Short Paper: Weak Runtime-Irrelevant Typing for Security

Matthías Páll Gissurarson
pallm@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

Agustín Mista
mista@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

Types indexed with extra type-level information are a powerful tool for statically enforcing domain-specific security properties. In many cases, this extra information is runtime-irrelevant, and so it can be completely erased at compile-time without degrading the performance of the compiled code. In practice, however, the added bureaucracy often disrupts the development process, as programmers must completely adhere to new complex constraints in order to even compile their code.

In this work we present WRIT, a plugin for the GHC Haskell compiler that relaxes the type checking process in the presence of runtime-irrelevant constraints. In particular, WRIT can automatically coerce between runtime equivalent types, allowing users to run programs even in the presence of some classes of type errors. This allows us to gradually secure our code while still being able to compile at each step, separating security concerns from functional correctness.

Moreover, we present a novel way to specify which types should be considered equivalent for the purpose of allowing the program to run, how ambiguity at the type level should be resolved and which constraints can be safely ignored and turned into warnings.

CCS CONCEPTS

• **Software and its engineering** → **Functional languages**; **Constraints**; **Domain specific languages**.

KEYWORDS

GHC; Type Checking; Compilers; Haskell

ACM Reference Format:
Matthías Páll Gissurarson and Agustín Mista. 2020. Short Paper: Weak Runtime-Irrelevant Typing for Security. In *15th Workshop on Programming Languages and Analysis for Security (PLAS'20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3411506.3417595

PLAS'20, November 13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8092-8/20/11…$15.00
https://doi.org/10.1145/3411506.3417595

1 PROGRAMMING WITH TYPE CONSTRAINTS

Enforcing domain-specific properties is a complicated task that developers are forced to carefully address when designing complex systems. In the functional programming realm, strongly-typed languages like Haskell are an advantage since *one can use the type system to enforce domain-specific constraints*! However, this technique is not without flaws. To illustrate some of the issues with this technique, suppose we are writing a library for information-flow control over labeled pure values – loosely inspired by the `MAC` library by Russo [13]. For simplicity, we assume that the only labels are `L` for public and `H` for secret data. Then, we can use *phantom* types [3, 8] to label arbitrary data with security labels:

```
data Label = L | H

newtype Labeled (l :: Label) a = Labeled a
```

As an example, the value `Labeled 42 :: Labeled L Int` represents a public integer, whereas `Labeled "1234" :: Labeled H String` represents a secret string. It is important to note that in Haskell, **`newtype`**s are representationally equal to the type they wrap, meaning that the runtime representation of `Labeled 42` is the same as the one for `42`. Later, labeled values can be combined according to different security policies using type constraints [6, 7], as an example, we can enforce that no information flows from `H` to `L` by defining the empty type class:

```
class ((l :: Label) <= (l' :: Label))
```

and defining instances of `(<=)` only for the flows we allow:

```
instance (L <= L)
instance (L <= H)
instance (H <= H)
```

Since there is no instance for the forbidden flow `H <= L`, any code that triggers the constraint `H <= L` during compilation will produce a type error. Note that the class `(<=)` has no methods, so it is represented by a computationally-irrelevant empty dictionary at runtime.

We can now use `(<=)` to implement combinators over labeled values that ensure that secrets do not leak into public data, e.g. the familiar **`zip`** combinator can be given the type:

```
zip :: (x <= z, y <= z) => Labeled x [a]
                        -> Labeled y [b]
                        -> Labeled z [(a, b)]
```

where `(x <= z, y <= z)` ensures that the label `z` of the output is greater or equal to both its inputs. Then, the definition:

```
bad :: Labeled L [(Usr, Pwd)]
bad = zip (Labeled [11111, 222222] :: Labeled L [Usr])
          (Labeled ["hun", "ter2"] :: Labeled H [Pwd])
```

will be rejected by GHC with a generic error indicating that we are missing a type class instance for the forbidden flow:

```
error: No instance for H <= L (...)
```

and indeed, we can see that there is a leak from the secret passwords in the list `["hun","ter2"]` to the public list `[(11111,"hun"),(222222,"ter2")]`. Ouch!

