A Typed Intermediate Representation for Dynamic Languages

MICKAËL LAURENT, Charles University, Czech Republic
JAKOB HAIN, Purdue University, United States
FILIP KRIKAVA, Czech Technical University, Czech Republic
SEBASTIÁN KRYNSKI, Czech Technical University, Czech Republic
JAN VITEK, Czech Technical University, Czech Republic

Dynamic programming languages pose significant challenges for optimizing compilers due to features such as dynamic typing, late binding, reflection, copy-on-write, and delayed evaluation. To generate efficient code, compilers must speculate on which dynamic features will be exercised and produce specialized code based on these assumptions. This paper presents the design of a statically typed, high-level intermediate representation that makes dynamic behaviors explicit and amenable to static analysis. Our IR combines gradual typing with ownership tracking, and explicitly represents promises, multiple function versions, and contextual dispatch. Together, these features directly support optimizations such as specialization, inlining, scope elision, and copy elimination. We formalize a core calculus, called FIŘ, that captures the essential features required for these optimizations. We provide an operational semantics, a type system, and flow and reflection analyses, and we prove the soundness of the type system.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**.

Additional Key Words and Phrases: JIT, compilation, gradual typing, dynamic language

1 Introduction

Optimizing compilers typically use multiple intermediate representations (IRs) to organize the compilation process into modular stages. Each IR highlights specific aspects of programs to enable targeted optimization passes. High-level IRs remain close to the source language, facilitating transformations guided by source-level semantics and supporting the verification of language-level properties and invariants. In contrast, low-level IRs are closer to the target machine, exposing control flow and resource management details required for optimizations such as instruction scheduling and register allocation. This paper focuses on the design of a typed, high-level IR tailored to the needs of just-in-time compilation of dynamic programming languages.

Dynamic languages—such as Python, PHP, or Julia—pose significant challenges for traditional compilation techniques. Features like dynamic typing, late binding, and reflection often prevent the compiler from reliably inferring program behavior through source code analysis alone.

`(a) -> a + 1`	`lambda a: a + 1`	`(lambda (a)(+ a 1))`	`function(a)a + 1`
JavaScript	Python	Scheme	R

Fig. 1. Implementations of "add 1" in four dynamic languages

Consider the archetypal function that takes its argument, adds 1, and returns it (Fig. 1). Even this function is not simple in a dynamic language, and generating efficient code for it is not easy. Though the intended use may be to add 1 to a number, in JavaScript, if the argument `a` is a string, the function actually performs string concatenation. In Python, the `+` operator may be redefined, making the function run arbitrary code. In Scheme, the `lambda` macro may be overridden, changing

Authors' Contact Information: Mickaël Laurent, mickael.laurent@matfyz.cuni.cz, Charles University, Prague, Czech Republic; Jakob Hain, jakobeha@fastmail.com, Purdue University, West Lafayette, Indiana, United States; Filip Krikava, filip.krikava@fit.cvut.cz, Czech Technical University, Prague, Czech Republic; Sebastián Krynski, skrynski@gmail.com, Czech Technical University, Prague, Czech Republic; Jan Vitek, vitekj@me.com, Czech Technical University, Prague, Czech Republic.

the entire expression. In R, not only can `+` and `function` be redefined before the expression is evaluated, but `a` may be a delayed thunk that runs arbitrary code as the function is being called.

This example illustrates that in dynamic languages, even simple code can elude static reasoning, rendering most standard compiler optimizations unsound or inapplicable. The key insight is that a function's behavior may depend on non-local properties of the program—such as whether operators have been redefined or whether functions perform reflection.

Building on this observation, just-in-time (JIT) compilers adopt a speculative strategy: they use observations of past runtime behavior (*feedback*) to predict future behavior, then generate code optimized for the predictions. The predictions for dynamic languages effectively replace the guarantees of static languages: for example, predicting that a variable will always hold a value of a particular runtime type, or that a block of code will not perform reflection. Optimizations that rely on those predictions include, respectively, scalar unboxing and code motion. Since predictions are not guarantees, the compiler inserts before any optimizations that rely on them *guards*—runtime checks that, on failure, divert control flow to the original unoptimized code.

In a compiler, design of the high-level IR is critical, as it determines which optimizations can be expressed and verified. A representative example is the Julia compiler, as described in [3]. According to the authors, the two most impactful optimizations are call inlining and function specialization, as these enable nearly all subsequent optimizations. Accordingly, the compiler is structured to apply these early transformations on a high-level IR, after which lower-level optimizations are delegated to LLVM, a separate compiler with its own IR.

What, then, should a high-level IR for a dynamic language include? As explained above, dynamism renders most optimizations impossible, but a compiler can use guarded predictions to create regions where it is guaranteed absent. The IR must make dynamism—and its absense—explicit, to enable said regions to be expressed, thus enabling optimizations typical for a static compiler. The precise set of features varies by language, but the goal is the same: convert properties that are resolved at runtime into ones that can be reasoned about statically.

Our thesis is that these properties can—and should—be captured in a static type system. Doing so serves two purposes: it enables compile-time verification that optimizations are applied correctly, and it provides a formal specification for compiler writers.

Our work builds on prior results from the development of the Ř compiler for the R programming language and its intermediate representation, PIR. We introduce FIŘ, a typed intermediate representation designed to support dynamic typing, runtime code loading, delayed evaluation, copy-on-write semantics, and unrestricted reflection.

The key features of FIŘ (Section 4) address these features at two levels—restricting sources of dynamism and enabling targeted optimizations:

- **Variables:** FIŘ distinguishes *named* from *register* variables; the former can be updated non-locally through reflection, while the latter are shielded from it.
- **Types:** Inspired by gradual typing, FIŘ supports a stratified type system in which variables range from statically typed to entirely dynamic.
- **Ownership:** A limited form of ownership tracks value lifetimes and transfer semantics, enabling optimization of copy-on-write behavior.
- **Specialization and Dispatch:** Functions can be compiled into multiple specialized versions; calls are dispatched to an applicable version at runtime or bound statically.
- **Inlining:** Function inlining exposes optimization opportunities while preserving the scopes of named variables.
- **Promises:** Delayed computations are represented as promises, supporting optimizations that target lazy evaluation.

Although motivated by R, these features are common across dynamic languages (Section 2), and we argue that FIŘ's design is broadly applicable. Moreover, while designed for a JIT compiler, none of its features depend on runtime feedback, so it can equally serve ahead-of-time compilers.

FIŘ is presented as a minimalistic calculus, inspired by Featherweight Java [15]. Validity is guaranteed by a combination of type checking, to ensure that variables and functions are used according to their declarations, and flow analysis, to ensure that variables are initialized before use (Section 5.1). FIŘ's reduced complexity enables us to provide a formal operational semantics (Section 5.2) and to prove the soundness of its type system (Section 5.3).

2 Dynamic languages

Dynamic languages are characterized by idiosyncratic designs; the particular set of features supported in any given language is driven by the needs of the application domain and the whims of the language designer. Table 1 describes which features are provided by six popular languages.

Feature	Julia	JavaScript	Python	PHP	Clojure	R
Dynamic typing	✓	✓	✓	✓	✓	✓
Code loading	✓	✓	✓	✓	✓	✓
Late binding	✓	✓	✓	✓	✓	✓
Reflection	✓	✓	✓	✓	✓	✓
Copy-on-write	—	—	—	✓	—	✓
Delayed evaluation	—	—	—	—	✓	✓

Table 1. Support for dynamic language features across six popular languages

Dynamic Typing. In dynamically typed languages, variables do not have fixed types; instead, type information is associated with values. As a result, a variable can be rebound to values of different types over the course of execution. This flexibility is a hallmark of all the languages discussed above.

Code Loading. In these languages, all code outside of the standard library is loaded during (as opposed to before) execution. New global variables and functions may be added, and most existing ones may be rebound or modified. Many contain "eval", a function that takes an arbitrary string as input, and parses and then evaluates it.

Late Binding. Late binding refers to the deferral of method resolution until runtime, when the types (or values) of arguments are known. In object-oriented languages, this is typically achieved through dynamic method dispatch; in functional languages, through higher-order functions. Additionally, many allow built-in functions—such as + —to be redefined or overloaded.

Reflection. Reflection is the ability of a program to inspect and modify its own structure and behavior at runtime. All the languages discussed support reflection to varying degrees. Julia is the most restricted, as it allows introspection but not reflective updates. R, on the other hand, is the most permissive—it supports call stack introspection and even allows local variables to be deleted dynamically.

Copy-on-Write. Copy-on-write is a memory management optimization in which multiple references share the same data until one of them attempts a modification, at which point a copy is made. This technique can be used to simulate referential transparency—for example, allowing functions to update arguments internally without those changes being externally visible.

Delayed Evaluation. In R, function arguments are lazily evaluated by default. When a function is called, its arguments are not immediately computed but instead represented as promises—deferred computations that are evaluated only when their values are needed. Clojure supports a similar, though explicit, mechanism via the `delay` construct, which postpones evaluation until the associated value is forced.

To illustrate the challenges compilers face when targeting such languages, consider the following reflective code snippet in R. This example demonstrates several language features that complicate optimization, and highlights the subtleties that arise from their interaction:

```
f = function(p, v) { a=1; p; print(a); v[1]='b' }
g = function(k, v) { assign(k, v, env=sys.frame(-1)) }
v = c(2, 3)      # Creation of a vector
f(g('a', 666), v)
print(v[1])
```

When executed, this program prints 666 and 2. The first printed value results from evaluating the argument `p` in function `f`. As arguments are lazily evaluated, `p` is actually a promise containing the code `g('a', 666)`. The expression `p` in the body of `f` forces its evaluation, triggering a call to `g` that uses reflection to update the local variable `a` in the caller's environment. The second value, 2, is printed because although `f` appears to mutate vector `v`, R's copy-on-write semantics ensure that this mutation applies to a duplicate. Additionally, the assignment of the string 'b' into a numeric vector causes implicit coercion of the entire vector to character type.

Each feature presents challenges to a compiler; however, it is often their interaction that makes reasoning and optimization especially difficult. Consider the following interdependencies:

- **Dynamic typing ⇒ late binding:** Without static type information, it is often impossible to determine which function is invoked at a call site, shifting many decisions to runtime.
- **Late binding ⇒ reflection:** When a callee is not known, the compiler conservatively assumes that the target function performs reflection—even if such behavior is rare.
- **Delayed evaluation ⇒ reflection:** Promises delay evaluation; the compiler often cannot predict if simply reading a variable will cause evaluation of reflective operations.
- **Code loading ⇒ late binding:** The ability to load code at runtime limits opportunities for static binding, as new definitions may change the program's behavior.

In summary, efficient compilation hinges on the compiler's ability to rule out—or at least localize—these dynamic behaviors. Dynamism may originate from the calling context, from function arguments, or from callees and code loaded during execution.

We choose to focus on the R language precisely because it represents a worst-case scenario. Among popular dynamic languages, R combines extensive use of reflection, lazy evaluation, dynamic typing, and mutable environments, making it a particularly demanding target for optimizing compilers.

3 Related Work

3.1 Compiler IRs

Aycock provides an accessible overview of early implementation strategies for dynamic languages [1], covering systems ranging from Smalltalk [7] and Self [4] to more modern environments such as Java and JavaScript [21, 25].

The idea of formally specifying and verifying compiler IRs is well established, though mostly for static languages. Java bytecode is both formally specified and verified [12, 17]; its bytecode verifier—a data-flow analysis ensuring variables are initialized before use—is analogous to FIŘ's well-flowedness

check, though Java's static typing makes this substantially simpler. Typed Assembly Language demonstrated that static types can preserve invariants even through low-level compilation [19]. FIŘ extends this idea to a setting where the source language lacks static types.

Other IR designs address different aspects of compilation. The sea-of-nodes representation makes all dependencies explicit in control and data flow graphs [5], enabling optimizations such as duplication removal and promise elimination, but operates at a lower level than FIŘ. CacheIR [6] targets JavaScript JIT compilation, organized around inline caching as a single optimization strategy; FIŘ targets a broader set of concerns, capturing multiple sources of dynamism in a type system. MLIR offers a general-purpose, extensible IR framework [16], but does not itself provide the type-level features FIŘ needs—gradual types, ownership, and reflection tracking—and formalizing type safety would require building these from scratch within its infrastructure.

3.2 Copy-on-Write, Ownership and Borrowing

Languages such as Swift and PHP implement copy-on-write semantics. Swift includes explicit `borrowing` and `consuming` annotations [13]: borrowing a value does not increment its reference count, while consuming a value transfers ownership to the callee. These correspond to FIŘ 's borrowed and owned parameter modes, respectively. PHP provides operations for assignment and aliasing; prior efforts in compilation have focused on distinguishing between them [14], but not on eliminating redundant copies. In FIŘ, aliasing cannot occur on owned variables: a variable that is aliased is considered shared, and copies will not be statically eliminated for it.

More broadly, ownership types constrain the use of aliases to heap-allocated objects [20], and borrowing relaxes strict ownership invariants [23]. Rust enforces these ideas via its borrow checker [18]. Immutable borrows in Rust correspond to FIŘ 's borrowing semantics: they are temporary and can reference owned values, which remain valid after the borrow expires. Rust's smart pointers resemble FIŘ 's shared values in that they do not enforce liveness, and any value assigned to them becomes immutable. Ownership in R is more complex than in any of these languages, due to lazy evaluation and first-class mutable environments. To handle this, FIŘ introduces an explicit **`consume`** instruction along with a flow-sensitive analysis to detect potential read-after-consume violations.

3.3 Gradual Typing

Gradual typing allows statically typed and dynamically typed code to coexist within the same language [22]. FIŘ adopts the three-level type stratification introduced in the Thorn programming language [24]: concrete types, whose values are statically guaranteed to conform to a kind; a dynamic type, for values whose type is entirely unknown; and *like types*, which record the compiler's expectation about a value without providing a static guarantee. In Thorn, for a variable of type `like string`, the compiler would check that it was only operated on as if it were a string, but would allow unchecked coercion from dynamic values. FIŘ repurposes this stratification for a compiler IR: like types record speculative information—such as feedback from previous executions—and an explicit type-cast is required to convert a like type to a concrete type, providing a natural point for deoptimization on failure (Section 4.6).

3.4 Deoptimization with Sourir and CoreIR

Sourir [11] and CoreIR [2] formalize speculative compilation and deoptimization. Both allow multiple versions of a function to coexist—typically an unoptimized version alongside optimized versions that include speculative assumptions. Sourir encodes speculation via an `assume` instruction: if the assumption holds, execution continues; otherwise, control transfers to the unoptimized version. CoreIR refines this with separate `Anchor` and `Assume` instructions, distinguishing non-deterministic deoptimization points from deterministic guard checks.

FIŘ shares the multi-version design but deliberately omits deoptimization from its formalization: including it would be straightforward but would obscure the core semantics with support for on-stack replacement (Section 4.10). Instead, FIŘ's type-cast mechanism serves an analogous role—a cast from a like type to a concrete type acts as a guard, providing a natural point for deoptimization on failure. Our implementation follows the approach of CoreIR for deoptimization.

3.5 The Ř compiler and PIR

FIŘ is designed as a replacement for PIR, the high-level intermediate representation of the Ř optimizing compiler for R (Fig. 2). The Ř compiler reuses the GNU-R front end to parse source code into an AST, and has its own interpreter and bytecode (RIR) as well as PIR for high-level transformations. Code generation is delegated to LLVM. PIR is an SSA-based IR [10]: each instruction has a type, effects, and immediates, where the type combines a set of kinds (*e.g.*, `real`, `int`, `str`) with flags tracking properties such as scalarity (**$**), objectness (+), and delayed evaluation (^). Fig. 3 shows the unoptimized identity function: `LdArg` loads an argument from the stack returning `val?^`, *i.e.* any value that might be missing or a promise; `MkEnv` creates an environment and stores the parameter in it; `LdVar` loads the variable into a register (`eR` signals possible failure); and `Force` evaluates the value if it was a promise, with **!** indicating possible reflective effects.

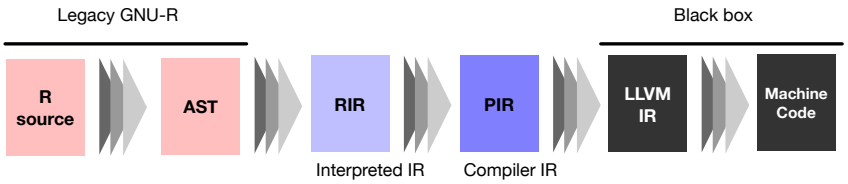

Fig. 2. Ř's existing IRs

The PIR type system is ad hoc. It has evolved into its current shape, rather than being designed. Its idiosyncrasies include the fact that types are implemented with flags, hindering composition. For example, the missing value type is represented as a kind but also as a flag resulting in complex types such as `val?^|?` indicating a promise that can evaluate into a missing value or is a missing value itself. FIŘ is a formally grounded redesign of PIR.

4 An IR for Dynamic Languages

An IR does not inherently create opportunities for new optimization; any transformation expressible in FIŘ can be performed on a lower-level IR. Instead, an IR should be evaluated on two criteria:

(1) does it make certain optimizations easier to express, and
(2) does it reduce the likelihood of introducing unsound transformations.

Compilers are filled with subtle invariants that are easy to violate as the code base evolves. FIŘ is deliberately opinionated in the features it exposes, and includes both a type checker and a flow analysis to detect violations of key invariants early. While the IR is tailored to R and the Ř compiler, the underlying principles are broadly applicable to dynamic languages—particularly those with fewer dynamic features than R.

