Toward a Typed Intermediate Language for R

Mickaël Laurent
Charles University, Prague, Czech Republic

Jakob Hain
Purdue University, West Lafayette, IN, USA

Filip Krikava
Czech Technical University, Prague, Czech Republic

Sebastián Krynski
Czech Technical University, Prague, Czech Republic

Jan Vitek
Northeastern University, Boston, MA, USA

Abstract

Compilers for dynamic languages often rely on intermediate representations with explicit type annotations to facilitate writing program transformations. This paper documents the design of a new typed intermediate representation for a just-in-time compiler for the R programming language called FIŘ. Type annotations, in FIŘ, capture properties such as sharing, the potential for effects, and compiler speculations. In this extended abstract, we focus on the sharing properties that may be used to optimize away some copies of values.

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers

Keywords and phrases JIT, compilation, static typing, ownership, copy-on-write, dynamic language

Digital Object Identifier 10.4230/OASIcs.Programming.2025.24

Category Extended Abstract

Funding This work was supported by the Czech Ministry of Education, Youth and Sports under program ERC-CZ, grant agreement LL2325, as well as by the Czech Science Foundation Grant No. 23-07580X.

Acknowledgements We would like to thank the reviewers and organizers of the VIMPL workshop.

1 Background

R is a dynamic language with lazy evaluation, first-class environments, and copy-on-write vectors, posing unique optimization challenges. FIŘ is a minimalistic calculus inspired by core languages, such as Featherweight Java [4] or Go [3], specifically designed to model and optimize the particular language features of R. FIŘ is meant to be the foundation of an intermediate representation for a just-in-time compiler. Its design enables and facilitates optimizations and its static type system can be used to improve confidence in their correctness. The key features of FIŘ, and the corresponding R features they are designed for, are contextual dispatch [1] and static types to elide runtime checks and effect inference; promise types for laziness; effects to disprove reflection; and explicit ownership to remove redundant copies and reference-count checks.

This paper focuses on ownership. Laziness will be only described to explain ownership, and contextual dispatch and effects will not be described further.

2 Ownership Overview

In R, shared vectors are implicitly copied before mutation to ensure value independence. FIŘ makes all copies explicit and allows mutation only when vectors are guaranteed unique. Every vector in FIŘ has a *ownership*: `fresh`, `owned`, `borrowed`, or `shared`. Only fresh and owned vectors are guaranteed unique. The borrowed vectors exist only in function parameters and remain unique after the call. Vectors returned by lazy promises, accessible by reflection, or with unknown aliases are shared. They can never be guaranteed unique and must be duplicated to become fresh or owned.

A vector is fresh if it is not referenced by any variable (*e.g.*, a newly-allocated vector). A fresh vector can be assigned to an owned or shared variable. An owned vector cannot be assigned to a variable, otherwise it would compromise uniqueness as it could be then assigned to multiple live variables. Effectively, fresh vectors have a reference count of zero, and owned vectors of at most one. A vector can be made fresh by copying. Furthermore, a vector in an owned register variable can become fresh by "using" the register; the ownership analysis ensures a register cannot be accessed after it is "use"d.

```
v1 : owned = [1,2,3]   # [1,2,3] allocates a fresh vector.
                       # It becomes owned when assigned to v1.
v2 : owned = v1        # Not allowed.
v3 : owned = use v1    # Allowed, v1 cannot be accessed anymore.
```

There are further restrictions on owned vectors. First, an owned vector cannot be passed to an owned parameter. An owned parameter may be "used", making the vector fresh. Second, it cannot be passed to a shared parameter. A vector in a shared parameter may persist after the call ends, thus possibly loosing its uniqueness. On the other hand, an owned vector can be passed to a borrowed parameter because borrowed vectors are explicitly guaranteed *not* to persist after calls, so the uniqueness is preserved (although the vector will not be unique during the call, FIŘ has no concurrency and borrowed vectors are read-only, so it cannot be mutated either).

```
f : (borrowed) -> shared = fun(v:borrowed) { print(v) }
g : (owned) -> shared  = fun(v:owned) { v[1] <- 42 ; print(v) }
v : owned = [1,2,3]
f(v)                  # this is allowed as v is still unique after the call
g(v)                  # this is not allowed (g may mutate or leak v)
g(use v)              # this is allowed, but v cannot be accessed anymore
```

In R, it is possible to access variables from other scopes using reflection. This means that variables that are subject to reflection cannot be considered owned. In FIŘ, we distinguish between *named variables*, which may be accessed through reflection, and *register variables*, which cannot. Named variables can be optimized into registers if the type system guarantees they are not subject to reflection. To that end, we need to do effect tracking.

