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, Northeastern University, United States

Dynamic programming languages are characterized by features such as dynamic typing, late binding, runtime code loading, and reflection. To generate efficient code for such languages, implementations must speculate on which of these dynamic features will actually be exercised during the execution of a given function. Based on these assumptions, optimized code can be produced. A common approach is to compile source programs into explicitly typed intermediate representations that make relevant behaviors manifest and amenable to static analysis. This paper presents key design decisions for a statically typed, high-level intermediate representation that directly supports optimizations such as specialization, devirtualization, inlining, and copy elimination. We propose a core calculus, called FIŘ, that captures essential features required for these optimizations. We give a formal semantics, a type system and flow analysis rules, 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

ACM Reference Format:
Mickaël Laurent, Jakob Hain, Filip Krikava, Sebastián Krynski, and Jan Vitek. 2025. A Typed Intermediate Representation for Dynamic Languages. 1, 1 (July 2025), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, operator overloading, and reflection often prevent the compiler from reliably inferring program behavior through source code analysis alone.

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, Northeastern University, Boston, Massachusetts, United States.

ACM XXXX-XXXX/2025/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
function(x,j){
  r = 0
  if (scalar(x)){ r = x+r }
  else if (length(x) > j){ r = x[j] }
  return(r)
}
```

Fig. 1. A simple function

Consider the function of Fig. 1, written in a hypothetical dynamic language. The ease of generating efficient code for it varies dramatically depending on the language's degree of dynamism. At first glance, the function appears to return its first argument `x` if it is a scalar numeric value, the `j`-th element of `x` if it is a vector, and `0` otherwise.

However, our expectations can subverted in multiple ways. Focusing on the case where `scalar(x)` holds, we might expect the function to behave as the identity on numeric values. Yet even this seemingly simple case is fraught. If `x` is not a number, implicit string coercions could be triggered. If the `+` operator is overloaded, the expression `x+r` could invoke arbitrary effectful code and return unexpected values. Worse still, if `scalar` performs reflection or invokes native APIs, it might alter the program state—e.g., by mutating or even removing variable `r` from scope. Of course, the `if else` construct itself may be redefined to evaluate its arguments in non-standard ways.

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 invoke reflective mechanisms.

Building on this observation, just-in-time (JIT) compilers adopt a speculative strategy: they treat past program behavior as a predictor of future behavior and generate code optimized for the patterns observed at runtime. The compilation process is typically organized around key transformations that reduce—or at least expose—the sources of dynamism, enabling the application of standard optimizations such as constant propagation, code motion, and dead code elimination.

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? Primarily, it should provide features that allow the absence of dynamism to be explicitly marked—features that grant the compiler permission to perform optimizations. For example, the IR might encode knowledge of an object's shape, the target of a function call, or guarantee that reflective features will not be invoked in certain regions of code.

The precise set of such features will vary depending on the target language, but the design process follows a common pattern: identify the aspects of the language that hinder compilation and find ways to restrict or make them explicit. In essence, the goal is to convert properties that are typically resolved at runtime into ones that can be known and reasoned about statically.

The central thesis underlying this paper is that incorporating as many of these features as possible into a static type system enables static verification of compiler optimizations and provide a clear, 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. In this paper, we introduce a new core intermediate representation, FIŘ, designed to support features including dynamic typing, runtime code loading, delayed evaluation, copy-on-write semantics, and unrestricted reflection.

Although FIŘ is motivated by the specifics of the R language, we argue that its principles and design choices are broadly applicable to other dynamic languages. FIŘ is presented as a minimalistic calculus, inspired by core languages such as Featherweight Java [14]. Its reduced complexity enables us to provide a formal operational semantics (Section 5.3) and to prove the soundness of its type system (also in Section 5.4).

The key features of our IR include (see Section 4):

- **Variables:** FIŘ distinguishes *named* from *register* variables; the former can be updated non-locally though reflection, while the latter are restricted to the current lexical scope and shielded from reflection.
- **Types:** Inspired by gradually typed languages, FIŘ supports a stratified type system: some variables are statically typed, others are gradually typed, and some remain entirely dynamic.
- **Ownership:** A limited form of ownership is supported, enabling basic tracking of value lifetimes and transfer semantics, particularly for optimizing copy-on-write behavior.
- **Specialization:** Functions in FIŘ can be compiled into multiple specialized versions, each associated with a context that determines its applicability.
- **Binding:** FIŘ has dynamic binding, with calls matched to an applicable version of a function at runtime, and static binding.
- **Inlining:** Function inlining is supported to expose opportunities for further optimizations while preserving the scopes of named variables.
- **Promises:** FIŘ includes a notion of promises to represent delayed computations, enabling support for optimizations that target lazy evaluation.

Validity of FIŘ programs is guaranteed by a combination of type checking to ensure that variables and functions are used according to their declaration and flow analysis to make sure that variables are initialized before they are accessed (Section 5.2). Section 3 describes related work including prior that supported speculation and deoptimization which are omitted from our IR.

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 popular dynamic 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. Dynamic languages support the ability to load and execute new code at runtime. This can be achieved either by programmatically including external libraries or by reflectively transforming code represented as text into executable form.