4.1 Syntax of FIŘ

FIŘ is a core calculus that gives semantics to an intermediate language for compiling dynamic languages such as R. It deliberately abstracts away many of the low-level details required in a full-featured intermediate representation. FIŘ includes only a minimal set of data types and control-flow

```
val?^ %1.0 = LdArg 0
env   e1.1 = MkEnv a=%1.0, parent=G
val?^ %3.0 = LdVar a, e1.1        eR
val?  %3.3 = Force %3.0, e1.1     !
void       Return %3.3
```

Fig. 3. An identity function in PIR

constructs: it is an imperative language over integers and mutable integer vectors. Environments are not first-class values: while variables are typed, environments are not accessible as values. Unlike PIR, FIŘ is not SSA: the same variable can be assigned multiple times.

$$
\begin{array}{rcl}
\texttt{fn} & ::= & \textbf{fun}\ \texttt{f}\{\texttt{ver}_1 \ldots \texttt{ver}_n\} \\
\texttt{ver} & ::= & (\texttt{rdefs}) \xrightarrow{\texttt{fx}} \texttt{t}\{\texttt{defs; e}\} \\
\texttt{rdefs} & ::= & \texttt{r}_0 : \texttt{t}_0 \ldots \texttt{r}_n : \texttt{t}_n \\
\texttt{defs} & ::= & \texttt{v}_0 : \texttt{t}_0 \ldots \texttt{v}_n : \texttt{t}_n \\
\texttt{t} & ::= & \texttt{k ox cx} \\
\texttt{k} & ::= & * \mid \textbf{V} \mid \textbf{I} \mid \textbf{v(I)} \mid \textbf{p}^{\texttt{fx}}\texttt{(t)} \\
\texttt{ox} & ::= & \textbf{o} \mid \textbf{b} \mid \textbf{s} \mid \textbf{f} \\
\texttt{fx} & ::= & \textbf{+} \mid \textbf{-} \\
\texttt{cx} & ::= & \texttt{!} \mid \texttt{?} \\
\texttt{sig} & ::= & \texttt{t}_0 \ldots \texttt{t}_n \xrightarrow{\texttt{fx}} \texttt{t} \\
\texttt{v} & ::= & \texttt{x} \mid \texttt{r}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{e} & ::= & \texttt{i} \mid \textbf{vec}(\texttt{e}_1 \ldots \texttt{e}_n) \mid \texttt{v} \mid \texttt{e[e]} \mid \texttt{e\$x} \\
& \mid & \texttt{v = e} \mid \texttt{e[e] = e} \mid \texttt{e\$x = e} \\
& \mid & \texttt{ver} \leftarrow (\texttt{e}_0 \ldots \texttt{e}_n) \\
& \mid & \texttt{f.i}(\texttt{e}_0 \ldots \texttt{e}_n) \\
& \mid & \texttt{f<sig>}(\texttt{e}_0 \ldots \texttt{e}_n) \\
& \mid & \texttt{e}\ \textbf{as}\ \texttt{t} \\
& \mid & \textbf{force}\ \texttt{e} \\
& \mid & \textbf{consume}\ \texttt{r} \\
& \mid & \textbf{dup}\ \texttt{e} \\
& \mid & \textbf{prom}^{\texttt{fx}}\texttt{<t>\{e\}} \\
& \mid & \texttt{e ; e}
\end{array}
$$

Fig. 4. Syntax

The full syntax of the FIŘ calculus is given in Fig. 4. A program consists of a *function table* F and an *expression* e. A function table consists of one or more *function* declarations, $\textbf{fun}\ \texttt{f}\{\texttt{ver}_1 \ldots \texttt{ver}_n\}$, where f is the function's unique name and $\texttt{ver}_i$ is a version. A *version*, $(\texttt{rdefs}) \xrightarrow{\texttt{fx}} \texttt{t}\ \{\ \texttt{defs; e}\ \}$, has typed argument definitions rdefs, local variables defs, a body expression e, a return type t, and a reflection bit fx. Variables are either *register variables* (r) or *named variables* (x); the distinction, motivated by reflection, is discussed in Section 4.3.

The expressions (Fig. 4) include standard constructs (integer and vector literals, variable reads and writes, sequencing), reflective access to named variables (e$x and e$x = e), three forms of function application (static calls, dispatched calls, and inlined versions), and constructs specific to FIŘ: type-casts (e **as** t), promise creation ($\textbf{prom}^{\texttt{fx}}$<t>{e}) and forcing (**force** e), duplication (**dup** e), and register consumption (**consume** r). Each is discussed in the following subsections. Examples use a slightly beautified syntax: we add reg and var annotations to distinguish between register and named variables, omit semicolons, allow declarations and assignments to be combined, and use defaults in type signatures (types with no ownership modifier are **s**, and those with no concreteness modifier are **!**).

4.2 Functions and Versions

In FIŘ, each function can have multiple implementations, or *versions*. Fig. 5 presents a source-language function definition (left) alongside three compiler-generated versions (right): a baseline version that handles any argument, a version specialized for a promise wrapping an integer vector,

and a version specialized for a borrowed integer vector. Internally, the compiler maintains a flat namespace, where each unique function in the program has a single entry. Thus, `fn#1` in Fig. 5 contains all the compiled versions of the function expression shown on the left. The mapping between source-level function names and compiler-internal identifiers is an implementation detail. For our purposes, we focus on the internal table of functions maintained by the compiler.

Each version is intended to be semantically equivalent to the source function under some assumptions about its execution context. Verifying this equivalence is beyond our scope; instead, the IR guarantees that each version—independently of its siblings—is well-typed, and satisfies a property we call well-flowed, meaning that registers are not used before they are defined (cf. Section 5).

A version has a *signature* that consists of argument types, return type, and reflection bit, as well as a *body* (e.g. `(reg r:I)→I { r+1 }`). Versions are created by the compiler to provide optimized implementations of a function. They differ in their signature and the transformations applied to the body. Versions need not be mutually exclusive in their signature, and the compiler is free to select among them—possibly considering additional factors such as the likelihood of deoptimization or resource requirements.

4.3 Scopes and Variables

Each version defines a *scope* in which variables can be defined. Our source language allows scopes to be shared with promises and to be inspected reflectively, so variables that appear in the source code must be preserved in case another function looks them up. To allow the compiler to eliminate scopes that are not reflectively inspected, FIŘ distinguishes two kinds of variables: *named variables*, which are looked up symbolically and are exposed to reflection, and *register variables*, which are not available to reflective operations and can be implemented in any way the compiler sees fit. Parameters are passed as register variables. A common pattern is to store them into named variables before calling a function that may perform reflection:

```
(reg r:*)+→* {
```

```
fn=function(a) {
   b = 0;
   b = length(a);
   return(b);
}
```

```
function
fun fn#1 {
version 1
   (reg r:p+(*))+→I {
      var a:* = r
      length<*+→I>(a)
   }
version 2
   (reg r:p(v(I)))→I {
      reg t:v(I)
      t = force r
      length.1(t)
   }
version 3
   (reg r:v(I)b)→I {
      length.1(r)
   }
}
```

register variable
promise type
named variable
reflective effect

Fig. 5. Source code (left) and FIŘ code with three versions (right)

```
    var a:* = r
    someFun<*+→*>(a)
}
```

While our source language allows shadowing and nested scopes, FIŘ does not model nesting; any lookup outside the current scope returns an unknown value.

Reflective reads and writes. A promise in FIŘ captures both an expression—its body—and the environment of the enclosing version. Given a variable `r` referencing a promise defined in a different version, the following mutates a named variable `x` in the captured environment:

```
r$x = 42
```

This can occur regardless of the expected type of `x`, or even its existence. Only named variables can be accessed reflectively; register variables can only be accessed from their definition scope.

Reflective access makes naive inlining unsound: inlining a version's body into the surrounding context merges their environments, altering behavior when reflection is present. Consider:

```
r1 = ((reg r:v(I)o)→I { r[1] = 2 }) <-- (consume r0)
r1 = (r0[1] = 2)
```

These two statements are equivalent since the version's body does not employ reflection. This contrasts with the following two lines, which have different behaviors (we assume `r0` is a promise capturing the current local environment):

```
r1 = ((reg r:p+(*))+→* { var x:* ; x = 41 ; r$x = 42 ; x }) <-- (consume r0)
var x:* ; r1 = (x = 41 ; r0$x = 42 ; x)
```

The first line yields `r1=41`, while the second yields `r1=42`.

4.4 Promises: Creation and force

A *promise* represents a delayed computation. In the source language, promises are *forced* implicitly when their value is needed, and their result is cached upon evaluation. In FIŘ, promises are first-class values with well-defined types that must be explicitly forced:

```
r : p(I)
r = prom<I>{ 42 }
force r
```

Here, `r` is a promise that, when forced, returns an integer. As a more realistic example, the source expression `f(x)` may be translated to:

$$\texttt{f}\,\texttt{<}\mathbf{p}^{+}\texttt{(}\mathbf{V}\texttt{)}\xrightarrow{+}\mathbf{V}\texttt{>}\,\texttt{(}\mathbf{prom}^{+}\texttt{<}\mathbf{V}\texttt{>\{}\mathbf{force}\ \texttt{x\}}\texttt{)}$$

This passes to `f` a promise that, when evaluated, reads and forces the variable `x`. Promises execute in the lexical scope in which they were created: they can read and update variables in that scope, and may even introduce new bindings—for instance, in `f(z=x+y)` if `z` was not previously defined.

In R, promises are automatically forced when assigned to a variable or returned from a function, so they do not outlive the scope in which they were created. Our type system leverages this invariant to simplify reasoning about promise lifetimes (Section 4.6). Because `force` triggers evaluation of a delayed computation—potentially with side effects—the compiler cannot freely reorder code around it. FIŘ's flow analysis ensures that variables are not read before they are written (Section 5.1).

4.5 Copy-on-Write

FIŘ supports copy-on-write semantics by making duplication explicit through a dedicated `dup` operation. All program points where a copy may be required must be explicitly marked with `dup`. The actual implementation of `dup` is left to the compiler: in a language like R that maintains an over-approximating reference count, `dup` needs to copy only when the count exceeds one. By making duplication explicit, the IR enables optimizations that eliminate unnecessary copies—copies that would otherwise be indistinguishable from semantically required ones when performed implicitly by other operations. The example below illustrates how the compiler inserts a `dup` before a vector is updated:

```
(reg r1: v(I)) : I {
  reg r2: v(I)o = dup r1
  r2[1] = 42
}
```

Once duplicated, the vector can be safely mutated multiple times without further copying. As a result, function arguments typically undergo a single duplication before any mutation, enabling efficient updates while preserving copy-on-write semantics. The interaction between duplication and ownership is discussed in Section 4.6.

4.6 Ownership and Gradual Typing

Types in FIŘ, ranged over by `t`, serve multiple purposes: they verify that compiler transformations preserve well-formedness, guide optimizations, and record compiler-generated hints. As described in Section 3.3, FIŘ adopts the three-level type stratification of Thorn [24]—concrete types, a dynamic type, and like types—and repurposes it for a compiler IR. In addition, the type system tracks ownership qualifiers, enabling optimizations such as copy elimination.

Formally, each type is composed of three components: a *kind* (`k`), defining the structure of the value (e.g. `I` for integers, `v(I)` for vectors of integers); an *ownership modifier* (`ox`); and a *concreteness modifier* (`cx`), indicating whether the type is certain (`!`) or speculated (`?`). A reflection bit (`fx`) extends the types of functions and promises (Section 4.7).

Expressions of a concrete type are guaranteed by the static type system to reduce to a value that conforms to their kind `k`. Consequently, no runtime checks or wrappers are needed. Like types are the only types that can be given to named variables, as we cannot statically prevent them from being reflectively reassigned to a value of a different kind. The dynamic type, represented by the kind `*`, captures any value; it can only be associated with the concreteness `?` (the `?` is elided in our examples).

The specific kinds are: scalar value (**`I`**), mutable vector (**`v(I)`**), and promise (**$\mathbf{p}^{\mathtt{fx}}(\mathtt{t})$**); the remaining kinds represent the union of the above: value (**`V`**) is the union of scalars and vectors, and any (**`*`**) is the union of all.

Ownership modifiers can be `f` (fresh), `o` (owned), `s` (shared), or `b` (borrowed). A value is fresh if it is not referenced by any variable (*e.g.*, a newly-allocated value). The rules for copy-on-write are that any shared mutable value must be duplicated before being updated:

```
reg r2:v(I)o! = ...
reg r2:v(I)s! = dup r1
r2[0] = 42
```

The `dup` operation performs a copy (if needed, the source language maintains reference counts) and changes the ownership from `s` to `o` (owned). An owned value is uniquely referenced, even though it can be temporarily borrowed during a function call as long as this function does not mutate it

and does not leak it (so that the value remains unique after the call). Borrowed types exist only in function parameters.

An owned value cannot be assigned to a variable, otherwise it would compromise uniqueness as it could be then assigned to multiple live variables. A fresh value can be assigned to an owned or shared variable. A value can be made fresh by copying using `dup`. Furthermore, a value in an owned register variable can become fresh by "consuming" the register (`consume r`); flow analysis ensures a register cannot be accessed after it is consumed.

```
(reg r1:v(I)o!)→v(I)o! { # `r1' is owned: it should not be aliased
    reg r2:v(I)o! ;
    r2 = r1 ; # Not allowed, because both `r1' and `r2' would reference the same value
    r2 = dup r1 ; # Ok, because `r1' points to a separate copy
    r2 = consume r1 ; # Ok, because it disallows any future access to `r1'
}
```

Ownership information can be used to statically eliminate useless copies of vectors. For instance, consider the following code:

```
(reg r1: v(I)o!) : I {
  reg r2: v(I)o! = dup r1
  r2[1] = 42
}
```

As `r1` is owned and is not accessed anymore after the duplication, `dup r1` can be replaced by `consume r1` (which is a no-op at runtime, but prevents `r1` from being accessed after this point):

```
(reg r1: v(I)o!) : I {
  reg r2: v(I)o! = consume r1
  r2[1] = 42
}
```

This code can then be simplified into the following:

```
(reg r1: v(I)o!) : I {
  r1[1] = 42
}
```

The syntax of types is permissive, but we restrict types that may appear in declarations. First, we define a left-associative component setter operator & on types:

$$\begin{array}{lcll} \mathsf{k}\,\mathsf{ox}\,\mathsf{cx} & \& & \mathsf{k}' & = & \mathsf{k}'\,\mathsf{ox}\,\mathsf{cx} \\ \mathsf{k}\,\mathsf{ox}\,\mathsf{cx} & \& & \mathsf{ox}' & = & \mathsf{k}\,\mathsf{ox}'\,\mathsf{cx} \\ \mathsf{k}\,\mathsf{ox}\,\mathsf{cx} & \& & \mathsf{cx}' & = & \mathsf{k}\,\mathsf{ox}\,\mathsf{cx}' \end{array}$$

We use a number of predicates for convenience:

$$\begin{array}{rcl} \mathsf{shared}(\mathsf{t}) & \equiv & (\mathsf{t} \,\&\, \mathbf{s}) = \mathsf{t} \\ \mathsf{owned}(\mathsf{t}) & \equiv & (\mathsf{t} \,\&\, \mathbf{o}) = \mathsf{t} \\ \mathsf{fresh}(\mathsf{t}) & \equiv & (\mathsf{t} \,\&\, \mathbf{f}) = \mathsf{t} \\ \mathsf{borrowed}(\mathsf{t}) & \equiv & (\mathsf{t} \,\&\, \mathbf{b}) = \mathsf{t} \\ \mathsf{like}(\mathsf{t}) & \equiv & (\mathsf{t} \,\&\, ?) = \mathsf{t} \\ \mathsf{vec}(\mathsf{t}) & \equiv & (\mathsf{t} \,\&\, \mathbf{v}(\mathbf{I}) \,\&\, \mathbf{!}) = \mathsf{t} \\ \mathsf{prom}(\mathsf{t}) & \equiv & \exists \mathsf{t}', \mathsf{fx}.\ (\mathsf{t} \,\&\, \mathbf{p}^{\mathsf{fx}}(\mathsf{t}') \,\&\, \mathbf{!}) = \mathsf{t} \\ \mathsf{value}(\mathsf{t}) & \equiv & (\mathsf{t} \,\&\, \mathbf{I} \,\&\, \mathbf{!} = \mathsf{t}) \text{ or } (\mathsf{t} \,\&\, \mathbf{v}(\mathbf{I}) \,\&\, \mathbf{!} = \mathsf{t}) \text{ or } (\mathsf{t} \,\&\, \mathbf{V} \,\&\, \mathbf{!} = \mathsf{t}) \end{array}$$

We can now define the meaning of *well-formed* for types:

$$\frac{\mathsf{t} = \mathsf{t} \,\&\, * \,\&\, ?}{\mathsf{wf}(\mathsf{t})} \qquad \frac{\mathsf{t} = \mathsf{t} \,\&\, \mathbf{I}}{\mathsf{wf}(\mathsf{t})} \qquad \frac{\mathsf{t} = \mathsf{t} \,\&\, \mathbf{v}(\mathbf{I})}{\mathsf{wf}(\mathsf{t})} \qquad \frac{\mathsf{t} = \mathsf{t} \,\&\, \mathbf{V}}{\mathsf{wf}(\mathsf{t})} \qquad \frac{\mathsf{wf}(\mathsf{t}') \qquad \mathsf{t} = \mathsf{t} \,\&\, \mathbf{p}^{\mathsf{fx}}(\mathsf{t}') \qquad \mathsf{shared}(\mathsf{t}') \qquad \mathsf{value}(\mathsf{t}')}{\mathsf{wf}(\mathsf{t})}$$

Each rule defines well-formedness constraints for a specific kind. For instance, the first rule ensures that a type of the any kind (∗) has a concreteness modifier of ? (`t = t & * & ?` is true if and only if `t` is of kind ∗ and of concreteness ?). The last rule ensures that the type returned by a promise is shared (it is referenced by the promise itself, which will return this same value each time it is forced) and cannot itself be a promise. The well-formedness constraints are extended to variable declarations with:

$$\frac{\texttt{wf(t)} \quad \texttt{shared(t)} \quad \texttt{like(t)}}{\texttt{wf(x : t)}} \qquad \frac{\texttt{wf(t)} \quad \neg\texttt{fresh(t)}}{\texttt{wf(r : t)}}$$

This ensures that named variables are always shared and that their concreteness is ?, and register variables cannot be fresh.