3 Ownership Analysis

Ownerships in FIŘ are checked with a flow-insensitive type analysis and a flow-sensitive action analysis. The type analysis checks that vectors are not used in violation of their ownership annotations; for example, that a shared vector is not assigned to an owned variable, or a borrowed argument is not passed to an owned parameter. The action analysis checks that every variable is initialized before its first access and not accessed after a "use".

The action analysis is implemented by computing the *action* of every expression, bottom-up. An action describes what registers the expression may interact with; those it may read before assigning, those it may assign, those it may “use”, and those it may *capture*. To understand the latter, consider the following example:

```
v : owned = [1,2,3]
p : shared = prom{ print(v) ; 42 } # Creates a promise (delayed computation)
g(use v)  # g is a function that takes an owned or shared parameter
print(p)  # Calling print on the promise p will trigger the evaluation of p
```

Promises, like zero-argument closures in other languages, are used to model laziness: in R, arguments passed to functions are not evaluated right away, but their evaluation may be triggered later when they are accessed. The code above defines a promise `p` that prints the vector in `v`, and then returns the constant `42`. Even though, at the moment of its creation, the promise `p` is allowed to access `v`, it is not true anymore when `p` is actually evaluated (when calling `print(p)`) because the register `v` has been used in the meantime. It is not always possible to statically determine when a promise will be forced, and this is why our action analysis considers that the registers captured by a promise indefinitely live, and thus disallows any future `use` of these registers.

The action analysis passes if no action computations reveal an access after use, a use of a captured register, and if the function body’s action has no “read before assignment” registers except parameters.

4 Related work

We have previously designed and implemented an IR for R: PIR [2]. PIR is not a minimal calculus but a full implementation, which models a large subset of R and has been deployed in a real compiler, Ř . PIR has contextual dispatch, types, and effects, but no ownership. We believe that, because we have implemented PIR’s type system while designing it, it has more edge-cases and complexity than necessary; even as we extend FIŘ’s type system to support as much as PIR, we hope to keep it simpler and more intuitive. Also, PIR’s type system is more naive and does not offer type safety guarantees, while FIŘ’s type system does.

Our analysis is loosely inspired by Rust’s borrow checker, which has been implemented in MIR [6] and formalized in *RustBelt* [5] and *Oxide* [7]. Rust’s owned values are analogous to FIŘ’s owned and fresh values, in that both are guaranteed unique. Rust’s immutable borrows are analogous to FIŘ’s borrowed values, in that they are temporary, and can store data from owned variables which remain owned after they expire. Rust’s *Rc* “smart pointers” are analogous to FIŘ’s shared values, in that they have no enforced liveness and any value assigned to them can never be mutated after. Rust’s mutable borrows have no FIŘ analogue, because they would never be generated, since functions in R cannot mutate vector parameters (FIŘ’s owned parameters are generated when the compiler discovers a vector that, in GNU-R, would be copied then discarded; they are not a feature of R but an optimization, however we have no optimization requiring an analogue to Rust’s mutable borrow).

Scoping and control flow is more complex in R than in Rust, due to R’s lazy semantics and first-class mutable environments. Thus we introduce an explicit `use` instruction, and define a flow-sensitive analysis to detect potential “read before assignment” and “read after use” faults.

References

1 Olivier Flückiger, Guido Chair, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. Contextual dispatch for function specialization. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020. doi:10.1145/3428288.

2 Olivier Flückiger, Guido Chari, Jan Jecmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. R melts brains: an IR for first-class environments and lazy effectful arguments. In *International Symposium on Dynamic Languages (DLS)*, 2019. doi:10.1145/3359619.3359744.

3 Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight go. *ACM Trans. Program. Lang. Syst.*, 2020. doi:10.1145/3428217.

4 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 2001. doi:10.1145/503502.503505.

5 Ralf Jung. *Understanding and evolving the Rust programming language.* Doctoral Thesis, Saarländische Universitäts- und Landesbibliothek, 2020. Accepted: 2020-09-09T07:57:28Z. doi:10.22028/D291-31946.

6 Niko Matsakis. Introducing MIR, April 2016. URL: https://blog.rust-lang.org/2016/04/19/MIR/.

7 Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The Essence of Rust, October 2021. arXiv:1903.00982 [cs]. doi:10.48550/arXiv.1903.00982.