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 dynamic languages allow built-in functions—such as + —to be redefined or overloaded at runtime.

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. 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 dynamic 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 value, results from evaluating the argument `p` in function `f`. Since `p` is a promise, its evaluation triggers a call to `g`, which 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 of dynamic languages hinges on the compiler's ability to rule out—or at least localize—the effects of dynamic features. These sources of dynamism may arise from the calling context, function arguments, or the transitive closure of reachable code.

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

This section reviews prior work related to intermediate representations in compilers, particularly in the context of dynamic languages.

3.1 Compiler IRs

Aycock provides an accessible overview of early implementation strategies for dynamic languages [1], covering systems ranging from Smalltalk [6] and Self [4] to more modern environments such as Java and JavaScript [20, 24].

While most compilers employ multiple intermediate representations to modularize the compilation process, these IRs are often treated as internal implementation details and are seldom documented comprehensively. One notable exception is the Java bytecode format, which is both clearly specified and formally studied [11, 16]. Since Java is statically typed and offers relatively limited dynamic behavior, the bytecode closely reflects the structure of the source language. A key addition, however, is the bytecode verifier—a simple form of data-flow analysis that ensures, among other things, that variables are initialized before use. The sea-of-nodes representation is a lower-level IR that makes all dependencies explicit in the control and data flow graphs [5]. Its structure is well-suited for optimizations such as duplication removal and promise elimination. Typed Assembly Language further demonstrated that static types can be applied to low-level IRs to preserve invariants during optimization—such as verifying that operands to an integer addition are indeed integers [18]. TAL assumes a statically typed source language, whereas our work addresses dynamically typed languages. Finally, MLIR offers a flexible, extensible IR framework capable of expressing a wide range of language features and analyses [15]. Although MLIR could support our target language, we chose to design an IR that more directly capture the language features of interest.

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 [12], that resemble FIŘ. In Swift, borrowing a value does not increment its reference count. Consuming parameters are mutable; a value passed to a consuming function cannot be reused by the caller afterward. PHP provides operations for assignment and aliasing; prior efforts in compilation have focused on distinguishing between them [13], but have not focused 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. However, our approach could be adapted to a more precise treatment of aliasing.

Ownership types refers to systems that constrain the use of aliases to heap allocated objects [19]. Wrigstad introduced the notion of **borrowing** as a way to relax strict ownership invariants [22]. Rust enforces ownership semantics via its borrow checker [17]. 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 due to the language's lazy evaluation model and the presence of first-class, mutable environments. To handle this, we introduce an explicit `use` instruction in FIŘ, along with a flow-sensitive analysis to detect potential read-after-use violations.

3.3 Gradual types and Like types

Gradual typing is a type system framework that allows both statically typed and dynamically typed code to coexist and interoperate within the same language. The gradual guarantee is a formal property that states that adding more precise types to a program should not change its behavior, except possibly to raise type errors earlier [21]. FIŘ follows the design of the Thorn programming language which, for performance reasons, eschews the gradual guarantee [23]. Thorn had concrete types (traditionally statically checked types), dynamic types (unchecked) and **like types**. The latter were checked for consistency but allowed to be assigned dynamic values. For a variable of type `like string`, the compiler would check that it was only operated on as if it was a string, but would allow unchecked coercion from dynamic types.

3.4 Deoptimization with Sourir and CoreIR

Prior work on formalizing just-in-time (JIT) compilation has introduced intermediate representations (IRs) that support both speculative compilation and deoptimization: Sourir [10] and CoreIR [2]. Both IRs adopt a similar design principle: they allow multiple versions of a function to coexist—typically at least one unoptimized version and one or more optimized versions that include speculative assumptions. **Sourir** extends a simple imperative intermediate language with a special `assume` instruction. This instruction encodes a speculative assumption: if the assumption holds at runtime, execution continues; if it fails, execution transfers to the corresponding unoptimized version of the function. This mechanism provides a foundation for deoptimization while preserving soundness of speculative optimizations. **CoreIR** takes a similar approach but introduces two distinct instructions: `Anchor deoptInfo label` and `Assume guardExp label`. The `Anchor` instruction is non-deterministic: it allows execution to either proceed normally or transition (via deoptimization) to a designated unoptimized code path. Anchors are primarily used during speculative optimization passes (e.g., speculation insertion) and are eliminated after optimization is complete. The `Assume` instruction is deterministic: it performs a deoptimization only if its guard expression evaluates to false. Unlike FIŘ, CoreIR does not index function versions with type information.

3.5 The Ř compiler and PIR

The Ř compiler is an optimizing compiler for the R language. Our work is done in the context of an extension of Ř , ans specifically of its PIR intermediate representation. Fig. 2 illustrates the compiler's various code representations. The original GNU-R code base is used to parse R source code into an AST. Then Ř has its own interpreter and bytecode, known as RIR, as well as an IR designed explicitly for high-level code transformations. Code generation is delegated to LLVM.

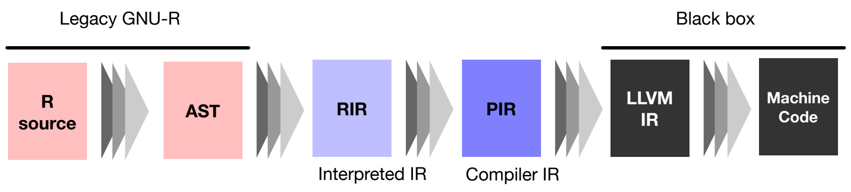


Fig. 2. Ř's existing IRs

PIR is an SSA-based IR [9]. An instruction has a type, effects, and immediates. The type is a combination of a kind and flags. A kind can be either an R type (*e.g.*, `real`, `int`, `env`), a union (*e.g.*, `real | int`), a missing value (`?`), or any value (`val`). Flags track relevant properties such as dimension (`$` is a scalar), objectness (`+` is not an object), delayed evaluation (`^` may be a promise).