Subtyping and casts. A type-cast `e` **`as`** `t` checks that e has type t at runtime (or a subtype of it). Subtyping on types has, for instance, type `Io!` be a subtype of `Io?`, itself a subtype of ∗. Two types can only be in subtyping relation if they have the same ownership: the ownership of a variable is invariant, and a change of ownership can only occur through an explicit operation. The role of type-casts is essential in FIŘ, as they are the only way to convert a like type into a concrete type. Thereby, any named variable must be cast before being used on an operation that requires a concrete type. Type casts can also be used on expressions with a concrete type in order to refine this type (e.g., refining a type **`Vo!`** into **`Io!`**).

Type-casts are inserted when some uncertain type information needs to be checked against the expectations of the compiler in order to guarantee a safe execution: reflection may have altered the type of a variable in a way that cannot be predicted statically, the type of an expression may have been speculated by the compiler to enable more optimizations, etc. In a practical implementation, type-casts should provide an alternative execution in the case of failure: for instance, it could trigger some deoptimization mechanism that switches to a more general version of the current function. However, FIŘ does not assume any specific deoptimization mechanism. A compiler implementation is free to implement speculation and deoptimization (such as CoreIR [2]) or not.

4.7 Reflection in the Type System

Promise types and function signatures carry a *reflection bit*: + indicates that the promise or function may use reflection, while - ensures it does not. For example, the kind of a promise that yields an integer and may perform reflection when forced is written $\mathbf{p}^{+}$`(Is!)`. The first time a promise is forced, its result is cached, so the value returned by a promise is considered shared.

Reflection does not directly impact the typeability of an expression, as named variables are already treated pessimistically: they have dynamic or like types. Still, annotating promises and function signatures with their potential for reflection is essential for optimizations—in particular, scope elision is only sound when the compiler can verify that no reflective operation observes a scope.

4.8 Binding Time

When considering a call such as `f(x)`, it may be impossible to determine statically which function is being invoked. In dynamic languages, this ambiguity can arise for several reasons: `f` may be an argument passed by the caller of the current function; it may refer to a global symbol that can be redefined at runtime; or it may be resolved through dynamic dispatch based on properties of the argument. Our source language supports all of these forms of dynamic binding.

FIŘ does not have built-in support for dynamic binding mechanisms; both static and dispatched calls point to statically-known functions, though the version may be dynamic. Unknown functions can be called by passing them to an intrinsically-defined “dynamic call” function from the source

language's reference implementation. In practice, we are able to statically resolve most calls using speculation, but that is beyond the scope of this paper.

A function call is a *static call* if the target version of the function is known at compile time. In contrast, a *dispatched call* refers to cases where the compiler selects the most appropriate version based on a combination of static and dynamic information available at the call site. The compiler may restrict the set of candidate versions statically and generate dispatch logic to choose among the remaining versions at runtime. We do not mandate how a compiler selects the "best" version; each implementation may rely on its own heuristics, profiling data, or runtime inspection. A dispatched call to `fn#1` is written as:

```
r = fn#1<p+(*)→+I>( r0 ) # Dispatched call
```

Here, the signature <p+(*)→+I> constrains the version of `fn#1` that the compiler will call; this signature is compatible with version 1 in Fig. 5. A more sophisticated compiler—such as Ř, which supports contextual dispatch [9]—might dynamically inspect the promise held in `r0`. If it determines that the promise is reflection-free and returns an integer, it could instead select version 2.

Dispatched calls are annotated with static type signatures for two reasons. First, some properties required to select a compatible version at runtime may not be dynamically recoverable: ownership qualifiers are computed statically and cannot be inferred from the program state, and the kind of an argument cannot be determined at runtime if the argument is `undef`. Second, to ensure soundness, the version selected at runtime must return a value that is type-compatible with the version selected statically. Two versions of the same function may produce different results for the same inputs—this is acceptable because each version is optimized under different assumptions—so long as the selected version conforms to the type system. The type system ensures that at least one version is type-compatible with the annotated static signature.

Static calls reference a version directly by index and do not require a signature. At runtime, the version is called without dispatch. A static call to version 3 of `fn#1` is written as:

```
r1 = fn#1.3( force r0 ) # Static call
```

Concrete type information can be used to devirtualize a dispatched call into a static one. For instance, a call to `fn#1` (Fig. 5) could be statically bound to version 3 if the argument is known to be a vector of integers (`v(I)o!`). Like types, which do not offer static guarantees, can provide hints for the same purpose. For example, a version typed with `I?` indicates an expectation that it will be called with an integer:

```
(reg r0:I?)→* {
    reg r1:I = r0 as I # deoptimize if fails, not shown here
    r1 + 1
}
```

The body must cast the like type to a concrete type before use, and should deoptimize on failure. In this context, casts serve as guards.

4.9 Inlining

When the compiler knows which version will be executed—such as in a static call—it may replace the call with the body of the corresponding version. Fig. 6 illustrates the syntax used to represent inlined code. A key point is that inlining preserves the scope of the inlined version: named variables defined within the callee remain in a distinct environment, separate from the caller. This is required because reflection can observe scope boundaries.

If the compiler can determine that no reflective operation will observe a particular scope, it may safely remove it. This optimization is known as scope elision and is separate from inlining. FIŘ does

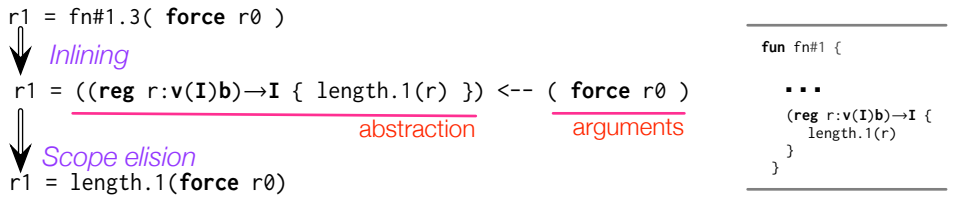

Fig. 6. Inlining and scope elision

not define how a scope is represented at runtime, but provides static information that may help optimize the representation, in particular the distinction between variables that can be accessed through reflection and those that cannot.

4.10 Discussion

The design of FIŘ draws on lessons from prior work on the Ř compiler infrastructure. This section discusses noteworthy design choices and simplifications.

Deoptimization. Our implementation follows the approach of CoreIR [2] for deoptimization. For simplicity, we omit these constructs from the definition of FIŘ. Including them would be straightforward but would obscure the core semantics with support for on-stack replacement. The only requirement that deoptimization adds is an identified base version for each function; this version is called when speculation fails.

Reflection. FIŘ allows named variables to be redefined or deleted via reflection. The specific reflection interface is not specified, but this possibility is why named variables can only have like types—if the source language did not expose local variables to reflection, they could be given stronger types. Consider a version that stores a value in named variable `a`, calls a possibly-reflective function `f`, and returns `a`:

```
(reg r:*)+→* {
  var a:*
  a = r
  f<* +→*>( 42 )
  a
}
```

In R, `f` can delete `a`, causing the lookup to return the value of `a` in an enclosing scope, or to fail.

Promises. In FIŘ, promises cannot escape their scope of creation. Without this restriction, any register accessed in a promise that outlives its scope would have to be considered shared, complicating the ownership analysis. This restriction is acceptable because in R, promises are forced when assigned or returned. It is possible for a promise to escape via a *delayed assignment*[1], or by returning the environment in which it lives (environments are first-class values in R), but these cases are rare and can be treated as reflection.

Omitted Features. The R language has features that are not modeled in FIŘ. Most are implemented by calls to native functions and do not need to be reflected in the IR. Two are worth mentioning: name lookups and attributes.

[1]A delayed assignment is a special instruction that assigns an expression to a variable without forcing the underlying promise.

The semantics of name lookup in R is surprisingly rich: a chain of environments is searched, and in some cases promises must be forced to determine which value to return. These complex lookups are limited to named variables and do not affect FIŘ because the compiler only reasons about names in the current scope; for names outside that scope, FIŘ makes no assumption and the implementation defaults to the source language lookup mechanism.

In R, any value can have attached attributes—a map from name to value. For example, a `class` attribute gives a value a particular behavior during object-oriented dispatch. The Ř compiler tracks only the absence of attributes, which is the overwhelmingly common case. This could be added to FIŘ as an extra bit on value types; a more fine-grained treatment is left to future work.

With these design choices in place, we now turn to the formal semantics of FIŘ.

5 Static and Dynamic Semantics of FIŘ

We first define a static semantics for FIŘ, composed of a type-checker and a flow analysis (Section 5.1). We then define a dynamic semantics (Section 5.2), and prove that programs that are well-typed and well-flowed cannot get stuck (Section 5.3).

5.1 Static Semantics

The static semantics of FIŘ are defined by two relations over versions, $[\![\mathtt{ver}]\!] : \mathtt{sig}$ and $\mathtt{F}[\![\mathtt{ver}]\!]$. These relations assert, respectively, that a version `ver` is well-typed and well-flowed. The function table F of a program is considered well-defined when every version within it satisfies both criteria. The typing relation guarantees adherence to constraints related to types, ownership, concreteness, and reflection. The flow relation ensures that register variables are initialized before they are used and prevents their access after a **`consume`** operation. Although both properties are required for safety, we define two separate analyses: the typing relation is checked by a flow-insensitive analysis (with an invariant type environment), while the flow relation is checked by a simpler but flow-sensitive analysis—mirroring the separation between type checking and borrow checking in Rust.

For convenience, we consider the function table F to be fixed, and allow the deduction rules for the judgments $[\![\mathtt{ver}]\!] : \mathtt{sig}$ and $\mathtt{F}[\![\mathtt{ver}]\!]$ to access it, without making this dependency explicit in the judgment.

We establish transitive, reflexive and antisymmetric subtyping relations as follows:

- For reflection: $\mathbf{-} \leq_f \mathbf{+}$.
- For ownership, subtyping is restricted to equality: $\mathtt{ox} \leq_o \mathtt{ox}$.
- For concreteness modifiers: $\mathbf{!} \leq_c \mathbf{?}$.
- For kinds:

$$\frac{}{\mathbf{I} \leq_k \mathbf{V}} \qquad \frac{}{\mathbf{v}(\mathbf{I}) \leq_k \mathbf{V}} \qquad \frac{}{\mathtt{k} \leq_k \mathbf{*}} \qquad \frac{\mathtt{t} \leq \mathtt{t}' \qquad \mathtt{fx} \leq_f \mathtt{fx}'}{\mathbf{p}^{\mathtt{fx}}(\mathtt{t}) \leq_k \mathbf{p}^{\mathtt{fx}'}(\mathtt{t}')}$$

- For types:

$$\frac{\mathtt{k} \leq_k \mathtt{k}' \qquad \mathtt{ox} \leq_o \mathtt{ox}' \qquad \mathtt{cx} \leq_c \mathtt{cx}'}{\mathtt{k}\,\mathtt{ox}\,\mathtt{cx} \leq \mathtt{k}'\,\mathtt{ox}'\,\mathtt{cx}'}$$

- For version signatures:

$$\frac{\mathtt{t'}_0 \leq \mathtt{t}_0 \ldots \mathtt{t'}_n \leq \mathtt{t}_n \qquad \mathtt{t} \leq \mathtt{t}' \qquad \mathtt{fx} \leq_f \mathtt{fx}'}{\mathtt{t}_0 \ldots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \leq_{sig} \mathtt{t'}_0 \ldots \mathtt{t'}_n \xrightarrow{\mathtt{fx}'} \mathtt{t}'}$$

Type checking. The [VER] rule shown next checks the definition of a version. The variables that can be used in a version are its parameters (all registers), and locals (registers and named variables); these variables and their types make up the type context which is used to type the version's body.

All type annotations must be well-formed. The reflection bit and return type of the version must be those inferred for its body, with the ownership component transformed as follows: if the body returns an owned value, the returned value is fresh (the only reference ceases to exist when exiting the version's scope); otherwise ownership is unchanged. A version that returns a borrowed value is untypeable. Versions can only return non-promise values; allowing a promise to escape its scope would force every register it captures to be considered shared (cf. Section 5.3).

$$[\textsc{Ver}]\ \frac{\begin{array}{c}\Gamma = \texttt{rdefs}, \texttt{defs} \qquad \texttt{wf}(\Gamma) \\ \Gamma \vdash [\![\texttt{e}]\!] : \texttt{t}' \\ \texttt{types}(\texttt{rdefs}) = \texttt{t}_0 \dots \texttt{t}_n \\ \Gamma \vdash \texttt{R}[\![\texttt{e}]\!] = \texttt{fx} \qquad \texttt{t} = \begin{cases} \texttt{t}' & \text{if } \texttt{shared}(\texttt{t}') \text{ or } \texttt{fresh}(\texttt{t}') \\ \texttt{t}' \,\&\, \mathbf{f} & \text{if } \texttt{owned}(\texttt{t}') \end{cases} \\ \texttt{value}(\texttt{t}) \qquad \texttt{wf}(\texttt{t}) \end{array}}{[\![(\texttt{rdefs}) \xrightarrow{\texttt{fx}} \texttt{t}\{\texttt{defs};\ \texttt{e}\}]\!] : \texttt{t}_0 \dots \texttt{t}_n \xrightarrow{\texttt{fx}} \texttt{t}}$$

The reflection analysis $\Gamma \vdash \texttt{R}[\![\texttt{e}]\!] = \texttt{fx}$ is described below (Fig. 8). Fig. 7 gives the typing judgments. These rely on the following auxiliary definition, which relates parameter types $\texttt{t}'$ with the argument types $\texttt{t}$ they can receive:

$$\frac{\begin{array}{c}\texttt{t} \,\&\, \mathbf{f} \leq \texttt{t}' \,\&\, \mathbf{f} \\ \texttt{shared}(\texttt{t}') \Rightarrow (\texttt{shared}(\texttt{t}) \text{ or } \texttt{fresh}(\texttt{t})) \\ \texttt{owned}(\texttt{t}') \Rightarrow \texttt{fresh}(\texttt{t}) \end{array}}{\texttt{match}(\texttt{t}, \texttt{t}')}$$

A parameter of type $\texttt{t}'$ can be given an argument of type $\texttt{t}$ if and only if: (1) the kind and concreteness of $\texttt{t}$ are subtypes of those of $\texttt{t}'$; and (2) if $\texttt{t}'$ expects a borrowed value, $\texttt{t}$ can have any ownership; if $\texttt{t}'$ expects a shared value, $\texttt{t}$ must be shared or fresh; if $\texttt{t}'$ expects an owned value, $\texttt{t}$ must be fresh. Note that function parameters must be well-formed, so they cannot have fresh ownership.

The only literals are integers, considered shared by convention (ownership is irrelevant for immutable values). All types are explicit; promise return and call signatures are checked, never inferred. Sequences have the type of their last element. Only vectors can be indexed, and only integers can be used as indices. Variables read via reflection always have unknown types, and variables of any type can be written via reflection; the latter named variable types are never concrete. Promises cannot be reflectively written, preventing them from escaping their scope of definition (cf. Section 5.3).

The type system as defined in Fig. 7 is not syntax-directed because of the subsumption rule Sub. This is a presentation choice for clarity and concision; subsumption can easily be inlined in other rules when needed.

Reflection analysis. Fig. 8 shows the analysis that checks reflection annotations: a function must have the reflection bit (+) if any of its applied versions or forced promises have the reflection bit. Note that the reflection analysis does not impact the typeability directly (reflection has no influence on the applicability of the deduction rules of the type system), but the information it brings is useful for optimizing code, as discussed in Section 1.

