ramsay. Intensional datatype refinement: With application to scalable verification of pattern-match safety. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[4] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.

[5] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, page 167–178, New York, NY, USA, 2006. Association for Computing Machinery.

[6] P. W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[7] S. Ramsay and C. Walpole. Ill-typed programs don't evaluate. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024.

[8] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169, 2008.

[9] Z. Zhou, A. Mishra, B. Delaware, and S. Jagannathan. Covering all the bases: Type-based verification of test input generators. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.