As shown so far, we can use Haskell's type system to accommodate domain-specific constraints about security labels using phantom types and type classes. Although this is a powerful strategy when it comes to writing domain-specific libraries [1, 9, 12, 16], it can be hard to use in practice:

- The code cannot be run unless it is provably secure, preventing users from testing the functional correctness of the program separately from its security properties.
- Users must tag all their data with an explicit `Label`, and cannot use features such as pattern matching without explicitly unwrapping and rewrapping the labels.
- Moreover, they need to tag both the secret and the public data, even though there might exist a sane default tag.
- The type errors are too general and hard to understand for users unfamiliar with Haskell's type system, and;
- Synthesizing type based suggestions [5] becomes harder, due to domain-specific constraints and ambiguous types.

2 WEAKENING RUNTIME-IRRELEVANT TYPING

In GHC, type checking is based on constraint-based type inference. Albeit intricate in practice, the algorithm works by traversing the code to accumulate a set of type constraints (defined as part of the type system specification) and then invokes the constraint solver to solve those constraints [19]. In the latest GHC, constraints come in three main flavours [17]:

- *Givens* from type signatures, for which we have evidence,
- *Wanteds* from expressions, for which we want evidence,
- *Deriveds*, which are constraints that any solution must satisfy but we do not require evidence of (e.g. equalities arising from functional dependencies and superclasses).

The constraint solver solves the wanteds with respect to the givens and the typing rules of GHC (which include creating and unifying type variables), making sure that the solution satisfies the deriveds [17, 19]. This process is capable of type checking complex programs, but isn't perfect when it comes to domain-specific constraints like `(<=)`.

Luckily, the type checker can be extended with plugins to handle additional type checking rules, for example to simplify naturals or invoking an SMT solver [4, 10]. Type checker plugins are invoked by the compiler in order to *a*) simplify givens, where a plugin might find a contradiction, and, *b*) whenever there are unsolved constraints that the type checker could not solve.

For the purpose of weakening the type checking of runtime-irrelevant types, we developed WRIT,[1] a plugin that extends GHC's type system by adding the rules seen in Figure 1 for when type checking would not be able to proceed otherwise. Users of the plugin can selectively apply these rules to runtime-irrelevant constraints and equalities by writing instances of the `Ignore`, `Discharge`, `Promote`, and `Default` type families [14, 20] as described in the rest of this section.

2.1 Ignoring Runtime-Irrelevant Constraints

In Haskell, users can define empty typeclasses that have no methods (like `(<=)`), which represent runtime-irrelevant constraints. However, we would like to be able to turn these constraints into compile time warnings, so that functional correctness of the program can be verified separately from its security. The IGNORE rule applies whenever there is an unsolved empty typeclass constraint with an instance of the `Ignore` family:

```
type family Ignore (c :: Constraint) :: Message
```

By defining an instance of the `Ignore` family for `(<=)`:

```
type instance Ignore (H <= L) =
  Msg (Text "Found forbidden flow from H to L!")
```

Users can specify that the constraint `H <= L` can be ignored with the message shown above. With this instance in scope and WRIT enabled, the error for the **bad** function defined earlier will be turned into the following warning:

```
warning: Found forbidden flow from H to L!
```

2.2 Discharging Runtime-Irrelevant Equalities

With runtime-irrelevant types, we often want to ignore nominal equalties of the form **a** ∼ **b**, which are specially handled GHC primitives. As an example, we might want to turn `L ~ H` into a warning when compiling insecure programs. The DISCHARGE rule applies to unsolved equalities of the form **a** ∼ **b**, for which there is an instance of the `Discharge` family for **a** and **b**:

```
type family Discharge (a :: k) (b :: k) :: Message
```

By defining an instance of `Discharge` for `L` and `H`:

```
type instance Discharge L H =
  Msg (Text "Using a public L as a secret H!")
```

Users can allow `L ~ H` with the message shown above. This in conjunction with ignoring `H <= L` effectively negates any guarantees that our library provides.