- A version is reflective if annotated as such. A version must be annotated as reflective if its body is reflective.
- Force is reflective if the type of the promise being forced has the reflective modifier. This makes the reflection analysis and the type system mutually recursive, though this definition is well-founded as recursive calls only apply on strict sub-expressions.

$$[\textsc{Lit}]\ \frac{}{\Gamma \vdash [\![\mathtt{i}]\!] : \mathbf{Is!}} \qquad [\textsc{Var}]\ \frac{}{\Gamma, \mathtt{v} : \mathtt{t} \vdash [\![\mathtt{v}]\!] : \mathtt{t}} \qquad [\textsc{Sub}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t} \quad \mathtt{t} \leq \mathtt{t}'}{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t}'} \qquad [\textsc{Seq}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t} \quad \Gamma \vdash [\![\mathtt{e}']\!] : \mathtt{t}'}{\Gamma \vdash [\![\mathtt{e}; \mathtt{e}']\!] : \mathtt{t}'}$$

$$[\textsc{Pro}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t}' \quad \Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx} \quad \mathtt{t} = \mathbf{p}^{\mathtt{fx}}(\mathtt{t}')\mathbf{s!} \quad \mathtt{wf}(\mathtt{t})}{\Gamma \vdash [\![\mathbf{prom}^{\mathtt{fx}} \langle \mathtt{t}' \rangle \{\mathtt{e}\}]\!] : \mathtt{t}} \qquad [\textsc{Vec}]\ \frac{\Gamma \vdash [\![\mathtt{e}_1]\!] : \mathbf{Is!} \ldots \Gamma \vdash [\![\mathtt{e}_n]\!] : \mathbf{Is!} \quad \mathtt{t}' = \mathbf{v}(\mathbf{I})\mathbf{f!}}{\Gamma \vdash [\![\mathbf{vec}(\mathtt{e}_1 \ldots \mathtt{e}_n)]\!] : \mathtt{t}'}$$

$$[\textsc{Rea}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t} \quad \Gamma \vdash [\![\mathtt{e}']\!] : \mathbf{Is!} \quad \mathtt{t}\ \&\ \mathbf{f} \leq \mathbf{v}(\mathbf{I})\mathbf{f!}}{\Gamma \vdash [\![\mathtt{e}[\mathtt{e}']]\!] : \mathbf{Is!}}$$

$$[\textsc{App}]\ \frac{[\![\mathtt{ver}]\!] : \mathtt{t}_0 \ldots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \quad \Gamma \vdash [\![\mathtt{e}_0]\!] : \mathtt{t}'_0 \ldots \Gamma \vdash [\![\mathtt{e}_n]\!] : \mathtt{t}'_n \quad \mathtt{match}(\mathtt{t}'_0, \mathtt{t}_0) \ldots \mathtt{match}(\mathtt{t}'_n, \mathtt{t}_n)}{\Gamma \vdash [\![\mathtt{ver} \leftarrow (\mathtt{e}_0 \ldots \mathtt{e}_n)]\!] : \mathtt{t}}$$

$$[\textsc{Cal}]\ \frac{\mathtt{sig}(F(\mathtt{f})(\mathtt{i})) = \mathtt{t}_0 \ldots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \quad \Gamma \vdash [\![\mathtt{e}_0]\!] : \mathtt{t}'_0 \ldots \Gamma \vdash [\![\mathtt{e}_n]\!] : \mathtt{t}'_n \quad \mathtt{match}(\mathtt{t}'_0, \mathtt{t}_0) \ldots \mathtt{match}(\mathtt{t}'_n, \mathtt{t}_n)}{\Gamma \vdash [\![\mathtt{f.i}(\mathtt{e}_0 \ldots \mathtt{e}_n)]\!] : \mathtt{t}}$$

$$[\textsc{Dis}]\ \frac{\mathtt{sig} = \mathtt{t}_0 \ldots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \quad \exists \mathtt{i}.\ \mathtt{sig}(F(\mathtt{f})(\mathtt{i})) \leq \mathtt{sig} \quad \Gamma \vdash [\![\mathtt{f.i}(\mathtt{e}_0 \ldots \mathtt{e}_n)]\!] : \mathtt{t}'}{\Gamma \vdash [\![\mathtt{f} \langle \mathtt{sig} \rangle (\mathtt{e}_0 \ldots \mathtt{e}_n)]\!] : \mathtt{t}} \qquad [\textsc{Cas}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t}' \quad \mathtt{wf}(\mathtt{t}) \quad \mathtt{t}' \leq \mathtt{t} \text{ or } \mathtt{t} \leq \mathtt{t}'}{\Gamma \vdash [\![\mathtt{e}\ \mathbf{as}\ \mathtt{t}]\!] : \mathtt{t}}$$

$$[\textsc{For}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathbf{p}^{\mathtt{fx}}(\mathtt{t})\mathbf{s!}}{\Gamma \vdash [\![\mathbf{force}\ \mathtt{e}]\!] : \mathtt{t}}$$

$$[\textsc{Use}]\ \frac{\Gamma \vdash [\![\mathtt{r}]\!] : \mathtt{t} \quad \mathtt{owned}(\mathtt{t}) \quad \mathtt{t}' = \mathtt{t}\ \&\ \mathbf{f}}{\Gamma \vdash [\![\mathbf{consume}\ \mathtt{r}]\!] : \mathtt{t}'} \qquad [\textsc{Dup}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t} \quad \mathtt{vec}(\mathtt{t}) \quad \mathtt{t}' = \mathtt{t}\ \&\ \mathbf{f}}{\Gamma \vdash [\![\mathbf{dup}\ \mathtt{e}]\!] : \mathtt{t}'} \qquad [\textsc{Ass}]\ \frac{\mathtt{t} = \Gamma(\mathtt{v}) \quad \Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t}' \quad \mathtt{match}(\mathtt{t}', \mathtt{t}) \quad \neg\mathtt{borrowed}(\mathtt{t})}{\Gamma \vdash [\![\mathtt{v} = \mathtt{e}]\!] : \mathtt{t}}$$

$$[\textsc{Wri}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathbf{v}(\mathbf{I})\mathbf{o!} \quad \Gamma \vdash [\![\mathtt{e}']\!] : \mathtt{t} \quad \Gamma \vdash [\![\mathtt{e}'']\!] : \mathtt{t} \quad \mathtt{t} = \mathbf{Is!}}{\Gamma \vdash [\![\mathtt{e}[\mathtt{e}'] = \mathtt{e}'']\!] : \mathtt{t}} \qquad [\textsc{Ref}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t} \quad \mathtt{prom}(\mathtt{t})}{\Gamma \vdash [\![\mathtt{e\$x}]\!] : *\mathbf{s?}}$$

$$[\textsc{RWr}]\ \frac{\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t} \quad \mathtt{prom}(\mathtt{t}) \quad \Gamma \vdash [\![\mathtt{e}']\!] : \mathtt{t}' \quad \mathtt{value}(\mathtt{t}') \quad \mathtt{shared}(\mathtt{t}')}{\Gamma \vdash [\![\mathtt{e\$x} = \mathtt{e}']\!] : \mathtt{t}'}$$

Fig. 7. Types

- An application is reflective if the version being applied is, and a call is reflective if the version being called is.
- Any expression containing reflective sub-expressions is reflective.

We define the reflection union $\mathtt{fx} \vee \mathtt{fx}'$ to be **+** if either $\mathtt{fx}$ or $\mathtt{fx}'$ is, otherwise **−**:

$$\begin{aligned} - \vee - &= - \\ - \vee + &= + \\ + \vee \mathtt{fx} &= + \end{aligned}$$

Flow Analysis. A program is well-flowed if register variables are initialized before their first access and never accessed after a **consume**. The challenge is that the point where a promise is evaluated may be difficult to know statically—in the general case it amounts to deciding reachability of every force instruction. A precise solution would involve a data-flow analysis modeling the flow of promises into force operations. We choose to approximate this analysis coarsely for the purpose of simplicity, enabling us to provide a correctness proof. An implementation may replace the analysis presented here with any sound approximation.

An *action* (R, W, U, C) is a tuple consisting of four sets of registers. It describes the registers that an expression may interact with:

- R: The registers that it may read before assigning. This does not include registers that are read after being assigned.
- W: The registers that it assigns.
- U: The registers that it may consume.
- C: The registers that it may capture, *i.e.* the registers that are read/assigned/used in a promise that it creates.

The notable restrictions that our flow analysis enforces are:

- R registers must be assigned and not used before the expression annotated with the action.
- W registers must not be used before. They don't have to be assigned before.
- C registers must not be used before or after.

U doesn't have intrinsic restrictions, although any register in U is guaranteed to be in R (so it must be assigned and not used before). U's purpose is to restrict *other* actions when it's composed with them.

$$[\textsc{RVer}]\ \frac{\Gamma = \mathtt{rdefs}, \mathtt{defs} \qquad \Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx}' \qquad \mathtt{fx}' \leq_f \mathtt{fx}}{\mathsf{R}[\![(\mathtt{rdefs}) \xrightarrow{\mathtt{fx}} \mathtt{t}\{\mathtt{defs};\ \mathtt{e}\}]\!] = \mathtt{fx}} \qquad [\textsc{RLit}]\ \frac{}{\Gamma \vdash \mathsf{R}[\![\mathtt{i}]\!] = -} \qquad [\textsc{RVar}]\ \frac{}{\Gamma \vdash \mathsf{R}[\![\mathtt{v}]\!] = -}$$

$$[\textsc{RSeq}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx} \quad \Gamma \vdash \mathsf{R}[\![\mathtt{e}']\!] = \mathtt{fx}' \quad \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'}{\Gamma \vdash \mathsf{R}[\![\mathtt{e}; \mathtt{e}']\!] = \mathtt{fx}''} \qquad [\textsc{RVec}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}_0; \ldots \mathtt{e}_n]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{R}[\![\mathbf{vec}(\mathtt{e}_0 \ldots \mathtt{e}_n)]\!] = \mathtt{fx}}$$

$$[\textsc{RApp}]\ \frac{\mathsf{R}[\![\mathtt{ver}]\!] = \mathtt{fx} \quad \Gamma \vdash \mathsf{R}[\![\mathtt{e}_0; \ldots \mathtt{e}_n]\!] = \mathtt{fx}' \quad \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'}{\Gamma \vdash \mathsf{R}[\![\mathtt{ver} \leftarrow (\mathtt{e}_0 \ldots \mathtt{e}_n)]\!] = \mathtt{fx}''}$$

$$[\textsc{RCal}]\ \frac{\mathtt{sig}(F(\mathtt{f})(\mathtt{i})) = \mathtt{t}_0 \ldots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \quad \Gamma \vdash \mathsf{R}[\![\mathtt{e}_0; \ldots \mathtt{e}_n]\!] = \mathtt{fx}' \quad \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'}{\Gamma \vdash \mathsf{R}[\![\mathtt{f.i}(\mathtt{e}_0 \ldots \mathtt{e}_n)]\!] = \mathtt{fx}''} \qquad [\textsc{RDis}]\ \frac{\mathtt{sig} = \mathtt{t}_0 \ldots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \quad \Gamma \vdash \mathsf{R}[\![\mathtt{e}_0; \ldots \mathtt{e}_n]\!] = \mathtt{fx}' \quad \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'}{\Gamma \vdash \mathsf{R}[\![\mathtt{f} \texttt{<sig>} (\mathtt{e}_0 \ldots \mathtt{e}_n)]\!] = \mathtt{fx}''}$$

$$[\textsc{RFor}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx} \quad \Gamma \vdash [\![\mathtt{e}]\!] : \mathbf{p}^{\mathtt{fx}'}(\mathtt{t}) \quad \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'}{\Gamma \vdash \mathsf{R}[\![\mathbf{force}\ \mathtt{e}]\!] = \mathtt{fx}''} \qquad [\textsc{RCas}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{R}[\![\mathtt{e}\ \mathbf{as}\ \mathtt{t}]\!] = \mathtt{fx}} \qquad [\textsc{RRefl}]\ \Gamma \vdash \mathsf{R}[\![\mathtt{r\$x}]\!] = +$$

$$[\textsc{RRea}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{R}[\![\mathtt{r}[\mathtt{e}]]\!] = \mathtt{fx}} \qquad [\textsc{RDup}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{R}[\![\mathbf{dup}\ \mathtt{e}]\!] = \mathtt{fx}} \qquad [\textsc{Use}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{R}[\![\mathbf{consume}\ \mathtt{e}]\!] = \mathtt{fx}}$$

$$[\textsc{RPro}]\ \frac{}{\Gamma \vdash \mathsf{R}[\![\mathbf{prom}^{\mathtt{fx}} \texttt{<t>} \{\mathtt{e}\}]\!] = -} \qquad [\textsc{RAss}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{R}[\![\mathtt{v} = \mathtt{e}]\!] = \mathtt{fx}} \qquad [\textsc{RWri}]\ \frac{\Gamma \vdash \mathsf{R}[\![\mathtt{e}; \mathtt{e}']\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{R}[\![\mathtt{r}[\mathtt{e}] = \mathtt{e}']\!] = \mathtt{fx}}$$

$$[\textsc{RRwr}]\ \Gamma \vdash \mathsf{R}[\![\mathtt{r\$x} = \mathtt{e}]\!] = +$$

Fig. 8. Reflection Analysis

If the following rule holds then the version is well-flowed:

$$\frac{\mathsf{F}[\![\mathsf{e}]\!] = (R, W, U, C) \qquad R \subseteq \{\mathsf{r}_0 \ldots \mathsf{r}_n\}}{\mathsf{F}[\![(\mathsf{r}_0 : \mathsf{t}_0 \ldots \mathsf{r}_n : \mathsf{t}_n) \xrightarrow{\mathsf{fx}} \mathsf{t}\{\mathsf{defs};\ \mathsf{e}\}]\!]}$$

The judgments of Fig. 9 rely on the following auxiliary definitions:

$$\begin{aligned} \mathtt{read}\ \mathsf{r} &= (\{\mathsf{r}\}, \emptyset, \emptyset, \emptyset) \\ \mathtt{write}\ \mathsf{r} &= (\emptyset, \{\mathsf{r}\}, \emptyset, \emptyset) \\ \mathtt{use}\ \mathsf{r} &= (\emptyset, \emptyset, \{\mathsf{r}\}, \emptyset) \\ \mathtt{capture}\ \mathsf{r} &= (\emptyset, \emptyset, \emptyset, \{\mathsf{r}\}) \end{aligned}$$

When two expressions are executed in sequence, the sequence's action is the composition $(;;)$ of their actions. Composition is only defined when valid, e.g. there is no way to compose an action that uses a register with an action that reads or assigns it, since a register cannot be read nor assigned after it is used.

$$[\textsc{Flit}]\ \frac{\mathsf{A} = (\emptyset, \emptyset, \emptyset, \emptyset)}{\mathsf{F}[\![\mathsf{i}]\!] = \mathsf{A}} \qquad [\textsc{Fvar}]\ \frac{\mathsf{A} = (\emptyset, \emptyset, \emptyset, \emptyset)}{\mathsf{F}[\![\mathsf{x}]\!] = \mathsf{A}} \qquad [\textsc{Fseq}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{F}[\![\mathsf{e}']\!] = \mathsf{A}' \qquad \mathsf{A}'' = \mathsf{A} ;; \mathsf{A}'}{\mathsf{F}[\![\mathsf{e}; \mathsf{e}']\!] = \mathsf{A}''}$$

$$[\textsc{Fmul}]\ \frac{\mathsf{F}[\![\mathsf{e}_0]\!] = \mathsf{A}_0 \ldots \mathsf{F}[\![\mathsf{e}_n]\!] = \mathsf{A}_n \qquad \mathsf{A}' = (\emptyset, \emptyset, \emptyset, \emptyset) ;; \mathsf{A}_0 ;; \ldots \mathsf{A}_n}{\mathsf{F}[\![\mathsf{e}_0 \ldots \mathsf{e}_n]\!] = \mathsf{A}'} \qquad [\textsc{Fvec}]\ \frac{\mathsf{F}[\![\mathsf{e}_1 \ldots \mathsf{e}_n]\!] = \mathsf{A}}{\mathsf{F}[\![\mathbf{vec}(\mathsf{e}_1 \ldots \mathsf{e}_n)]\!] = \mathsf{A}}$$

$$[\textsc{Fapp}]\ \frac{\mathsf{F}[\![\mathsf{ver}]\!] \qquad \mathsf{F}[\![\mathsf{e}_0 \ldots \mathsf{e}_n]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{ver} \leftarrow (\mathsf{e}_0 \ldots \mathsf{e}_n)]\!] = \mathsf{A}}$$

$$[\textsc{Fcal}]\ \frac{\mathsf{F}[\![\mathsf{e}_0 \ldots \mathsf{e}_n]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{f.i}(\mathsf{e}_0 \ldots \mathsf{e}_n)]\!] = \mathsf{A}} \qquad [\textsc{Fdis}]\ \frac{\mathsf{F}[\![\mathsf{e}_0 \ldots \mathsf{e}_n]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{f}\ \texttt{<sig>}\ (\mathsf{e}_0 \ldots \mathsf{e}_n)]\!] = \mathsf{A}} \qquad [\textsc{Fcas}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{e}\ \mathbf{as}\ \mathsf{t}]\!] = \mathsf{A}}$$

$$[\textsc{For}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}}{\mathsf{F}[\![\mathbf{force}\ \mathsf{e}]\!] = \mathsf{A}} \qquad [\textsc{Freg}]\ \frac{\mathsf{A} = \mathtt{read}\ \mathsf{r}}{\mathsf{F}[\![\mathsf{r}]\!] = \mathsf{A}} \qquad [\textsc{Frefl}]\ \frac{\mathsf{A} = \mathtt{read}\ \mathsf{r}}{\mathsf{F}[\![\mathsf{r\$x}]\!] = \mathsf{A}} \qquad [\textsc{Frea}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{A}' = \mathtt{read}\ \mathsf{r} ;; \mathsf{A}}{\mathsf{F}[\![\mathsf{r}[\mathsf{e}]]\!] = \mathsf{A}'}$$

$$[\textsc{Fdup}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}}{\mathsf{F}[\![\mathbf{dup}\ \mathsf{e}]\!] = \mathsf{A}} \qquad [\textsc{Fuse}]\ \frac{\mathsf{A} = \mathtt{read}\ \mathsf{r} ;; \mathtt{use}\ \mathsf{r}}{\mathsf{F}[\![\mathbf{consume}\ \mathsf{r}]\!] = \mathsf{A}} \qquad [\textsc{Fpro}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{A} = (R, W, U, C) \qquad C' = R \cup W \cup U \cup C}{\mathsf{F}[\![\mathbf{prom}^{\mathsf{fx}}\texttt{<t>}\{\mathsf{e}\}]\!] = (R, \emptyset, U, C')}$$

$$[\textsc{Fass1}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{x} = \mathsf{e}]\!] = \mathsf{A}} \qquad [\textsc{Fass2}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{A}' = \mathsf{A} ;; \mathtt{write}\ \mathsf{r}}{\mathsf{F}[\![\mathsf{r} = \mathsf{e}]\!] = \mathsf{A}'} \qquad [\textsc{Fwri}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{F}[\![\mathsf{e}']\!] = \mathsf{A}' \qquad \mathsf{A}'' = \mathtt{read}\ \mathsf{r} ;; \mathsf{A} ;; \mathsf{A}' ;; \mathtt{write}\ \mathsf{r}}{\mathsf{F}[\![\mathsf{r}[\mathsf{e}] = \mathsf{e}']\!] = \mathsf{A}''}$$

$$[\textsc{Frwr}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{A}' = \mathtt{read}\ \mathsf{r} ;; \mathsf{A} ;; \mathtt{write}\ \mathsf{r}}{\mathsf{F}[\![\mathsf{r\$x} = \mathsf{e}]\!] = \mathsf{A}'}$$

Fig. 9. Flow Analysis

$$\frac{\mathtt{A} = (R, W, U, C) \quad \mathtt{A}' = (R', W', U', C') \quad U \cap (R' \cup W' \cup U' \cup C') = \emptyset \quad C \cap U' = \emptyset}{\mathtt{A} \,;;\, \mathtt{A}' = (R \cup (R' - W), W \cup W', U \cup U', C \cup C')}$$

Our flow analysis runs on each subexpression of the body of a version and composes their actions. Analysis fails if this action cannot be computed, or if its R set is not a subset of the version's parameters. The former means there is a read/assign/capture after consume or a capture before consume; the latter means there is a read before assign.