```
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

Fig. 3 shows the unoptimized identity function. The `LdArg` loads an argument from stack returning `val?^`, *i.e.* any value that might be missing or a promise. The next instruction creates an environment, and stores the function parameter `a` in it. The `LdVar` loads `a` from the environment into a new register. The `eR` effects signal that it can possibly fail. `Force` evaluates the loaded value if it was a promise. The return type `%3.3` is `val?`, *i.e.* it can no longer be a promise. Lastly, the symbol (**!**) indicates that forcing the promise can have reflective effects. 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. Our work is an attempt to present formally grounded redesign of that IR.

4 An IR for Dynamic Languages

This section presents the features of FIř relevant to dynamic languages. Examples are given in slightly beautified syntax[1], one that we use in our practical implementation, however all features discussed in this section are formalized in the next section.

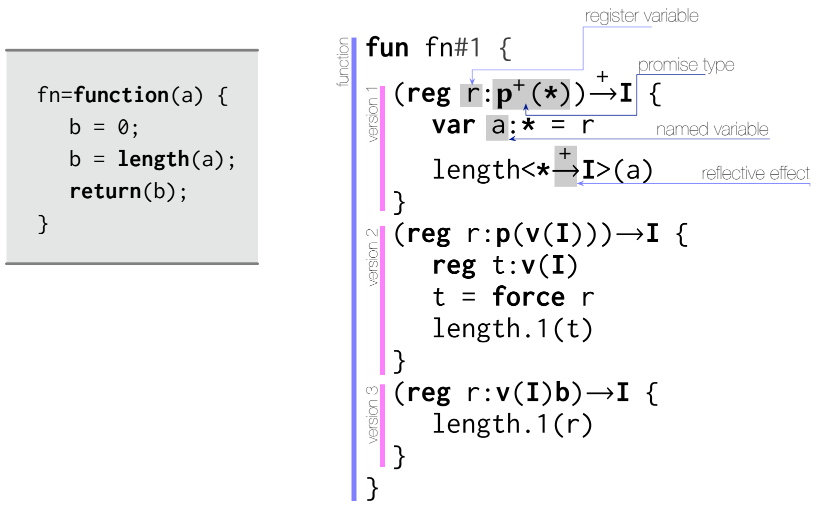
It is important to emphasize that an IR does not inherently create opportunities for new optimization; any transformation expressible in FIř can be performed on a lower-level IR. Instead, an IR should be evaluated on two key criteria:

(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. These help ensure that optimizations remain sound and that the IR remains a stable and reliable target for compiler transformations. While the IR is tailored to a specific source language and compiler infrastructure, we believe that the underlying principles are broadly applicable to dynamic languages—particularly those with fewer dynamic features than R.

We begin with an example. Fig. 4 presents a source-language function definition (left) alongside the corresponding code emitted by the compiler (right). A distinctive characteristic of our IR is that it allows each function to be compiled into a set of variants, each indexed by a signature. Each variant 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 variant—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. Next, we review the features of FIř.

[1]Specifically, we add reg and var annotations to distinguish between register and named variables, omit semi-colons, allow declarations and assignments to be combined, and use defaults in type signatures (types with no ownership modifier are **s**, and no concreteness are **!**).



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

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

Fig. 4. Source code (left) and FIŘ code with three versions (right)

4.1 Functions and Abstractions

A FIŘ *function* is composed of one or more *abstractions*. We refer to these abstractions as variants or versions of the function. An abstraction has a *signature* that lists argument types and return type as well as the potential effects of the abstraction, and a *body*.

```
(reg r:I)→I { r+1 }
```

Variants are created by the compiler to provide optimized version of a function. They differ in their signature and the transformations applied to the body. Variants 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.

Internally, the compiler maintains a flat namespace, where each unique function in the program has a single entry. Thus, `fn#1` in Fig. 4 corresponds to the compiled version of the function expression shown on the left. The mapping between names is an implementation detail. For our purposes, we focus on the internal table of functions maintained by the compiler.