2.3 Promoting Representationally-Equivalent Types

A special case of discharging is when **a** and **b** have the kind `(*)`, the kind of base types in Haskell. Discharging the equality **a** ∼ **b** effectively *promotes* **a** to **b**, meaning that **a** is treated as a **b**. This is only runtime-irrelevant when **a** and **b** have the same runtime representation, making **a** ∼ **b** runtime-irrelevant. This coincides with the `Coercible` constraint in GHC [2], so to handle this common case we define `Promote`:

```
type family Promote (a :: *) (b :: *) :: Message
```

[1] The WRIT plugin is available at https://github.com/tritlo/writ-plugin.

And define an instance of `Discharge` for types of kind `(*)`:

```
type instance Discharge (a :: *) (b :: *) =
  OnlyIf (Coercible a b) (Promote a b)
```

Then, by defining an instance of `Promote` for labeled values:

```
type instance Promote a (Labeled l a) =
  Msg (Text "Promoting unlabeled " :<>: ShowType a
       :<>: Text " to " :<>: ShowType (Labeled l a))
```

Users can use any base type `a` (like `Int`) as a `Labeled l a`, where `l` is either `L` or `H`, e.g., it becomes possible to write: `[1,2] :: Labeled L [Int]`, where `[1,2]` is promoted and treated as a public `[Int]`.

2.4 Defaulting Runtime-Irrelevant Type Variables

When programming using runtime-irrelevant types, it frequently occurs that the type of a phantom type variable cannot be inferred. However, it is often the case that there is a "sane" value to choose when there are no restrictions, such as the label `L` for labeled data. The Default rule applies whenever there is an unsolved constraint with a free type variable of kind k for which there is an instance of the `Default` family:

```
type family Default k :: k
```

By defining an instance of the `Default` family for `Label`:

```
type instance Default Label = L
```

Users can specify that any free type variables of kind `Label` in an unsolved constraint should be set to `L`.

Now With Less Cruft! After defining the instances as shown above, WRIT can use them to weaken our library's domain-specific constraints. Users can then easily express and run (possibly insecure) programs operating on labeled values as if they had the underlying type without overhead:

```
labeledOr :: Labeled L [Bool] -> Labeled L Bool
labeledOr (x:xs) = if x then True else labeledOr xs
labeledOr _      = True
```

2.5 Ensuring Runtime-Irrelevance

Since it is not always safe to ignore or discharge, we allow users to recover some safety by using the `OnlyIf` constructor, as used above in the `Discharge` instance for `(*)` to assert `Coercible`. The OnlyIf rule is used to unravel $\mathrm{F}\,a_1 \ldots a_n \sim \mathrm{Msg}\,m_b$' when $\mathrm{F}\,a_1 \ldots a_n$ reduces to an OnlyIf $c\;m_a$, and adds the additional constraints c and $m_a \sim m_b$ as obligations. This eventually results in an equality of the form $\mathrm{Msg}\,m_a \sim \mathrm{Msg}\,m_b$, causing GHC to unify m_b with m_a, inferring the message to be emitted. Note that OnlyIf $a\;b$ only holds if both a and b hold, and b is only emitted if a holds.

2.6 Turning Type-Errors into Warnings

To model the fact that we often want to turn type-errors into warnings, all our rules produce a set of messages, M, which is a union of the messages produced by any obligations. The Discharge and Ignore rule add a user defined message to the set, whereas the Default rule adds a standardized message. The user defined messages are built using GHC's user type-error combinators, which allows them to use type families to compute the message [15]. The resulting set of messages is reported as warnings at the end of type checking, or alternatively as type-errors, if the user passes the plugin the `keep-errors` flag.

3 IMPLEMENTATION

WRIT operates by examining the wanted and derived constraints passed to the plugin by GHC. Messages are handled as a set of logs with type variables for the messages and their origin. The logs are finalized before they are output, with the type variables representing messages are replaced with the messages themselves.