5.2 Dynamic Semantics

We define a small-step operational semantics for FIŘ with evaluation contexts [8]. The program state is captured by a *heap* (H), mapping *references* ($\mathtt{o}$) to values ($\mathtt{0}$), written $\mathtt{o}_0 \mapsto \mathtt{0}_0 \ \ldots \ \mathtt{o}_n \mapsto \mathtt{0}_n$. References include `undef`, which does not occur on the left of a mapping.

A FIŘ expression that reduces to `undef` models a source-language error or control-flow diversion. We consider `undef` an instance of every type, so that failable operations like indexing and casting can be typed as if they cannot fail; our type system focuses on preventing stuck states, and `undef` is not stuck.

Values are defined as follows:

$\mathtt{0}$	::=	$\mathtt{i}$	integers
	|	$\mathbf{vec}(\mathtt{i}_0 \ldots \mathtt{i}_n)$	mutable vectors
	|	$\langle \mathtt{e}, \mathtt{k}, \mathtt{o} \rangle$	unevaluated promises
	|	$\langle \mathtt{o}_1, \mathtt{k}, \mathtt{o}_2 \rangle$	cached promises
	|	$(\mathtt{v}_0 \mapsto \mathtt{o}_0 \ldots \mathtt{v}_n \mapsto \mathtt{o}_n)$	environments

The syntax is extended with the following terms:

$\mathtt{e}$	::=	...	
	|	$\mathtt{o}$	references
	|	$\{\mathtt{e}\}_{\mathtt{o}}$	version scope
	|	$\{\mathtt{e}\}^{\mathtt{o}'}_{\mathtt{o}}$	promise scope

Evaluation contexts are defined as follows:

$$\begin{array}{rcl}
\mathcal{E} & ::= & [\,] \mid \mathbf{vec}(\mathtt{o}_1 \ldots \mathtt{o}_k, \mathcal{E}, \mathtt{e}_1 \ldots \mathtt{e}_n) \mid \mathcal{E}[\mathtt{e}] \mid \mathtt{o}[\mathcal{E}] \mid \mathcal{E}\$\mathtt{x} \\
& \mid & \mathtt{v} = \mathcal{E} \mid \mathcal{E}[\mathtt{e}] = \mathtt{e} \mid \mathtt{o}[\mathcal{E}] = \mathtt{e} \mid \mathtt{o}[\mathtt{o}] = \mathcal{E} \mid \mathcal{E}\$\mathtt{x} = \mathtt{e} \mid \mathtt{o}\$\mathtt{x} = \mathcal{E} \\
& \mid & \mathtt{ver} \leftarrow (\mathtt{o}_1 \ldots \mathtt{o}_k, \mathcal{E}, \mathtt{e}_1 \ldots \mathtt{e}_n) \\
& \mid & \mathtt{f.i}(\mathtt{o}_1 \ldots \mathtt{o}_k, \mathcal{E}, \mathtt{e}_1 \ldots \mathtt{e}_n) \mid \mathtt{f}\texttt{<sig>}(\mathtt{o}_1 \ldots \mathtt{o}_k, \mathcal{E}, \mathtt{e}_1 \ldots \mathtt{e}_n) \\
& \mid & \mathcal{E}\ \mathbf{as}\ \mathtt{t} \mid \mathbf{force}\,\mathcal{E} \mid \mathbf{consume}\,\mathtt{r} \mid \mathbf{dup}\,\mathcal{E} \mid \mathcal{E}\ ;\ \mathtt{e}
\end{array}$$

The relation $H[\![\mathtt{e}]\!]_{\mathtt{o}} \implies H'[\![\mathtt{o}']\!]_{\mathtt{o}}$ transforms a configuration consisting of a heap H and expression $\mathtt{e}$ in an environment referenced by $\mathtt{o}$, to an updated heap H' and expression $\mathtt{e}'$. Reduction proceeds through context:

$$H[\![\mathcal{E}[\mathtt{e}]]\!]_{\mathtt{o}} \implies H'[\![\mathcal{E}[\mathtt{e}']]\!]_{\mathtt{o}} \qquad \text{if} \qquad H[\![\mathtt{e}]\!]_{\mathtt{o}} \implies H'[\![\mathtt{e}']\!]_{\mathtt{o}}$$

The main judgments appear in Fig. 10. The semantics deal with out-of-bounds array indexing and reads of undefined named variables by producing `undef` (the rather mundane rules [DNVarUndef], [DWriUndef], [DReaUndef] and [DRrdUndef] are not shown in the figure). Furthermore, the following rule propagates `undef` to the top level:

$$H[\![\mathcal{E}[\mathtt{e}]]\!]_{\mathtt{o}} \implies H'[\![\mathcal{E}[\mathtt{undef}]]\!]_{\mathtt{o}} \qquad \text{if} \qquad H[\![\mathtt{e}]\!]_{\mathtt{o}} \implies H'[\![\mathtt{undef}]\!]_{\mathtt{o}}$$

The above rule is prioritized over the other judgments.

Several expressions allocate on the heap. Literals (DLit), promise constructors (DPro), and vector constructors (DVec) each create a fresh reference and reduce to it. A dup expression (DDup) copies its argument into a fresh reference. An application (DApp1) allocates a new environment initialized with the parameter values; this environment is released when the application returns (DApp2).

The remaining rules manipulate existing environments. A variable expression (DVar) looks up its variable in the enclosing environment; a **consume** expression (DCon) does the same but additionally removes the mapping, since the register cannot be accessed again. Assignment (DAss) maps a variable to a reference in the enclosing environment, and reflective read (DRrd) and write (DRwr) operate on the promise's captured environment instead. In contrast, a vector write (DWri) mutates the vector in-place rather than remapping a variable; the vector must be referenced at most once in the heap, otherwise the reduction is stuck.

Control flow is handled by calls, dispatch, forcing, and casts. A static call (DCal) looks up a version by function and index and reduces to an application. A dispatch call (DDis) selects the version whose signature is a subtype of the annotation at the call site and whose parameter types are satisfied by the runtime argument types. Forcing a cached promise (DFor1) returns the cached value; forcing an unevaluated promise (DFor2, DFor3) evaluates its body in the promise's environment and caches the result. A cast (DCas) evaluates to its argument if the kind matches, otherwise to `undef`. Sequences (DSeq) evaluate left to right and reduce to the last element. Rules DCtx1 and DCtx2 are mechanical context switches that evaluate version or promise code in the appropriate scope.

The rules rely on the following notation:

- `o fresh` denotes a reference `o` not occurring in the heap.
- The `env` function extracts the environment reference captured by a promise ($\mathsf{env}(\langle \_, \mathsf{k}, \mathsf{o}\rangle) = \mathsf{o}$).
- $H(\mathsf{o})$ returns the value of `o` in H; if that value is an environment, $H(\mathsf{o})(\mathsf{v})$ returns the value mapped to `v`.
- $(H, \mathsf{o} \mapsto 0)$ and $(H\ (\mathsf{o}'), \mathsf{x} \mapsto \mathsf{o}'')$ update a mapping; $H \setminus \mathsf{o}$ and $H(\mathsf{o}) \setminus \mathsf{v}$ remove one.
- `sig(ver)` returns the signature of `ver`.
- $F(\mathsf{f})(\mathsf{i})$ returns version `i` of function `f`.
- `refs` (H, o) returns the number of occurrences of `o` on the right-hand side of environment mappings in H.

The $H \vdash \mathsf{o} \leq_{dyn} \mathsf{t}$ relation is defined as follows:

$$\frac{}{H \vdash \mathsf{undef} \leq_{dyn} \mathsf{t}} \qquad \frac{H(\mathsf{o}) = \mathbf{vec}(\_) \quad \mathbf{v(I)f!} \leq \mathsf{t}\ \&\ \mathbf{f}}{H \vdash \mathsf{o} \leq_{dyn} \mathsf{t}} \qquad \frac{H(\mathsf{o}) = \mathsf{i} \quad \mathbf{If!} \leq \mathsf{t}\ \&\ \mathbf{f}}{H \vdash \mathsf{o} \leq_{dyn} \mathsf{t}} \qquad \frac{H(\mathsf{o}) = \langle \_, \mathsf{k}, \_\rangle \quad \mathsf{k}\mathbf{s!} \leq \mathsf{t}}{H \vdash \mathsf{o} \leq_{dyn} \mathsf{t}}$$

5.3 Type Safety

If a program is well-typed and well-flowed, then its reduction cannot get stuck (although it can reduce to `undef`). This section overviews the type safety proof and the invariants preserved throughout execution; a detailed proof appears in Appendix A (Theorem A.29).

Theorem (Type Safety). *If every version in the function table F is well-defined (*i.e. *well-typed and well-flowed), and if $\varnothing \vdash [\![\mathsf{f.i}()]\!] : \mathsf{t}$, then either $\varnothing[\![\mathsf{f.i}()]\!]$ diverges by the small step semantics, or it reduces to $H'[\![\mathsf{o}']\!]$ for some H' and o' such that $H' \vdash \mathsf{o}' \leq_{dyn} \mathsf{t}$.*

The proof is decomposed into two parts: one for the type checker (Appendix A.2) and one for the flow analysis (Appendix A.3). Combining their safety results yields the theorem above.

$$[\textsc{DLit}]\ \frac{\mathsf{o}'\ \mathtt{fresh} \quad H' = H, \mathsf{o}' \mapsto \mathsf{i}}{H[\![\mathsf{i}]\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DPro}]\ \frac{\mathsf{o}'\ \mathtt{fresh} \quad H' = H, \mathsf{o}' \mapsto \langle \mathsf{e}, \mathbf{p}(\mathsf{t}\ \mathsf{fx}), \mathsf{o}\rangle}{H[\![\mathbf{prom}^{\mathsf{fx}}\texttt{<t>}\{\mathsf{e}\}]\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}']\!]_{\mathsf{o}}}$$

$$[\textsc{DVec}]\ \frac{\mathsf{o}'\ \mathtt{fresh} \quad H(\mathsf{o}_1) = \mathsf{i}_1 \ldots H(\mathsf{o}_n) = \mathsf{i}_n \quad H' = H, \mathsf{o}' \mapsto \mathbf{vec}(\mathsf{i}_1 \ldots \mathsf{i}_n)}{H[\![\mathbf{vec}(\mathsf{o}_1 \ldots \mathsf{o}_n)]\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DVar}]\ \frac{\mathsf{o}' = H(\mathsf{o})(\mathsf{v})}{H[\![\mathsf{v}]\!]_{\mathsf{o}} \implies H[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DAss}]\ \frac{0 = H(\mathsf{o}), \mathsf{v} \mapsto \mathsf{o}' \quad H' = H, \mathsf{o} \mapsto 0}{H[\![\mathsf{v} = \mathsf{o}']\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}']\!]_{\mathsf{o}}}$$

$$[\textsc{DSeq}]\ \frac{}{H[\![\mathsf{o}'\ ;\ \mathsf{e}]\!]_{\mathsf{o}} \implies H[\![\mathsf{e}]\!]_{\mathsf{o}}} \qquad [\textsc{DDup}]\ \frac{\mathsf{o}''\ \mathtt{fresh} \quad H' = H, \mathsf{o}'' \mapsto H(\mathsf{o}')}{H[\![\mathbf{dup}\ \mathsf{o}']\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}'']\!]_{\mathsf{o}}} \qquad [\textsc{DCon}]\ \frac{\mathsf{o}' = H(\mathsf{o})(\mathsf{r}) \quad 0 = H(\mathsf{o}) \setminus \mathsf{r} \quad H' = H, \mathsf{o} \mapsto 0}{H[\![\mathbf{consume}\ \mathsf{r}]\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}']\!]_{\mathsf{o}}}$$

$$[\textsc{DCas}]\ \frac{H \vdash \mathsf{o} \leq_{dyn} \mathsf{t} \Rightarrow \mathsf{o}' = \mathsf{o} \quad H \vdash \mathsf{o} \not\leq_{dyn} \mathsf{t} \Rightarrow \mathsf{o}' = \mathsf{undef}}{H[\![\mathsf{o}'\ \mathbf{as}\ \mathsf{t}]\!]_{\mathsf{o}} \implies H[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DFor1}]\ \frac{H(\mathsf{o}') = \langle \mathsf{o}'', \mathsf{k}, \mathsf{o}'''\rangle}{H[\![\mathbf{force}\ \mathsf{o}']\!]_{\mathsf{o}} \implies H[\![\mathsf{o}'']\!]_{\mathsf{o}}}$$

$$[\textsc{DFor2}]\ \frac{H(\mathsf{o}') = \langle \mathsf{e}, \mathsf{k}, \mathsf{o}''\rangle \quad H' = H, \mathsf{o}' \mapsto \langle \mathsf{undef}, \mathsf{k}, \mathsf{o}''\rangle}{H[\![\mathbf{force}\ \mathsf{o}']\!]_{\mathsf{o}} \implies H'[\![\{\mathsf{e}\}^{\mathsf{o}'}_{\mathsf{o}''}]\!]_{\mathsf{o}}} \qquad [\textsc{DFor3}]\ \frac{H' = H, \mathsf{o}''' \mapsto \langle \mathsf{o}', \mathsf{k}, \mathsf{o}''\rangle}{H[\![\{\mathsf{o}'\}^{\mathsf{o}'''}_{\mathsf{o}''}]\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}']\!]_{\mathsf{o}}}$$

$$[\textsc{DDis}]\ \frac{\begin{array}{c}\text{smallest i such that} \\ \mathtt{sig}(F(\mathsf{f})(\mathsf{i})) = \mathsf{t}_0 \ldots \mathsf{t}_n \xrightarrow{\mathsf{fx}} \mathsf{t} \\ \mathsf{t}_0 \ldots \mathsf{t}_n \xrightarrow{\mathsf{fx}} \mathsf{t} \leq_{sig} \mathtt{sig} \\ H \vdash \mathsf{o}_0 \leq_{dyn} \mathsf{t}_0 \ldots H \vdash \mathsf{o}_n \leq_{dyn} \mathsf{t}_n \\ \mathsf{e} = \mathsf{f.i}(\mathsf{o}_0 \ldots \mathsf{o}_n)\end{array}}{H[\![\mathsf{f}\texttt{<sig>}(\mathsf{o}_0 \ldots \mathsf{o}_n)]\!]_{\mathsf{o}} \implies H[\![\mathsf{e}]\!]_{\mathsf{o}}} \qquad [\textsc{DApp1}]\ \frac{\begin{array}{c}\mathsf{o}'\ \mathtt{fresh} \\ \mathsf{ver} = (\mathsf{r}_0 : \mathsf{t}_0 \ldots \mathsf{r}_n : \mathsf{t}_n) \xrightarrow{\mathsf{fx}} \mathsf{t}\{\mathtt{defs};\ \mathsf{e}\} \\ 0 = \mathsf{r}_0 \mapsto \mathsf{o}_0 \ldots \mathsf{r}_n \mapsto \mathsf{o}_n \\ H' = H, \mathsf{o}' \mapsto 0\end{array}}{H[\![\mathsf{ver} \leftarrow (\mathsf{o}_0 \ldots \mathsf{o}_n)]\!]_{\mathsf{o}} \implies H'[\![\{\mathsf{e}\}_{\mathsf{o}'}]\!]_{\mathsf{o}}}$$