4.2 Scopes and Variables

Each abstraction defines a *scope* in which variables can be defined. Our source language allows scopes to be shared with promises and to be inspected reflectively. Thus, variables that appear in the source code should be preserved in case another function decide to look them up reflectively.

One effective optimization is to eliminate a scope and all of its variables when it can be shown that reflection cannot inspect it. To enable this optimization, FIŘ supports two kinds of variables, *named variables* which are looked up symbolically and are exposed to reflection, and *register variables* which can be implemented in anyway the compiler sees fit and are not available to reflective operations.

Our source language allows shadowing of named variables and has nested scopes. FIŘ does not model nesting, so any lookup outside the current scope returns an unknown value.

The arguments of an abstraction are passed by register. A common pattern is to store them into named variables before calling a functions that may perform reflection. For instance, the source declaration `function(a) someFun(a)` can be compiled as follows:

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

where the register argument is explicitly copied to a named variable before calling a function that may perform reflection.

4.3 Promises: Creation and force

A *promise* represents a delayed computation. For example, the expression `f(x+y)` passes to `f` a promise that, when evaluated, computes `x+y`. Notably, promises execute in the lexical scope in which they were created. As a result, they can read and update register and named variables in that scope, and may even introduce new bindings—for instance, in `f(z=x+y)` if `z` was not previously defined. Promises, in the source language, are *forced* implicitly when their value is needed, and their result is cached upon evaluation.

In FIŘ, promises are represented explicitly. They are treated as first-class values with well-defined types and must be explicitly forced before use:

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

In this example, `r` is a promise that, when forced, returns an integer. The promise body here is a simple expression that evaluates to `42`.

A subtle but important language specific detail is that, in our source language, promises are automatically forced when assigned to a variable or returned from a function: thus promises do not outlive the scope in which they were created. Our type system leverages this invariant to simplify reasoning about promise lifetimes and side effects.

The `force` instruction creates some indeterminacy regarding when promises are evaluated and when their side effects are observed. FIŘ incorporates flow analysis that prevents read before write errors.

4.4 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. For example, in a language like R that maintains an approximate reference count, `dup` needs to copy only when the reference count exceeds one. By making duplication explicit, the IR enables optimizations that eliminate unnecessary copies —something that is difficult or impossible when copies are implicitly performed 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, mutable function arguments typically undergo a single duplication before any mutation, enabling efficient updates while preserving copy-on-write semantics.

4.5 Types: ownership, graduality and effects

Types in FIŘ serve multiple purposes: they are used to verify that compiler transformations preserve well-formedness, to enable and guide optimizations, and to record compiler-generated hints. To support this range of uses, FIŘ adopts a gradual type system, building on the work of Wrigstad *et al.* [23].

At a high level, the type system distinguishes between three kinds of types: *Concrete types*, which describe values whose types are known precisely; a *dynamic type*, used when the type of a value is entirely unknown; and *Like types*, which represent values the compiler expects to conform to a certain type, but for which it lacks static guarantees. In addition, the type system tracks ownership qualifiers for values, enabling optimizations such as copy elimination and enforcing safe behavior in the presence of reflection and delayed evaluation. This information is particularly important when reasoning about functions and promises that may observe or mutate shared state. Formally, each type in FIŘ is composed of three components: a *kind* `k`, defining the structure of the values (e.g. `I` for integers, `v(I)` for vectors of integers); an *ownership modifier*, and a *concreteness modifier* `cx`, indicating whether the type is certain (`!`) or speculated (`?`).

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).

Information about the kind of an expression can be used by the compiler to devirtualize a call, or to statically select the best version of a function. For instance, a call to `fn#1` (Fig. 4) could be statically bound to the version 3 if the argument is known to be a vector of integers (e.g. `v(I)o!`). Although they do not offer static guarantees, like types are be used to provide hints to the compiler. For example, one could type an abstraction with a `I?` to indicate that we have some reason to expect 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, if it is to use that information, will cast (and should deoptimize in case of failure or have a branch).

Ownership modifiers can be either `f` (fresh), `o` (owned), `s` (shared), and `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 is 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 change 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 "using" the register (`use r`); flow analysis ensures a register cannot be accessed after it is used.

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

This code can then be simplified into the following:

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

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.

Promise types and function signatures also hold information about their *effects*. In FIŘ, effects hold a single bit of information: + indicates that the promise or function may make use of reflection, while - ensures they do not. As such, the kind of a promise yielding an integer and that may perform reflection when forced is noted $\mathbf{p}^{+}$`(Is!)`. Values returned by a promise are shared as they can be accessed reflectively. Although effects only store one bit of information (can/cannot reflect), one could easily extend them to distinguish reflective operations (e.g. reflective read versus writes). Effects do not directly impact the typeability of an expression, as named variables are already treated pessimistically: they are either dynamic or like types. Still, annotating promises and function signatures with their potential effects is essential for optimizations.