The plugin applies the Default rule by generating constraints of the form `a ~ Default k` for any free type variable `a` of kind `k` in unsolved constraints, Then, e.g. `Default Label` will reduce to `L`, and the variable `a` is set to `L` in the context. In Haskell there are two types of type variables, rigid and flexible. Rigid type variables are variables mentioned in the givens, i.e. the constraints. Flexible type variables are type variables instantiated from a `forall`. For example, in `return :: Monad m => a -> m a`, `m` is a rigid type variable, while `a` is flexible type variable. When we default a type variable, we must distinguish between rigid and flexible type variables: for rigid type variables, the generated constraints take the form of a given, with assertion from WRIT that `a` is equivalent to `Default Label` as the evidence. For flexible type variables, we do not require evidence, so it suffices to emit a derived to unify `a` with `Default Label`.

For the Ignore rule, the plugin asserts that the constraint holds, which corresponds to the empty typeclass having an instance. It also emits a constraint that applying the `Ignore` family to the constraint results in a message wrapped in the `Msg` constructor, and adds it to the set of messages as a new type variable that will unify with the message itself.

Similarly for the Discharge rule, WRIT generates a proof by assertion that `a ~ b` holds (e.g. `L ~ H`), and adds the obligation that Discharge $a\;b$ reduces to a Msg m, with the fresh flexible type variable m added to the set of messages M. The evidence is an assertion in the form of a zero-cost coercion [2], which is safe for runtime-irrelevant types which have the same runtime representation.

WRIT applies the OnlyIf rule by generating an assertion that OnlyIf $c\;m_a \sim m_b$ and checking that both c and $m_a \sim m_b$ hold. As an optimization, we solve equalities of the form:

$$\mathrm{OnlyIf}\;c_1\;(\mathrm{OnlyIf}\;c_2\;(\ldots(\mathrm{OnlyIf}\;c_n\;\mathrm{Msg}\;m_a))) \sim \mathrm{Msg}\;m_b$$

by checking all the constraints $c_1, \ldots, c_n$ and $\mathrm{Msg}\,m_a \sim \mathrm{Msg}\,m_b$, causing GHC to unify m_a with m_b.

4 CONCLUSIONS AND FUTURE WORK

We presented WRIT, a type checker plugin for GHC to weaken the type checking process for runtime-irrelevant constraints and representationally-equivalent types. We believe our work will facilitate developers to adopt more secure programming practices in Haskell with less overhead, since it is now possible to start doing so in a more gradual manner. As this is a work in progress, there are a few avenues for future work:

Safety. The WRIT plugin gives users a lot of freedom and allows them to override the typing rules used in Haskell. We have yet to investigate which rules can be safely defined by the user, what can

$$\frac{\Gamma, \text{Default}\ k, a \sim \text{Default}\ k \vdash c : \text{Constraint},\ M}{\Gamma, a : k \in \text{FV}(c), \text{Default}\ k \vdash c : \text{Constraint},\ M \cup \{m_{\text{def}}\}}\ \textsc{Default} \qquad \frac{\Gamma, \text{Ignore}\ c \vdash \text{Ignore}\ c \sim \text{Msg}\ m,\ M}{\Gamma, \text{Ignore}\ c \vdash c : \text{Constraint},\ M \cup \{m\}}\ \textsc{Ignore}$$

$$\frac{\Gamma, \text{Discharge}\ a\ b \vdash \text{Discharge}\ a\ b \sim \text{Msg}\ m,\ M}{\Gamma, \text{Discharge}\ a\ b \vdash a \sim b,\ M \cup \{m\}}\ \textsc{Discharge} \qquad \frac{\Gamma \vdash c : \text{Constraint},\ M_c \qquad \Gamma \vdash m_a \sim m_b,\ M}{\Gamma \vdash \text{OnlyIf}\ c\ m_a \sim m_b,\ M_c \cup M}\ \textsc{OnlyIf}$$

Figure 1: The typing rules that WRIT extends GHC's type system with. The judgement $\Gamma, \text{F}\,a_1 \ldots a_n \vdash c, M$ here judges that with an instance $\text{F}\,a_1 \ldots a_n$ in the context the constraint (or equality) c holds with the set of output messages M. Here, we write $c : \text{Constraint}$ to denote a well formed constraint c, and m_{def} is a compiler generated message based on the source expression.

go wrong if they define an invalid rule, and whether we can prevent users from defining such rules.