$$[\textsc{DApp2}]\ \frac{H' = H \setminus \mathsf{o}''}{H[\![\{\mathsf{o}'\}_{\mathsf{o}''}]\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DCal}]\ \frac{F(\mathsf{f})(\mathsf{i}) = \mathsf{ver} \quad \mathsf{e} = \mathsf{ver} \leftarrow (\mathsf{o}_0 \ldots \mathsf{o}_n)}{H[\![\mathsf{f.i}(\mathsf{o}_0 \ldots \mathsf{o}_n)]\!]_{\mathsf{o}} \implies H[\![\mathsf{e}]\!]_{\mathsf{o}}}$$

$$[\textsc{DWri}]\ \frac{\begin{array}{c}H(\mathsf{o}') = \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_n) \\ \mathtt{refs}(H, \mathsf{o}') \leq 1 \\ H(\mathsf{o}'') = \mathsf{j} \qquad H(\mathsf{o}''') = \mathsf{i} \\ 0 = \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_{j-1}, \mathsf{i}, \mathsf{i}_{j+1} \ldots \mathsf{i}_n) \\ H' = H, \mathsf{o}' \mapsto 0\end{array}}{H[\![\mathsf{o}'[\mathsf{o}''] = \mathsf{o}''']\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}''']\!]_{\mathsf{o}}} \qquad [\textsc{DRea}]\ \frac{\begin{array}{c}H(\mathsf{o}') = \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_n) \\ \mathsf{o}'''\ \mathtt{fresh} \qquad H(\mathsf{o}'') = \mathsf{j} \\ H' = H, \mathsf{o}''' \mapsto \mathsf{i}_j\end{array}}{H[\![\mathsf{o}'[\mathsf{o}'']]\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}''']\!]_{\mathsf{o}}}$$

$$[\textsc{DRrd}]\ \frac{\mathsf{o}'' = \mathsf{env}(H(\mathsf{o}')) \quad \mathsf{o}''' = H(\mathsf{o}'')(\mathsf{x})}{H[\![\mathsf{o}'\$\mathsf{x}]\!]_{\mathsf{o}} \implies H[\![\mathsf{o}''']\!]_{\mathsf{o}}} \qquad [\textsc{DRwr}]\ \frac{\mathsf{o}''' = \mathsf{env}(H, \mathsf{o}') \qquad 0 = H(\mathsf{o}'), \mathsf{x} \mapsto \mathsf{o}'' \quad H' = H, \mathsf{o}''' \mapsto 0}{H[\![\mathsf{o}'\$\mathsf{x} = \mathsf{o}'']\!]_{\mathsf{o}} \implies H'[\![\mathsf{o}'']\!]_{\mathsf{o}}}$$

$$[\textsc{DCtx1}]\ \frac{H[\![\mathsf{e}]\!]_{\mathsf{o}'} \implies H'[\![\mathsf{e}']\!]_{\mathsf{o}'}}{H[\![\{\mathsf{e}\}_{\mathsf{o}'}]\!]_{\mathsf{o}} \implies H'[\![\{\mathsf{e}'\}_{\mathsf{o}'}]\!]_{\mathsf{o}}} \qquad [\textsc{DCtx2}]\ \frac{H[\![\mathsf{e}]\!]_{\mathsf{o}'} \implies H'[\![\mathsf{e}']\!]_{\mathsf{o}'}}{H[\![\{\mathsf{e}\}^{\mathsf{o}''}_{\mathsf{o}'}]\!]_{\mathsf{o}} \implies H'[\![\{\mathsf{e}'\}^{\mathsf{o}''}_{\mathsf{o}'}]\!]_{\mathsf{o}}}$$

Fig. 10. Dynamic Semantics

Safety of the type checker. The type checker alone does not guarantee type safety: it does not prevent reading or consuming a register before initialization, or after it has been consumed. The flow analysis handles those properties. The safety theorem for the type checker therefore applies to a modified dynamic semantics with two additional rules:

$$[\textsc{DRegUndef}]\ \frac{\mathsf{r} \notin \mathrm{dom}(H(\mathsf{o}))}{H[\![\mathsf{r}]\!]_{\mathsf{o}} \implies H[\![\mathsf{undef}]\!]_{\mathsf{o}}} \qquad [\textsc{DConUndef}]\ \frac{\mathsf{r} \notin \mathrm{dom}(H(\mathsf{o}))}{H[\![\textbf{consume}\ \mathsf{r}]\!]_{\mathsf{o}} \implies H'[\![\mathsf{undef}]\!]_{\mathsf{o}}}$$

Because `undef` propagates to the top level, these rules treat an invalid register access as an unchecked runtime error rather than a stuck reduction.

The proof is decomposed into two lemmas: *type preservation* (reduction steps preserve typeability) and *progress* (a well-typed expression can always be reduced unless it is already a reference o).

Proving these lemmas requires extending the type system to handle transient expressions that arise during reduction—references o and context switches $\{\mathsf{e}\}_{\mathsf{o}'}$ and $\{\mathsf{e}\}^{\mathsf{o}''}_{\mathsf{o}'}$. The heap H is given as an additional input so that references can be typed according to it, and the type environment Γ is replaced by a mapping Δ from scopes o to type environments Γ.

Type preservation relies on several invariants that must hold throughout execution:

Typed environments Every environment in the heap H has an associated type environment in Δ.

Valid types Every environment in H is compatible with its associated type environment in Δ (i.e. its bindings have the right types).

Typeable promises For each promise in H, the captured expression is typeable and has the right type.

Valid promise scoping Promises cannot escape their scope of creation (the environment they capture must be on the call stack).

Valid borrowing Any reference in H with an owned type according to Δ cannot appear elsewhere in H, except in a more recent environment (higher on the call stack) where it has a borrowed type.

The first three invariants state that all the variables in any runtime environment must match the types of our typing environment Δ. The two last invariants are used to ensure ownership is respected, so that DWri will never fail (only vectors with a reference count of at most 1 can be assigned). In particular, the *valid promise scoping* property ensures that the scope of a function is unreachable once this function returns (it cannot be leaked in a promise).

Formal definitions appear in Appendix A.2 (Definition A.7), together with proofs of type preservation and progress.

Safety of the flow analysis. The flow analysis guarantees that registers are neither read nor consumed before initialization, nor accessed after consumption. Equivalently, for any well-flowed expression the rules DRegUndef and DConUndef never apply. Combining this with the type checker's guarantees yields type safety for the dynamic semantics of Section 5.2.

As with the type checker, correctness is decomposed into two lemmas: *preservation of well-flowedness* (a well-flowed expression remains well-flowed after reduction) and *correctness of well-flowedness* (DRegUndef and DConUndef cannot apply to well-flowed expressions).

As before, the flow analysis must be extended to handle transient expressions. Actions A are replaced by mappings $\mathbb{A}$ from scopes o to actions A, and rules are added for references and context switches. These extensions and their proofs appear in Appendix A.3.

6 Conclusion

This paper presents FIŘ, a typed high-level intermediate representation for compiling dynamic languages. FIŘ makes dynamic behaviors explicit—reflective accesses, dispatched calls, promise evaluation, and vector copies—while combining gradual typing with ownership tracking. This design enables classical optimizations such as specialization, inlining, scope elision, and copy elimination, allowing compilers to reason locally about transformations without compromising correctness.

We formalized FIŘ's operational semantics along with a static semantics comprising a type system, flow analysis, and reflection analysis, and proved type safety. Although FIŘ is informed by R's semantics, its core abstractions—tracking reflection, ownership, and delayed evaluation—apply to any dynamic language with similar features.

We are building a new JIT compiler for R using an extended variant of FIŘ. Although we do not yet have the necessary perspective to assess the long-term benefits of FIŘ, our implementation validates that FIŘ is effective in a real-world compiler: we can compile R into native code, and have straightforwardly implemented optimizations including variable elision and inlining.

Acknowledgments

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.

References

[1] John Aycock. 2003. A Brief History of Just-In-Time. *ACM Computing Surveys (CSUR)* 35, 2 (2003). doi:10.1145/857076.857077

[2] Aurele Barriere, Olivier Flückiger, Sandrine Blazy, David Pichardie, and Jan Vitek. 2021. Formally Verified Speculation and Deoptimization in a JIT Compiler. *Proc. ACM Program. Lang.* 5, POPL (2021). doi:10.1145/3434327

[3] Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). doi:10.1145/3276490

[4] Craig Chambers, David Ungar, and Elgin Lee. 1991. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. doi:10.1145/117954.117959

[5] Clifford Click. 1995. *Combining Analyses, Combining Optimizations*. Ph. D. Dissertation. Rice University. https://hdl.handle.net/1911/16837

[6] Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J. Nelson Amaral. 2023. CacheIR: The Benefits of a Structured Representation for Inline Caches. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. doi:10.1145/3617651.3622979

[7] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/800017.800542

[8] Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 2 (1992). doi:10.1016/0304-3975(92)90014-7

[9] Olivier Flückiger, Guido Chair, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). doi:10.1145/3428288

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

[11] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. *Proc. ACM Program. Lang.* 2, POPL (2018). doi:10.1145/3158137

[12] Stephen Freund and John Mitchell. 2003. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning* 30, 3 (May 2003). doi:10.1023/A:1025011624925

[13] Michael Gottesman, Joe Groff, and John McCall. 2024. `borrowing` *and* `consuming` *parameter ownership modifiers*. RFC 0377. Apple. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0377-parameter-ownership-modifiers.md

[14] Andrei Homescu and Alex Şuhan. 2011. HappyJIT: a tracing JIT compiler for PHP. In *Symposium on Dynamic Languages (DLS)*. doi:10.1145/2047849.2047854

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* (2001). doi:10.1145/503502.503505

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL] https://arxiv.org/abs/2002.11054

[17] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. The Java® Virtual Machine Specification, Java SE 8 Edition. https://docs.oracle.com/javase/specs/jvms/se8/html/

[18] Niko Matsakis. 2016. Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR/

[19] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems* 21 (1999). doi:10.1145/319301.319345

[20] James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *European Conference on Object-Oriented Programming (ECOOP)*. doi:10.1007/BFb0054091

[21] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM)*. doi:10.5555/1267847.1267848

[22] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (ECOOP)*. doi:10.1007/978-3-540-73589-2_2

[23] Tobias Wrigstad. 2006. *Ownership-Based Alias Management*. Doctoral Dissertation. Department of Information Technology, Uppsala University. https://wrigstad.com/papers/thesis.pdf

[24] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/1706299.1706343

[25] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Symposium on Dynamic Languages (DLS)*. doi:10.1145/2384577.2384587

A Proofs

A.1 Full dynamic semantics

The full small-step semantics is the same as in Section 5.2, with the two extra rules [DRegUndef] and [DConUndef] (in blue). Reduction steps are written $H[\![\mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{e}']\!]_{\mathsf{o}}$. We write $H[\![\mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow^{*} H'[\![\mathsf{e}']\!]_{\mathsf{o}}$ for a reduction of any number of steps, and $H[\![\mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow^{\infty}$ for a diverging reduction.

$$[\textsc{DLit}]\ \frac{\mathsf{o}'\ \mathtt{fresh} \quad H' = H, \mathsf{o}' \mapsto \mathsf{i}}{H[\![\mathsf{i}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DPro}]\ \frac{\mathsf{o}'\ \mathtt{fresh} \quad H' = H, \mathsf{o}' \mapsto \langle \mathsf{e}, \mathbf{p}(\mathsf{t}\ \mathsf{fx}), \mathsf{o}\rangle}{H[\![\mathbf{prom}^{\mathsf{fx}}\texttt{<t>}\{\mathsf{e}\}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}']\!]_{\mathsf{o}}}$$

$$[\textsc{DVec}]\ \frac{\mathsf{o}'\ \mathtt{fresh} \quad H(\mathsf{o}_1) = \mathsf{i}_1 \ldots H(\mathsf{o}_n) = \mathsf{i}_n \quad H' = H, \mathsf{o}' \mapsto \mathbf{vec}(\mathsf{i}_1 \ldots \mathsf{i}_n)}{H[\![\mathbf{vec}(\mathsf{o}_1 \ldots \mathsf{o}_n)]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}']\!]_{\mathsf{o}}}$$

$$[\textsc{DVar}]\ \frac{\mathsf{o}' = H(\mathsf{o})(\mathsf{v})}{H[\![\mathsf{v}]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DNVarUndef}]\ \frac{\mathsf{x} \notin \mathrm{dom}(H(\mathsf{o}))}{H[\![\mathsf{x}]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{undef}]\!]_{\mathsf{o}}} \qquad [\textsc{DConUndef}]\ \frac{\mathsf{r} \notin \mathrm{dom}(H(\mathsf{o}))}{H[\![\mathbf{consume}\ \mathsf{r}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{undef}]\!]_{\mathsf{o}}}$$

$$[\textsc{DDup}]\ \frac{\mathsf{o}''\ \mathtt{fresh} \quad H' = H, \mathsf{o}'' \mapsto H(\mathsf{o}')}{H[\![\mathbf{dup}\ \mathsf{o}']\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}'']\!]_{\mathsf{o}}} \qquad [\textsc{DCon}]\ \frac{\mathsf{o}' = H(\mathsf{o})(\mathsf{r}) \quad H' = H, \mathsf{o} \mapsto H(\mathsf{o}) \setminus \mathsf{r}}{H[\![\mathbf{consume}\ \mathsf{r}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DRegUndef}]\ \frac{\mathsf{r} \notin \mathrm{dom}(H(\mathsf{o}))}{H[\![\mathsf{r}]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{undef}]\!]_{\mathsf{o}}}$$

$$[\textsc{DFor1}]\ \frac{H(\mathsf{o}') = \langle \mathsf{o}'', \mathsf{k}, \mathsf{o}'''\rangle}{H[\![\mathbf{force}\ \mathsf{o}']\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{o}'']\!]_{\mathsf{o}}} \qquad [\textsc{DFor2}]\ \frac{H(\mathsf{o}') = \langle \mathsf{e}, \mathsf{k}, \mathsf{o}''\rangle \quad H' = H, \mathsf{o}' \mapsto \langle \mathsf{undef}, \mathsf{k}, \mathsf{o}''\rangle}{H[\![\mathbf{force}\ \mathsf{o}']\!]_{\mathsf{o}} \rightsquigarrow H'[\![\{\mathsf{e}\}^{\mathsf{o}'}_{\mathsf{o}''}]\!]_{\mathsf{o}}} \qquad [\textsc{DFor3}]\ \frac{H' = H, \mathsf{o}''' \mapsto \langle \mathsf{o}', \mathsf{k}, \mathsf{o}''\rangle}{H[\![\{\mathsf{o}'\}^{\mathsf{o}'''}_{\mathsf{o}''}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}']\!]_{\mathsf{o}}}$$

$$[\textsc{DApp1}]\ \frac{\mathsf{o}'\ \mathtt{fresh} \qquad \mathsf{ver} = (\mathsf{r}_0 : \mathsf{t}_0 \ldots \mathsf{r}_n : \mathsf{t}_n) \xrightarrow{\mathsf{fx}} \mathsf{t}\{\mathtt{defs};\ \mathsf{e}\} \quad 0 = \mathsf{r}_0 \mapsto \mathsf{o}_0 \ldots \mathsf{r}_n \mapsto \mathsf{o}_n \qquad H' = H, \mathsf{o}' \mapsto 0}{H[\![\mathsf{ver} \leftarrow (\mathsf{o}_0 \ldots \mathsf{o}_n)]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\{\mathsf{e}\}_{\mathsf{o}'}]\!]_{\mathsf{o}}} \qquad [\textsc{DApp2}]\ \frac{H' = H \setminus \mathsf{o}''}{H[\![\{\mathsf{o}'\}_{\mathsf{o}''}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}']\!]_{\mathsf{o}}}$$

$$[\textsc{DCal}]\ \frac{F(\mathsf{f})(\mathsf{i}) = \mathsf{ver} \quad \mathsf{e} = \mathsf{ver} \leftarrow (\mathsf{o}_0 \ldots \mathsf{o}_n)}{H[\![\mathsf{f.i}(\mathsf{o}_0 \ldots \mathsf{o}_n)]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{e}]\!]_{\mathsf{o}}} \qquad [\textsc{DDis}]\ \frac{\begin{array}{c}\text{smallest i such that} \\ \mathsf{sig}(F(\mathsf{f})(\mathsf{i})) = \mathsf{t}_0 \ldots \mathsf{t}_n \xrightarrow{\mathsf{fx}} \mathsf{t} \qquad \mathsf{t}_{\emptyset} \ldots \mathsf{t}_n \xrightarrow{\mathsf{fx}} \mathsf{t} \leq_{sig} \mathsf{sig} \\ H \vdash \mathsf{o}_0 \leq_{dyn} \mathsf{t}_0 \ldots H \vdash \mathsf{o}_n \leq_{dyn} \mathsf{t}_n \qquad \mathsf{e} = \mathsf{f.i}(\mathsf{o}_0 \ldots \mathsf{o}_n)\end{array}}{H[\![\mathsf{f}\texttt{<sig>}(\mathsf{o}_0 \ldots \mathsf{o}_n)]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{e}]\!]_{\mathsf{o}}}$$