4.6 Binding Time and Inlining

When considering a call such as `f(x)`, it is 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. However, FIŘ does not have built-in support for dynamic binding mechanisms. Instead, in the case of object-oriented dispatch—which can be highly complex—it is delegated to user-level functions from the source language's reference implementation. FIŘ focuses on the binding decisions made by the compiler when selecting a version of a known function.

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. For dispatched calls, we do not mandate how a compiler selects the "best" version of a function; 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<`$\mathbf{p}^{+}$`(*)`$\xrightarrow{+}$`I>( r0 ) # Dispatched call`

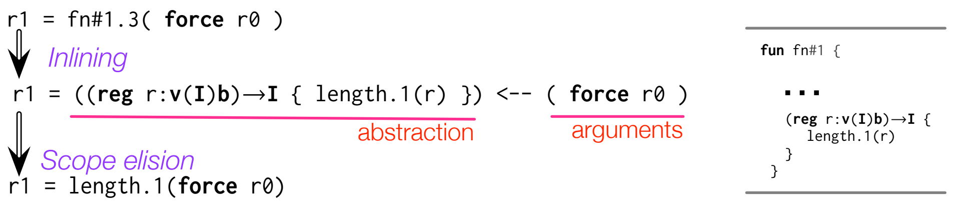
Here, the signature `<`$\mathbf{p}^{+}$`(*)`$\xrightarrow{+}$`I>` constrains the version of `fn#1` that the compiler will call; this signature is compatible with version 1 in Fig. 4. A more sophisticated compiler—such as Ř, which supports contextual dispatch [8]—might dynamically inspect the promise held in `r0`. If it determines that the promise is effect-free and returns an integer, it could instead select version 2.

Dispatched calls are annotated with static type signatures for two reasons. Firstly, some of the properties required to select a compatible version at runtime may not be dynamically recoverable. For example, ownership qualifiers are computed statically and cannot be inferred from the program state. Similarly, the kind of an argument cannot be determined at runtime if the argument is `undef`. Secondly, to ensure soundness, the version selected at runtime must return a value that is type-compatible with the version selected statically. In FIŘ, it is permissible for two versions of the same function to produce different results for the same inputs, so long as the selected version conforms to the type system.

Static calls do not require a signature as they specify the version number explicitly; the call is checked statically, and the abstraction is called directly without searching for an applicable version, as in:

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

Finally, FIŘ supports *inlining*. When the compiler knows which abstraction will be executed—such as in a static call—it may choose to replace the call with the body of the corresponding abstraction. Fig. 5 illustrates the syntax used to represent inlined code. A key point is that inlining preserves the scope of the inlined abstraction. That is, named variables defined within the callee remain scoped to a distinct environment, separate from the caller. This design is required by the semantics of the source language, which allows reflection to observe the presence and boundaries of variable scopes. In other words, scope boundaries are semantically visible and must be respected.



```
r1 = fn#1.3( force r0 )
r1 = ((reg r:v(I)b)→I { length.1(r) }) <-- ( force r0 )
r1 = length.1(force r0)
```

```
fun fn#1 {
  ...
  (reg r:v(I)b)→I {
    length.1(r)
  }
}
```

Fig. 5. Inlining and scope elision

The removal of such scopes—known as scope elision—is typically performed as a separate optimization pass. If the compiler can determine that no reflective operation will observe a particular scope, it may safely remove it. Importantly, this transformation does not require explicit representation or support in the IR itself.

4.7 Discussion

The design of Fiř draws on lessons from prior work on the Ř compiler infrastructure. Here are some noteworthy design choices and simplifications:

Deoptimization. Our implementation uses the deoptimization construct of CoreIR unchanged [2]. For simplicity we have chosen to omit them from the definition of Fiř. Including them would be relatively straightforward, but would obscure the semantics of the IR with support for on stack replacement. The only added requirement that deoptimization places on the IR is the need for an identified base (or unoptimized) version for each function, it is that version that will be called when speculation fails.

Reflection. The specific reflection interface is not specified. However, implicit to Fiř's design is the possibility for named variables to be redefined or even deleted. If the source language did not expose local variables to reflection, it would be possible to give them stronger types. Consider the following abstraction, it take a value, stores it in named variable `a`, calls a possibly-reflective function `f` and then returns `a`:

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

In R, `f` can delete the named variable which results in the lookup of `a` to return either the value of `a` in an enclosing scope (or at the top level), or fail.

Promises. In Fiř, promises cannot escape their scope of creation. This simplifies the ownership analysis as without this restriction, any register accessed in a promise that outlives its scope of creation should be considered shared. This restriction is acceptable since in R, promises are forced when they are assigned and returned. Still, it is possible in R for a promise to escape its scope of creation by using a *delayed assignment* or by returning the environment in which it lives. However, these cases are rare and can be treated as reflection.

Omitted Features. The R language has a number of features that are not presented here. Most of them are implemented by calls to native functions and do not deserve to be reflected in the IR. Two of those are worth mentioning: name lookups and attributes.

The semantics of name lookup in R is surprisingly rich, a chain of environment is searched and in some cases it is necessary to force promises to know 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 of that scope, the implementation defaults the source language lookup mechanism.