Overlaps. Neither the compiler plugin nor the formalization deal with what happens when the user defined instances overlap, which can cause the typing rules of WRIT to overlap and it is unclear which one to choose. In the plugin itself, this is handled by preferring Discharge to Ignore and Ignore to Default. It is clear however that the choice should not affect the semantics of the compiled program (something yet to be proven), but which typing rule is preferred can affect the errors or warnings emitted in the process. One possibility is to design a heuristic that selects the most specific typing rule applicable, to emit more concrete (and useful) messages, as opposed to more generic ones.

Dynamic and Gradual Typing. We want to investigate how relaxing the type checking process could interact with Haskell's dynamic typing capabilities [11]. Whenever the type checker finds two expression producing a type mismatch error, it might be possible to promote them both to Haskell's dynamic representation, `Dynamic`. In this light, the invalid list expression `[42, "hello"]` could be promoted to a list of dynamic values by promoting both `42` and `"hello"` to a unified dynamic representation, i.e. `[42, "hello"] :: [Dynamic]`. Then, dynamically typed values could be demoted to concrete types via runtime checks inserted automatically. This mechanism could shorten the gap between Haskell, a strongly typed language, and dynamically typed languages like Python or Erlang by simply toggling a compiler plugin, enabling us to do module-based *gradual* typing [18].

ACKNOWLEDGMENTS

Special thanks to Claudia Cauli, Alejandro Russo, David Sands, and Andrei Sabelfeld for their feedback. This work was partially supported by both the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation; and the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023).

REFERENCES

[1] Jan Bracker and Andy Gill. 2014. Sunroof: A Monadic DSL for Generating JavaScript. In *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324* (San Diego, CA, USA) *(PADL 2014)*. Springer-Verlag, Berlin, Heidelberg, 65–80. https://doi.org/10.1007/978-3-319-04132-2_5

[2] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2014. Safe Zero-Cost Coercions for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 189–202. https://doi.org/10.1145/2628136.2628141

[3] James Cheney and Ralf Hinze. 2003. *Phantom types.* Technical Report. Cornell University.

[4] Iavor S. Diatchki. 2015. Improving Haskell Types with SMT. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2804302.2804307

[5] Matthías Páll Gissurarson. 2018. Suggesting Valid Hole Fits for Typed-Holes (Experience Report). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 179–185. https://doi.org/10.1145/3242744.3242760

[6] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996), 109–138. https://doi.org/10.1145/227699.227700

[7] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell workshop*. 1–16.

[8] Daan Leijen and Erik Meijer. 2000. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages* (Austin, Texas, USA) *(DSL '99)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/331960.331977

[9] Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell* (Baltimore, Maryland, USA) *(Haskell '10)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/1863523.1863533

[10] Divesh Otwani and Richard A. Eisenberg. 2018. The Thoralf Plugin: For Your Fancy Type Needs. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 106–118. https://doi.org/10.1145/3242744.3242754

[11] John Peterson. 1993. *Dynamic typing in Haskell.* Technical Report. Technical Report YALEU/DCS/RR-1022, Yale University, Department of Computer Science.

[12] Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1411286.1411290

[13] Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP '15)*. Association for Computing Machinery, New York, NY, USA, 280–288. https://doi.org/10.1145/2784731.2784756

[14] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type Checking with Open Type Functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/1411204.1411215

[15] Alejandro Serrano and Jurriaan Hage. 2017. Type Error Customization in GHC: Controlling Expression-Level Type Errors by Type-Level Programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages* (Bristol, United Kingdom) *(IFL 2017)*. Association for Computing Machinery, New York, NY, USA, Article 2, 15 pages. https://doi.org/10.1145/3205368.3205370

[16] Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. 2016. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming* 27 (2016), e5. https://doi.org/10.1017/S0956796816000241

[17] GHC Team. 2020. *The ghc-8.10.1 library Constraint module.* Retrieved August 18, 2020 from https://hackage.haskell.org/package/ghc-8.10.1/docs/src/Constraint.html

[18] Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4, Article 16 (Dec. 2018), 55 pages. https://doi.org/10.1145/3229061

[19] Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. 2011. Outsidein(x) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4–5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

[20] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. Association for Computing Machinery, New York, NY, USA, 53–66.