$$[\textsc{DWriUndef}]\ \frac{H(\mathsf{o}_1) = \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_n) \quad H(\mathsf{o}_2) = \mathsf{j} \qquad \mathsf{j} < 0 \text{ or } \mathsf{j} > \mathsf{n}}{H[\![\mathsf{o}_1[\mathsf{o}_2] = \mathsf{o}_3]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{undef}]\!]_{\mathsf{o}}} \qquad [\textsc{DWri}]\ \frac{\begin{array}{c}H(\mathsf{o}') = \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_n) \qquad \mathsf{refs}(H, \mathsf{o}') \leq 1 \\ H(\mathsf{o}'') = \mathsf{j} \qquad H(\mathsf{o}''') = \mathsf{i} \\ 0 = \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_{\mathsf{j}-1}, \mathsf{i}, \mathsf{i}_{\mathsf{j}+1} \ldots \mathsf{i}_n) \qquad H' = H, \mathsf{o}' \mapsto 0\end{array}}{H[\![\mathsf{o}'[\mathsf{o}''] = \mathsf{o}''']\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}''']\!]_{\mathsf{o}}}$$

$$[\textsc{DReaUndef}]\ \frac{H(\mathsf{o}_1) = \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_n) \quad H(\mathsf{o}_2) = \mathsf{j} \qquad \mathsf{j} < 0 \text{ or } \mathsf{j} > \mathsf{n}}{H[\![\mathsf{o}_1[\mathsf{o}_2]]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{undef}]\!]_{\mathsf{o}}} \qquad [\textsc{DRea}]\ \frac{H(\mathsf{o}') = \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_n) \quad \mathsf{o}'''\ \mathtt{fresh} \quad H(\mathsf{o}'') = \mathsf{j} \qquad H' = H, \mathsf{o}''' \mapsto \mathsf{i}_{\mathsf{j}}}{H[\![\mathsf{o}'[\mathsf{o}'']]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}''']\!]_{\mathsf{o}}}$$

$$[\textsc{DAss}]\ \frac{0 = H(\mathsf{o}), \mathsf{v} \mapsto \mathsf{o}' \quad H' = H, \mathsf{o} \mapsto 0}{H[\![\mathsf{v} = \mathsf{o}']\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DRrd}]\ \frac{\mathsf{o}'' = \mathsf{env}(H(\mathsf{o}')) \quad \mathsf{o}''' = H(\mathsf{o}'')(\mathsf{x})}{H[\![\mathsf{o}'\$\mathsf{x}]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{o}''']\!]_{\mathsf{o}}} \qquad [\textsc{DRwr}]\ \frac{\mathsf{o}''' = \mathsf{env}(H, \mathsf{o}') \qquad 0 = H(\mathsf{o}'), \mathsf{x} \mapsto \mathsf{o}'' \quad H' = H, \mathsf{o}''' \mapsto 0}{H[\![\mathsf{o}'\$\mathsf{x} = \mathsf{o}'']\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{o}'']\!]_{\mathsf{o}}}$$

$$[\textsc{DCas}]\ \frac{\begin{array}{c}H \vdash \mathsf{o} \leq_{dyn} \mathsf{t} \Rightarrow \mathsf{o}' = \mathsf{o} \\ H \vdash \mathsf{o} \not\leq_{dyn} \mathsf{t} \Rightarrow \mathsf{o}' = \mathsf{undef}\end{array}}{H[\![\mathsf{o}'\ \mathbf{as}\ \mathsf{t}]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{o}']\!]_{\mathsf{o}}} \qquad [\textsc{DSeq}]\ \frac{}{H[\![\mathsf{o}'\ ;\ \mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow H[\![\mathsf{e}]\!]_{\mathsf{o}}} \qquad [\textsc{DCtx1}]\ \frac{H[\![\mathsf{e}]\!]_{\mathsf{o}'} \rightsquigarrow H'[\![\mathsf{e}']\!]_{\mathsf{o}'}}{H[\![\{\mathsf{e}\}_{\mathsf{o}'}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\{\mathsf{e}'\}_{\mathsf{o}'}]\!]_{\mathsf{o}}}$$

$$[\textsc{DCtx2}]\ \frac{H[\![\mathsf{e}]\!]_{\mathsf{o}'} \rightsquigarrow H'[\![\mathsf{e}']\!]_{\mathsf{o}'}}{H[\![\{\mathsf{e}\}^{\mathsf{o}''}_{\mathsf{o}'}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\{\mathsf{e}'\}^{\mathsf{o}''}_{\mathsf{o}'}]\!]_{\mathsf{o}}} \qquad [\textsc{DCtx3}]\ \frac{H[\![\mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{undef}]\!]_{\mathsf{o}}}{H[\![\mathcal{E}[\mathsf{e}]]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{undef}]\!]_{\mathsf{o}}} \qquad [\textsc{DCtx4}]\ \frac{H[\![\mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{e}']\!]_{\mathsf{o}}}{H[\![\mathcal{E}[\mathsf{e}]]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathcal{E}[\mathsf{e}']]\!]_{\mathsf{o}}}$$

A.2 Proofs for the static semantics

We prove type safety for the small-step semantics defined above. Due to the rules [DRegUndef] and [DConUndef], this theorem does not ensure correct register initialization; that property is established by the flow analysis (Section A.3).

Definition A.1 (Well-typed version). A version ver is well-typed if and only if there exists $t_0 \dots t_n, t$ such that $[\![ver]\!] : t_0 \dots t_n \xrightarrow{fx} t$ holds.

In the following, we assume that every version ver in our function table F is well-typed.

The proof proceeds via a type preservation lemma (Lemma A.9) and a progress lemma (Lemma A.10). Both require extending the type system to handle intermediate results—references o, version scopes $\{e\}_o$, and promise scopes $\{e\}_o^{o'}$—by taking the current heap and a heap descriptor as additional inputs:

Definition A.2 (Heap descriptor). We call heap descriptor Δ a partial mapping from references o to type environments Γ.

Definition A.3 (Dynamic kind). We define the dynamic kind of a runtime value 0 as follows:

$$\mathtt{dynkind}(i) = \mathbf{I} \qquad \mathtt{dynkind}(\mathbf{vec}(i_0 \dots i_n)) = \mathbf{v}(\mathbf{I}) \qquad \mathtt{dynkind}(\langle \_, k, \_ \rangle) = k$$

$$[\textsc{Var}]\ \frac{t = \Delta(o)(v)}{\Delta, H \vdash_o [\![v]\!] : t} \qquad [\textsc{Undef}]\ \frac{}{\Delta, H \vdash_o [\![\mathtt{undef}]\!] : t}$$

$$[\textsc{Dyn}]\ \frac{\begin{array}{c} k = \mathtt{dynkind}(H(o')) \qquad H(o') = \langle \_, \_, o'' \rangle \Rightarrow o'' \in \mathrm{dom}(H) \\ ox = \mathbf{s} \Rightarrow \mathtt{owner}(H, \Delta, o') = \{\} \qquad ox = \mathbf{o} \Rightarrow \mathtt{owner}(H, \Delta, o') = \{o\} \qquad ox = \mathbf{f} \Rightarrow o'\ \mathtt{fresh} \end{array}}{\Delta, H \vdash_o [\![o']\!] : k\ ox\ \mathbf{!}}$$

$$[\textsc{ASc}]\ \frac{\begin{array}{c} \Delta, H \vdash_{o'} [\![e]\!] : t' \\ t = \begin{cases} t' & \text{if } \mathtt{shared}(t') \text{ or } \mathtt{fresh}(t') \\ t' \,\&\, \mathbf{f} & \text{if } \mathtt{owned}(t') \end{cases} \\ \mathtt{value}(t) \qquad \mathtt{wf}(t) \end{array}}{\Delta, H \vdash_o [\![\{e\}_{o'}]\!] : t} \qquad [\textsc{PSc}]\ \frac{\begin{array}{c} \Delta, H \vdash_{o'} [\![e]\!] : t \\ \mathtt{value}(t) \qquad \mathtt{shared}(t) \end{array}}{\Delta, H \vdash_o [\![\{e\}_{o'}^{o''}]\!] : t}$$

where $\mathtt{owner}(H, \Delta, o) = \{o' \mid \exists r.\ H(o')(r) = o \text{ and } \mathtt{owned}(\Delta(o')(r))\}$ and $o'\ \mathtt{fresh}$ holds when o' does not appear in H except on the left-hand side of a binding, and has no other occurrence in the expression.

The remaining typing rules carry H and o recursively and are otherwise identical to those in Section 5.1. The rule [Ver] needs no additional parameters: since we do not reduce under versions, no reference, version scope, or promise scope can appear in a version body, so [Ver] is reused as is:

$$[\textsc{Ver}]\ \frac{\begin{array}{c} \Gamma = \mathtt{rdefs}, \mathtt{defs} \qquad \mathtt{wf}(\Gamma) \qquad \Gamma \vdash [\![e]\!] : t' \qquad \mathtt{types}(\mathtt{rdefs}) = t_0 \dots t_n \qquad \Gamma \vdash \mathtt{R}[\![e]\!] = fx \\ t = \begin{cases} t' & \text{if } \mathtt{shared}(t') \text{ or } \mathtt{fresh}(t') \\ t' \,\&\, \mathbf{f} & \text{if } \mathtt{owned}(t') \end{cases} \\ \mathtt{value}(t) \qquad \mathtt{wf}(t) \end{array}}{[\![(\mathtt{rdefs}) \xrightarrow{fx} t\{\mathtt{defs};\ e\}]\!] : t_0 \dots t_n \xrightarrow{fx} t}$$

Proposition A.4. *If $\Gamma \vdash [\![e]\!] : t$ (typing judgement from Section 5.1), then for any o, H, Δ such that $\Delta(o) = \Gamma$, we have $\Delta, H \vdash_o [\![e]\!] : t$.*

Proof. Straightforward. □

Definition A.5 (Dynamic/static environment matching). Let H be a heap, o a reference, Γ a type environment. We say that H matches Γ at reference o if and only if:

- $H(\mathsf{o}) = \mathtt{E}$.
- $\forall \mathsf{r} \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\mathtt{E}).\ \Gamma(\mathsf{r})\ \&\ ! = \Gamma(\mathsf{r}) \Rightarrow \mathtt{dynkind}(\mathtt{E}(\mathsf{r})) \leq \mathtt{kind}(\Gamma(\mathsf{r}))$.
- $\forall \mathsf{r} \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\mathtt{E}).\ \mathtt{owned}(\Gamma(\mathsf{r})) \Rightarrow \mathtt{refs}(H, \mathtt{E}(\mathsf{r})) \leq 1$.

We define call contexts C as follows:

$$C \quad ::= \quad \ldots \quad | \quad \{C\}^{\mathsf{o}}_{\mathsf{o}}$$

where $\ldots$ includes all the cases of the grammar of $\mathcal{E}$ (defined in Section 5.2).

Definition A.6 (Call stack). Let e be an expression. The call stack of e is the longest sequence of references $\mathsf{o}_1, ..., \mathsf{o}_n$ such that $\mathsf{e} = C_1[\{\ldots C_n[\{\mathsf{e}'\}_{\mathsf{o}_n}]\ldots\}_{\mathsf{o}_1}]$ for some $C_1, ..., C_n$ and e'.

A call stack (a sequence of references) is written $\vec{\mathsf{o}}$. The concatenation of two call stacks $\vec{\mathsf{o}}$ and $\vec{\mathsf{o}}'$ is written $\vec{\mathsf{o}}, \vec{\mathsf{o}}'$.

Definition A.7 (Heap validity). The heap H is valid for the heap descriptor Δ and call stack $\mathsf{o}_1, ..., \mathsf{o}_n$ if and only if:

Typed environments For every binding $(\mathsf{o} \mapsto \mathtt{E}) \in H$, there exists a unique $\mathtt{i}$ such that $\mathsf{o}_{\mathtt{i}} = \mathsf{o}$.

Valid types For every $\mathtt{i}$, $H \setminus \{\mathsf{o}_{\mathtt{i}+1}, ..., \mathsf{o}_n\}$ matches $\Delta(\mathsf{o}_{\mathtt{i}})$ at reference $\mathsf{o}_{\mathtt{i}}$.

Typeable promises For every $(\mathsf{o}'' \mapsto \langle \mathsf{e}, \mathsf{k}, \mathsf{o} \rangle) \in H$, if $\mathsf{o} \in \mathrm{dom}(H)$ then $\mathsf{k} = \mathbf{p}^{\mathsf{fx}}(\mathsf{k}'\ \mathbf{s}\ \mathsf{cx})$ for some k', cx and fx, and $\Delta, H \vdash_{\mathsf{o}} [\![\mathsf{e}]\!] : \mathsf{k}'\ \mathbf{s}\ \mathsf{cx}$.

Valid promise scoping For every $\mathtt{i}$, for every $(\mathsf{v} \mapsto \mathsf{o}') \in H(\mathsf{o}_{\mathtt{i}})$ with $H(\mathsf{o}') = \langle \_, \mathsf{k}, \mathsf{o}'' \rangle$, we have $\mathsf{o}'' = \mathsf{o}_{\mathtt{i}'}$ for some $\mathtt{i}' \leq \mathtt{i}$.

Valid borrowing For every $\mathtt{i}$ and $\mathtt{i}' > \mathtt{i}$, for every $\mathsf{r} \in \mathrm{dom}(\Delta(\mathsf{o}_{\mathtt{i}})) \cap \mathrm{dom}(H(\mathsf{o}_{\mathtt{i}}))$, for every $\mathsf{v} \in \mathrm{dom}(H(\mathsf{o}_{\mathtt{i}'}))$: $(\mathtt{owned}(\Delta(\mathsf{o}_{\mathtt{i}})(\mathsf{r}))$ and $(H(\mathsf{o}_{\mathtt{i}})(\mathsf{r}) = H(\mathsf{o}_{\mathtt{i}'})(\mathsf{v})))$ $\Rightarrow \mathtt{borrowed}(\Delta(\mathsf{o}_{\mathtt{i}'})(\mathsf{v}))$.

Definition A.8 (Quadruplet validity). A quadruplet $(H, \Delta, \vec{\mathsf{o}}, \mathsf{e})$ is valid if and only if H is valid for the heap descriptor Δ and call stack $(\vec{\mathsf{o}}, \vec{\mathsf{o}}')$, with $\vec{\mathsf{o}}'$ being the call stack of e.

Lemma A.9 (Preservation of type and validity). *Let $(H, \Delta, \vec{\mathsf{o}}, \mathsf{e})$ be a valid quadruplet. If $\Delta, H \vdash_{\mathsf{o}} [\![\mathsf{e}]\!] : \mathtt{t}$ and $H[\![\mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{e}']\!]_{\mathsf{o}}$, then there exists Δ' such that $\Delta', H' \vdash_{\mathsf{o}} [\![\mathsf{e}']\!] : \mathtt{t}$ and $(H', \Delta', \vec{\mathsf{o}}, \mathsf{e}')$ is valid.*

Proof. We proceed by structural induction on the derivation $\Delta, H \vdash_{\mathsf{o}} [\![\mathsf{e}]\!] : \mathtt{t}$.

If the root of the derivation is a [Sub], we can conclude by using the induction hypothesis.

Other rules are structural, thus we can match on the syntax of e (we only include the most interesting cases):

o' Impossible case as no reduction step can apply on o'.

$\{\mathsf{o}_1\}_{\mathsf{o}_2}$ The reduction rule [DApp2] applies, we thus have $\mathsf{e}' = \mathsf{o}_1$ and $H' = H \setminus \mathsf{o}_2$. We can conclude with a [Dyn] rule as $\mathsf{o}_1 \neq \mathsf{o}_2$ ($H(\mathsf{o}_2)$ is an environment $\mathtt{E}$, while $H(\mathsf{o}_1)$ is not). Note that we use the *valid promise scoping* property to ensure the new heap $H \setminus \mathsf{o}_2$ does not have accessible "orphan" promises.

$\{\mathsf{e}_1\}_{\mathsf{o}_2}$ The reduction rule [DCtx1] applies, we conclude by applying the induction hypothesis on e_1.

$\{\mathsf{o}_1\}^{\mathsf{o}_3}_{\mathsf{o}_2}$ The reduction rule [DFor3] applies, we thus have $\mathsf{e}' = \mathsf{o}_1$ and $H' = H, \mathsf{o}_3 \mapsto \langle \mathsf{o}_1, \mathsf{k}, \mathsf{o}_2 \rangle$, we can thus conclude with a [Dyn] rule.

$\{e_1\}^{o_3}_{o_2}$ The reduction rule [DCtx2] applies, we conclude by applying the induction hypothesis on e_1.

v If the reduction rule [DVar1] applies, we can thus conclude with a [Dyn] rule (using the *valid types* property and the *valid promise scoping* property). Otherwise, if the rule [DVar2] or [DVar3] applies, we conclude with a [Undef] rule.

o′ **as** t′ The reduction rule [DCas] applies. If the dynamic cast failed, we get $e' = \texttt{undef}$, in which case we can conclude with the rule [Undef]. Otherwise, from the typing derivation of e, we can extract a typing derivation for o′ of a type t″ of the same ownership as t′. As the cast succeeded, we know that $\texttt{dynkind}(H(o')) \leq \texttt{kind}(t')$, and we can thus derive the type t′ for o′.

force o′ The reduction rule [DFor1] or [DFor2] applies. In both cases, we can conclude using the *typeable promises* property.

ver ← (o₀ … oₙ) The rule [DApp1] applies. We must type the body of ver, which can easily be done by extracting the subderivation of the body from $[\![\texttt{ver}]\!] : t_0 \dots t_n \xrightarrow{fx} t$ and applying Proposition A.4 to type the body under the new heap descriptor $\Delta, o' \mapsto \Gamma$ with $\Gamma = \texttt{rdefs}, \texttt{defs}$.

f.i(o₀ … oₙ) The rule [DCal] applies. We can conclude using the fact that versions in F are well-typed, and using Proposition A.4.

f <sig> (o₀ … oₙ) The rule [DDis] applies. Using the fact that the version selected by the [DDis] rule has a smaller signature than the version used in our typing derivation, this case can be concluded similarly to the previous case.

o′$x = o″ The rule [DRwr] applies. The typing derivation of e gives the guarantee that o″ is not a promise, which can be used to guarantee the preservation of the *valid promise scoping* property, and that it is shared, which, thanks to the *valid borrowing* property, can be used to guarantee that the *valid types* property is preserved.