In R, any value can have a set of attached attributes; the equivalent of a map from name to value. So, for example, the `class` attribute can be attached to any value to give that value a particular behavior during object-oriented dispatch. The Ř compiler only supports attributes in as much as being able to note their absence which is the overwhelmingly common case. This could be easily added to Fiř as an extra bit on the type of values. A more fine grained treatment is left to future work.

5 Syntax and Semantics of FIŘ

FIŘ is a core calculus that gives semantics to an intermediate language for compiling dynamic languages such as R. As such, it includes the features described in the previous section and deliberately abstracts away many of the low-level details required in a full-featured intermediate representation.

Reflecting this design philosophy, FIŘ includes only a minimal set of data types and control-flow 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.

5.1 Syntax

$$
\begin{array}{rcl}
\mathtt{fn} & ::= & \textbf{fun}\ \mathtt{f\{abs_1 \ldots abs_n\}} \\
\mathtt{abs} & ::= & \mathtt{(rdefs)} \xrightarrow{\mathtt{fx}} \mathtt{t\{defs;\ e\}} \\
\mathtt{rdefs} & ::= & \mathtt{r_0 : t_0 \ldots r_n : t_n} \\
\mathtt{defs} & ::= & \mathtt{v_0 : t_0 \ldots v_n : t_n} \\
\mathtt{t} & ::= & \mathtt{k\ ox\ cx} \\
\mathtt{k} & ::= & \boldsymbol{*} \mid \mathbf{V} \mid \mathbf{I} \mid \mathbf{v}(\mathbf{I}) \mid \mathbf{p}^{\mathtt{fx}}(\mathtt{t}) \\
\mathtt{ox} & ::= & \mathbf{o} \mid \mathbf{b} \mid \mathbf{s} \mid \mathbf{f} \\
\mathtt{fx} & ::= & + \mid - \\
\mathtt{cx} & ::= & ! \mid ? \\
\mathtt{sig} & ::= & \mathtt{t_0 \ldots t_n} \xrightarrow{\mathtt{fx}} \mathtt{t} \\
\mathtt{v} & ::= & \mathtt{x} \mid \mathtt{r}
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{e} & ::= & \mathtt{i} \mid \mathbf{vec}(\mathtt{e_1 \ldots e_n}) \mid \mathtt{v} \mid \mathtt{e[e]} \mid \mathtt{e\$x} \\
 & \mid & \mathtt{v = e} \mid \mathtt{e[e] = e} \mid \mathtt{e\$x = e} \\
 & \mid & \mathtt{abs} \longleftarrow \mathtt{(e_0 \ldots e_n)} \\
 & \mid & \mathtt{f.i(e_0 \ldots e_n)} \\
 & \mid & \mathtt{f<sig>(e_0 \ldots e_n)} \\
 & \mid & \mathtt{e}\ \textbf{as}\ \mathtt{t} \\
 & \mid & \textbf{force}\ \mathtt{e} \\
 & \mid & \textbf{use}\ \mathtt{r} \\
 & \mid & \textbf{dup}\ \mathtt{e} \\
 & \mid & \textbf{prom}^{\mathtt{fx}}\mathtt{<t>\{e\}} \\
 & \mid & \mathtt{e\ ;\ e}
\end{array}
$$

Fig. 6. Syntax

The full syntax of the FIŘ calculus is given in Fig. 6. A program consists of a *function table* F and an *expression* e. A function table consists of one or more *function* declarations, **fun** f{abs$_1$ ... abs$_n$}, where f is the function's unique name, and abs$_i$ is an *abstraction*. An abstraction, $\mathtt{(rdefs)} \xrightarrow{\mathtt{fx}} \mathtt{t}$ { defs; e }, has typed argument definitions rdefs, local variables defs, a body expression e, a return type t and an effect modifier fx. FIŘ distinguishes between *register variables*, ranged over by r, and *named variables*, ranged over by x. Parameters are restricted to register variables, while the body can define both named and register variables.

FIŘ's expressions are: integer literals (i), vector literals (**vec**(e$_1$ ... e$_n$)), variables (v) which can be either register variables (r) or named variables (x), indexed vector reads (e[e]), reflective named variable reads (e$x), variable writes (v = e), indexed vector writes (e[e] = e), reflective named variable writes (e$x = e), inlined abstraction applications (abs ⟵ (e$_0$ ... e$_n$)), static function calls (f.i(e$_0$ ... e$_n$)), dispatched function calls (f <sig> (e$_0$ ... e$_n$)), type-casts (e **as** t), promise forces (**force** e), register variable uses (**use** r), duplications (**dup** e), promise literals (**prom**$^{\mathtt{fx}}$<t>{e}), and sequencing (e ; e).

Reflective reads and writes. To model the challenges posed by dynamic features, we include a limited form of reflection in the calculus. This restricted reflective capability is sufficient to capture the difficulties it presents to an optimizing compiler when one function can modify the state of another.

Creating a promise in FIŘ captures both an expression—its body—and the environment of the enclosing abstraction. Given a variable r referencing a promise defined in a different abstraction, the following code demonstrates how to mutate a named variable x in the captured environment:

```
r$x = 42
```

This operation can occur regardless of the expected type of x, or even its existence. Note that only named variables can be accessed and assigned reflectively, register variables can only be accessed directly from their definition scope.

This ability to access named variables reflectively may leak the structure of the environments. In particular, inlining only the body of an abstraction is unsound precisely because it would merge the abstraction's environment with the surrounding context, potentially altering behavior in the presence of reflection. Consider the following example:

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

These two statements have equivalent meanings since the abstraction's body does not employ reflection. This contrasts with the following two lines that have different behaviors due to the use of reflection (we assume that r0 is a promise capturing the current local environment):

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

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

Reflection introduces a compelling challenge in language design, as any variable subject to reflection inherently possesses uncertain types. Operations that induce reflective effects, indicated by a + in their signatures, can alter the types of all involved variables.

Types. In FIř, types, ranged over by t, are composed of: a *kind* (k), an *ownership modifier* (ox), and a *concreteness modifier* (cx). Effect modifiers, fx, extend the types of functions and promises: **+** indicates that reflective operations may be performed and **–** guarantees otherwise. 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. The ownership modifiers are: owned (**o**), borrowed (**b**), shared (**s**), and fresh (**f**). Finally, the possible concreteness modifiers are: concrete (**!**), or like (?).

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

$$\begin{array}{lcccl} \mathtt{k\,ox\,cx} & \& & \mathtt{k'} & = & \mathtt{k'\,ox\,cx} \\ \mathtt{k\,ox\,cx} & \& & \mathtt{ox'} & = & \mathtt{k\,ox'\,cx} \\ \mathtt{k\,ox\,cx} & \& & \mathtt{cx'} & = & \mathtt{k\,ox\,cx'} \end{array}$$

We use a number of predicates for convenience:

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

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

$$\frac{\mathtt{t} = \mathtt{t}\ \&\ *\ \&\ ?}{\mathtt{wf(t)}} \qquad \frac{\mathtt{t} = \mathtt{t}\ \&\ \mathbf{I}}{\mathtt{wf(t)}} \qquad \frac{\mathtt{t} = \mathtt{t}\ \&\ \mathbf{v}(\mathbf{I})}{\mathtt{wf(t)}} \qquad \frac{\mathtt{t} = \mathtt{t}\ \&\ \mathbf{V}}{\mathtt{wf(t)}} \qquad \frac{\mathtt{wf(t')} \quad \begin{array}{c} \mathtt{t} = \mathtt{t}\ \&\ \mathbf{p}^{\mathtt{fx}}(\mathtt{t'}) \\ \mathtt{shared(t')} \end{array} \quad \mathtt{value(t')}}{\mathtt{wf(t)}}$$

These rules ensure that the any kind ($*$) has a concreteness modifier of ?, and that the type returned by a promise is shared and cannot itself be a promise. The well-formedness constraints are extended to variable declarations with:

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

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

Casts. A type-cast e **as** t checks that *e* has type *t* at runtime. 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. Types casts can also be used on expressions with a concrete type in order to refine this type (e.g., refining a type **V**o! into **I**o!).

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 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.

5.2 Static Semantics

The static semantics of FIŘ are defined by two relations over abstractions, $[\![\mathsf{abs}]\!] : \mathtt{sig}$ and $\mathsf{F}[\![\mathsf{abs}]\!]$. These relations assert, respectively, that an abstraction abs is well-typed and well-flowed. The function table *F* of a program is considered well-defined when every abstraction within it satisfies both criteria. The typing relation guarantees adherence to constraints related to types, ownership, concreteness, and effects. The flow relation ensures that register variables are initialized before they are used and prevents their access after a use operation. For convenience, we consider the function table *F* to be fixed, and allow the deduction rules for the judgments $[\![\mathsf{abs}]\!] : \mathtt{sig}$ and $\mathsf{F}[\![\mathsf{abs}]\!]$ to access it, without making this dependency explicit in the judgment.

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

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

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

- For types:

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

- For abstraction signatures:

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

Type checking. The [Abs] rule shown next checks the definition of an abstraction. The variables that can be used in an abstraction 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 abstraction's body. All type annotations must be well-formed. The effect modifier and return type of the abstraction must be those inferred for its body, with the ownership component transformed as follows: if the body returns an owned value, then the value returned by the abstraction is considered fresh (because the only reference to that value does not exist anymore when exiting the scope of the abstraction); otherwise, if the body returns a fresh or shared value, the ownership is unchanged. Note that the abstraction body should not return a borrowed value (in such a case, the abstraction is untypeable). Also, abstractions can only return non-promise values, otherwise it would allow promises to escape their scope of definition, which is undesirable as discussed below.