□

Lemma A.10 (Progress). *Let* $(H, \Delta, \vec{o}, e)$ *be a valid quadruplet. If* $\Delta, H \vdash_o [\![e]\!] : t$, *then either* e *is a reference* o′ *or* $H[\![e]\!]_o \rightsquigarrow H'[\![e']\!]_o$ *for some* H' *and* e′.

Proof. We proceed by structural induction on the derivation $\Delta, H \vdash_o [\![e]\!] : t$.

If the root of the derivation is a [Sub], we can conclude by using the induction hypothesis.

Other rules are structural, thus we can match on the syntax of e (we only include the most interesting cases):

o′ Trivial.

$o_1[o_2] = o_3$ Our typing derivation must end with a [Wri] rule, which requires o_1 to have type **v(I)o!**. It yields that $\texttt{dynkind}(H(o_1)) \leq \textbf{v(I)}$ and thus $H(o_1)$ is a vector. From $\texttt{owner}(H, \Delta, o') = \{o\}$ and the *valid types* property, we can deduce $\texttt{refs}(H, o') \leq 1$, which allows to conclude that either the rule [DWri] or [DWriUndef] applies.

force o′ The typing derivation gives a $\textbf{p}^{fx}(t)\textbf{s!}$ type to o′, which ensures that $H(o')$ is a promise. Thus, either [DFor1] or [DFor2] can be applied.

force e′ The typing derivation gives a $\textbf{p}^{fx}(t)\textbf{s!}$ type to e′, we can thus apply the induction hypothesis and conclude with a [DCtx3] or [DCtx4] rule.

f <sig> (o₀ … oₙ) The typing derivation gives the types $t'_0 \dots t'_n$ to the arguments, with $t'_0 \leq t_0, \dots, t'_n \leq t_n$ and $\texttt{sig}(F(f)(i)) = t_0 \dots t_n \xrightarrow{fx} t'$ for some i and fx. Thus, using the *valid types* property, we know that the rule [DDis] can be applied on the version i (or on a version at a smaller position if applicable).

□

Theorem A.11 (Type safety).
If $\varnothing \vdash [\![\texttt{f.i()}]\!] : \texttt{t}$, *then either* $\varnothing[\![\texttt{f.i()}]\!]_{\texttt{o}} \rightsquigarrow^{\infty}$ *or* $\varnothing[\![\texttt{f.i()}]\!]_{\texttt{o}} \rightsquigarrow^{*} H'[\![\texttt{o}']\!]_{\texttt{o}}$ *for some* H' *and* $\texttt{o}'$ *such that* $H' \vdash \texttt{o}' \leq_{dyn} \texttt{t}$.

Proof. Immediate consequence of Proposition A.4, of the preservation (Lemma A.9) and progress (Lemma A.10). □

We recall that this theorem, as well as all the lemmas in the section, assume that every version ver in our function table F is well-typed.

A.3 Proofs for the flow analysis

We prove that the flow analysis ensures correct register initialization: if an expression is well-flowed, the rules [DRegUndef] and [DConUndef] cannot apply after any number of steps. Combined with the type safety of the previous section, this yields type safety for the dynamic semantics of Section 5.2, where accessing an uninitialized register results in a stuck reduction.

Definition A.12 (Well-flowed version). A version ver is well-flowed if and only if $\texttt{F}[\![\texttt{ver}]\!]$ holds.

Definition A.13 (Well-defined version). A version ver is well-defined if and only if ver is well-typed and well-flowed.

In the following, we assume that every version ver in our function table F is well-defined.

We first generalize the flow analysis so that it applies to intermediate results of a reduction, not only source expressions. We then prove that the action returned by this generalized analysis is preserved by reduction (Lemma A.25), and that a well-flowed expression cannot be reduced using [DRegUndef] or [DConUndef] (Lemma A.27).

Proposition A.14 (Associativity of composition). *The composition operator* ;; *is associative.*

Proof. Let $\texttt{A}_1 = (R_1, W_1, U_1, C_1)$, $\texttt{A}_2 = (R_2, W_2, U_2, C_2)$, and $\texttt{A}_3 = (R_3, W_3, U_3, C_3)$. The expressions $(\texttt{A}_1 \;;;\; \texttt{A}_2) \;;;\; \texttt{A}_3$ and $\texttt{A}_1 \;;;\; (\texttt{A}_2 \;;;\; \texttt{A}_3)$ are both defined if and only if:

- $U_1 \cap (R_2 \cup W_2 \cup U_2 \cup C_2) = \varnothing$, and
- $(U_1 \cup U_2) \cap (R_3 \cup W_3 \cup U_3 \cup C_3) = \varnothing$, and
- $C_1 \cap U_2 = \varnothing$, and
- $(C_1 \cup C_2) \cap U_3 = \varnothing$

and when defined, are equal to the expression (R', W', U', C') where:

- $R' = R_1 \cup (R_2 \setminus W_1) \cup (R_3 \setminus (W_1 \cup W_2))$
- $W' = W_1 \cup W_2 \cup W_3$
- $U' = U_1 \cup U_2 \cup U_3$
- $C' = C_1 \cup C_2 \cup C_3$

□

Definition A.15. We define a partial order $\leq$ over actions as follows:

$$(R, W, U, C) \leq (R', W', U', C') \Leftrightarrow R \subseteq R', W' \subseteq W, U \subseteq U', C \subseteq C'$$

Proposition A.16 (Monotonicity of composition). *The composition operator* ;; *is monotonic with respect to the order* $\leq$.

Proof. Straightforward. □

Definition A.17 (Heap actions). A heap action $\mathbb{A}$ is a total mapping from references o to actions A. We write $\emptyset$ for the heap action mapping every reference to $(\varnothing, \varnothing, \varnothing, \varnothing)$, and $(\texttt{o} \mapsto \texttt{A})$ for the heap action mapping o to A and every other reference to $(\varnothing, \varnothing, \varnothing, \varnothing)$.

We extend the composition $;;$ to work on heap actions: $\forall \mathsf{o}.\ (\mathbb{A} ;;\ \mathbb{A}')(\mathsf{o}) = \mathbb{A}(\mathsf{o}) ;;\ \mathbb{A}'(\mathsf{o})$. We also extend the order $\leq$: $\mathbb{A} \leq \mathbb{A}' \iff \forall \mathsf{o}.\ \mathbb{A}(\mathsf{o}) \leq \mathbb{A}'(\mathsf{o})$. The properties above about $;;$ still hold.

The flow analysis of Section 5.1 is extended. It now takes a reference o as additional input, and returns heap actions:

$$[\text{FDyn}]\ \frac{}{\mathsf{F}[\![\mathsf{o}']\!]_{\mathsf{o}} = \emptyset} \qquad [\text{FASc}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!]_{\mathsf{o}'} = \mathbb{A}}{\mathsf{F}[\![\{\mathsf{e}\}_{\mathsf{o}'}]\!]_{\mathsf{o}} = \mathbb{A}} \qquad [\text{FPSc}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!]_{\mathsf{o}'} = \mathbb{A}}{\mathsf{F}[\![\{\mathsf{e}\}^{\mathsf{o}''}_{\mathsf{o}'}]\!]_{\mathsf{o}} = \mathbb{A}}$$

The remaining rules carry o recursively and lift each action A to $\mathsf{o} \mapsto \mathsf{A}$; for instance:

$$[\text{FReg}]\ \frac{\mathbb{A} = \mathsf{o} \mapsto \mathtt{read}\ \mathsf{r}}{\mathsf{F}[\![\mathsf{r}]\!]_{\mathsf{o}} = \mathbb{A}}$$

The rule [FVer] needs no additional parameters: since we do not reduce under versions, no reference, version scope, or promise scope can appear in a version body, so [FVer] from Section 5.1 is reused as is:

$$[\text{FVer}]\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = (R, W, U, C) \qquad R \subseteq \{\mathsf{r}_0 \ldots \mathsf{r}_n\}}{\mathsf{F}[\![(\mathsf{r}_0 : \mathsf{t}_0 \ldots \mathsf{r}_n : \mathsf{t}_n) \xrightarrow{\mathsf{fx}} \mathsf{t}\{\mathtt{defs}; \mathsf{e}\}]\!]}$$

Proposition A.18. *If* $\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}$*, then for any reference* o*, we have* $\mathsf{F}[\![\mathsf{e}]\!]_{\mathsf{o}} = \mathbb{A}$ *with* $\mathbb{A} = \mathsf{o} \mapsto \mathsf{A}$.

Proof. Follows from the definition. □

Definition A.19 (Delayed action). For an action $\mathsf{A} = (R, W, U, C)$, we define the action $\mathtt{delay}(\mathsf{A})$ as follows:

$$\mathtt{delay}(\mathsf{A}) = (R, \emptyset, U, R \cup W \cup U \cup C)$$

Proposition A.20. *For any action* A, $\mathsf{A} \leq \mathtt{delay}(\mathsf{A})$.

Proof. Straightforward. □

Delaying some actions is a way to make them commutative, as stated by the proposition below.

Proposition A.21 (Commutativity of composition on delayed actions). *For any actions* A_1 *and* A_2*, we have* $\mathtt{delay}(A_1) ;; \mathtt{delay}(A_2) = \mathtt{delay}(A_2) ;; \mathtt{delay}(A_1)$.

Proof. Straightforward. □

Definition A.22 (Pending actions). For a heap H, the pending action $\mathtt{pending}(H)$ is the heap action recursively defined as follows (cases by order of decreasing priority):

$$\begin{aligned} \mathtt{pending}(\emptyset) &= \emptyset \\ \mathtt{pending}(H, \mathsf{o} \mapsto \langle \mathsf{e}, \mathsf{k}, \mathsf{o}' \rangle) &= \mathtt{pending}(H) ;;\ (\mathsf{o}' \mapsto \mathtt{delay}(\mathsf{F}[\![\mathsf{e}]\!])) \\ \mathtt{pending}(H, \mathsf{o} \mapsto 0) &= \mathtt{pending}(H) \end{aligned}$$

Note that the order of the bindings in the heap does not matter, according to Proposition A.21.

Definition A.23 (Immediate actions). For a heap H, the immediate action $\mathtt{immediate}(H)$ is the heap action defined as follows:

$$\mathtt{immediate}(H)(\mathsf{o}) = \begin{cases} (\emptyset, \mathrm{dom}(H(\mathsf{o})), \emptyset, \emptyset) & \text{if } H(\mathsf{o}) \text{ is an environment,} \\ (\emptyset, \emptyset, \emptyset, \emptyset) & \text{otherwise} \end{cases}$$

Definition A.24. The action of a heap H, expression e and reference o is defined as follows:

$$\mathtt{action}(H, \mathsf{e}, \mathsf{o}) = \mathtt{immediate}(H) ;; \mathtt{pending}(H) ;; \mathsf{F}[\![\mathsf{e}]\!]_{\mathsf{o}}$$

Lemma A.25 (Preservation of well-flowedness).
If $\texttt{action}(H, \texttt{e}, \texttt{o}) = \mathbb{A}$ *and* $H[\![\texttt{e}]\!]_\texttt{o} \rightsquigarrow H'[\![\texttt{e}']\!]_\texttt{o}$, *then* $\texttt{action}(H', \texttt{e}', \texttt{o}) \leq \mathbb{A}$.

Proof. We proceed by structural induction on e.
We match on the syntax of e (we only include the most interesting cases):

- $\texttt{o}'$ Impossible case as no reduction step can apply on $\texttt{o}'$.
- $\texttt{prom}^{\texttt{fx}}\texttt{<t>}\{\texttt{e}\}$ The rule [DPro] applies. By adding a promise capturing the current environment in the heap H, its delayed action gets composed to the pending action $\texttt{pending}(H)$, but in the same time it gets removed from $\texttt{F}[\![\texttt{e}]\!]_\texttt{o}$. We can conclude by using the commutativity of delayed actions (Proposition A.21).
- $\{\texttt{o}_1\}^{\texttt{o}_3}_{\texttt{o}_2}$ The reduction rule [DFor3] applies, we thus have $\texttt{e}' = \texttt{o}_1$ and $H' = H, \texttt{o}_3 \mapsto \langle \texttt{o}_1, \texttt{k}, \texttt{o}_2 \rangle$. We can trivially conclude as adding a cached promise to the heap has no effect on the pending actions.
- $\{\texttt{e}_1\}^{\texttt{o}_3}_{\texttt{o}_2}$ The reduction rule [DCtx2] applies, we conclude by applying the induction hypothesis on $\texttt{e}_1$.
- $\texttt{force}\ \texttt{o}'$ The rule [DFor1] or [DFor2] applies. In the first case, we can conclude immediately. Otherwise, we have $\texttt{e}' = \{\texttt{e}''\}^{\texttt{o}'}_{\texttt{o}''}$ and $H' = H, \texttt{o}' \mapsto \langle \texttt{undef}, \texttt{k}, \texttt{o}'' \rangle$, meaning that the delayed action of e″ gets removed from $\texttt{pending}(H)$ (as the promise is updated with the cache undef), and the action of e″ is added to $\texttt{F}[\![\texttt{e}]\!]_\texttt{o}$ instead. We can conclude using Proposition A.20.
- $\texttt{f.i}(\texttt{o}_0 \ldots \texttt{o}_\texttt{n})$ The rule [DCal] applies. We can conclude using the fact that versions in F are well-flowed, and using Proposition A.18.
- $\texttt{o}_1\ ;\ \texttt{e}_2$ The rule [DSeq] applies. We thus have $e' = \texttt{e}_2$. We can trivially conclude this case.
- $\texttt{e}_1\ ;\ \texttt{e}_2$ The rule [DCtx3] or [DCtx4] applies. In the first case, we can immediately conclude. Otherwise, we have $e' = \texttt{e}'_1\ ;\ \texttt{e}_2$. We apply the induction hypothesis on $\texttt{e}_1$, yielding $\texttt{action}(H', \texttt{e}'_1, \texttt{o}) \leq \texttt{action}(H, \texttt{e}_1, \texttt{o})$. We conclude by using the monotonicity of the composition operator (Proposition A.16).

□

Definition A.26 (Well-flowedness). For a heap H, expression e and reference o, we say that $(H, \texttt{e}, \texttt{o})$ is well-flowed, written $\texttt{wfl}(H, \texttt{e}, \texttt{o})$, if and only if $\mathbb{A} = \texttt{action}(H, \texttt{e}, \texttt{o})$ is defined and $\forall(\texttt{o}' \mapsto (R, W, U, C)) \in \mathbb{A}.\ R = \varnothing$.

Lemma A.27 (Correctness of well-flowedness). *If* $\texttt{wfl}(H, \texttt{e}, \texttt{o})$, *then the reduction rules* [*DRegUndef*] *and* [*DConUndef*] *do not apply.*

Proof. Straightforward structural induction on e.
In order for the reduction rule [DRegUndef] to apply, e must be a register r. As $\texttt{F}[\![\texttt{r}]\!]_\texttt{o} = \texttt{o} \mapsto (\{\texttt{r}\}, \varnothing, \varnothing, \varnothing)$, this means that, in order for $\texttt{wfl}(H, \texttt{e}, \texttt{o})$ to hold, we must have $\texttt{immediate}(H)(\texttt{o}) \leq (\varnothing, \{\texttt{r}\}, \varnothing, \varnothing)$, and thus $\texttt{r} \in \texttt{dom}(H(\texttt{o}))$. The same applies for the reduction rule [DConUndef]. □

Theorem A.28 (Correctness of the flow analysis).
If $\varnothing[\![\texttt{f.i}()]\!]_\texttt{o} \rightsquigarrow^* H'[\![\texttt{e}']\!]_\texttt{o}$, *then* $\varnothing[\![\texttt{f.i}()]\!]_\texttt{o} \Rightarrow^* H'[\![\texttt{e}']\!]_\texttt{o}$.
If $\varnothing[\![\texttt{f.i}()]\!]_\texttt{o} \rightsquigarrow^\infty$, *then* $\varnothing[\![\texttt{f.i}()]\!]_\texttt{o} \Rightarrow^\infty$.

Proof. Immediate consequence of Proposition A.18, Lemma A.25 and Lemma A.27. □

We recall that this last theorem, as well as all the lemmas in the section, assume that every version ver in our function table F is well-defined.

Theorem A.29 (Type Safety).
If $\varnothing \vdash [\![\texttt{f.i()}]\!] : \texttt{t}$, *then either* $\varnothing[\![\texttt{f.i()}]\!]_{\texttt{o}} \Rightarrow^{\infty}$ *or* $\varnothing[\![\texttt{f.i()}]\!]_{\texttt{o}} \Rightarrow^{*} H'[\![\texttt{o}']\!]_{\texttt{o}}$ *for some* H' *and* $\texttt{o}'$ *such that* $H' \vdash \texttt{o}' \leq_{dyn} \texttt{t}$.

Proof. Immediate consequence of Theorem A.11 and Theorem A.28. □