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

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

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

A parameter of type $\mathtt{t}'$ can be given an argument of type $\mathtt{t}$ if and only if: (1) the kind and concreteness of $\mathtt{t}$ are subtypes of the kind and concreteness of $\mathtt{t}'$; and (2*i*) if $\mathtt{t}'$ expects a borrowed value, then $\mathtt{t}$ can be of any ownership, (2*ii*) if $\mathtt{t}'$ expects a shared value, then $\mathtt{t}$ can be either shared or fresh, or (2*iii*) if $\mathtt{t}'$ expects an owned value, then $\mathtt{t}$ must be fresh. Note that function parameters must be well-formed, and thus they cannot be given types with a fresh ownership.

The only literals are integers, and integers are always shared. All types are explicit; promise return and call signatures are checked, never inferred. Sequences have the type of their last element, the initial elements' types are checked but otherwise ignored. 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. Lastly, promises cannot be reflectively written, which prevents them from escaping their scope of definition: this is required for the type safety to hold, as explained in Section 5.4. Note that the type system as defined in Fig. 7 is not syntax-directed because of the subsumption rule Sub. However, this is only a presentation choice we made for clarity and concision, as subsumption can easily be inlined in other rules whenever needed.

Effect analysis. Fig. 8 shows the analysis that checks effect annotations: a function must have the reflection modifier (+) if any of its applied abstractions or forced promises have the reflection modifier. Note that the effect analysis does not impact the typeability directly (effects have 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.

$$\text{[Lit]}\ \frac{}{\Gamma \vdash \llbracket i \rrbracket : \mathbf{Is!}} \qquad \text{[Var]}\ \frac{}{\Gamma, v : t \vdash \llbracket v \rrbracket : t} \qquad \text{[Sub]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : t \\ t \leq t'\end{array}}{\Gamma \vdash \llbracket e \rrbracket : t'} \qquad \text{[Seq]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : t \\ \Gamma \vdash \llbracket e' \rrbracket : t'\end{array}}{\Gamma \vdash \llbracket e; e' \rrbracket : t'}$$

$$\text{[Pro]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : t' \\ \Gamma \vdash E\llbracket e \rrbracket = fx \\ t = \mathbf{p}^{fx}(t')\mathbf{s!} \qquad wf(t)\end{array}}{\Gamma \vdash \llbracket \mathbf{prom}^{fx}\texttt{<}t'\texttt{>}\{e\} \rrbracket : t} \qquad \text{[Vec]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e_1 \rrbracket : \mathbf{Is!} \ldots \Gamma \vdash \llbracket e_n \rrbracket : \mathbf{Is!} \\ t' = \mathbf{v(I)f!}\end{array}}{\Gamma \vdash \llbracket \mathbf{vec}(e_1 \ldots e_n) \rrbracket : t'} \qquad \text{[Rea]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : t \\ \Gamma \vdash \llbracket e' \rrbracket : \mathbf{Is!} \\ t \,\&\, \mathbf{f} \leq \mathbf{v(I)f!}\end{array}}{\Gamma \vdash \llbracket e[e'] \rrbracket : \mathbf{Is!}}$$

$$\text{[App]}\ \frac{\begin{array}{c}\llbracket abs \rrbracket : t_0 \ldots t_n \xrightarrow{fx} t \\ \Gamma \vdash \llbracket e_0 \rrbracket : t'_0 \ldots \Gamma \vdash \llbracket e_n \rrbracket : t'_n \\ match(t'_0, t_0) \ldots match(t'_n, t_n)\end{array}}{\Gamma \vdash \llbracket abs \leftarrow (e_0 \ldots e_n) \rrbracket : t} \qquad \text{[Cal]}\ \frac{\begin{array}{c}sig(F(f)(i)) = t_0 \ldots t_n \xrightarrow{fx} t \\ \Gamma \vdash \llbracket e_0 \rrbracket : t'_0 \ldots \Gamma \vdash \llbracket e_n \rrbracket : t'_n \\ match(t'_0, t_0) \ldots match(t'_n, t_n)\end{array}}{\Gamma \vdash \llbracket f.i(e_0 \ldots e_n) \rrbracket : t}$$

$$\text{[Dis]}\ \frac{\begin{array}{c}sig = t_0 \ldots t_n \xrightarrow{fx} t \\ \exists i.\ sig(F(f)(i)) \leq sig \\ \Gamma \vdash \llbracket f.i(e_0 \ldots e_n) \rrbracket : t'\end{array}}{\Gamma \vdash \llbracket f\,\texttt{<}sig\texttt{>}\,(e_0 \ldots e_n) \rrbracket : t} \qquad \text{[Cas]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : t' \\ wf(t) \\ t' \leq t \text{ or } t \leq t'\end{array}}{\Gamma \vdash \llbracket e\ \mathbf{as}\ t \rrbracket : t} \qquad \text{[For]}\ \frac{\Gamma \vdash \llbracket e \rrbracket : \mathbf{p}^{fx}(t)\mathbf{s!}}{\Gamma \vdash \llbracket \mathbf{force}\ e \rrbracket : t}$$

$$\text{[Use]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket r \rrbracket : t \\ owned(t) \\ t' = t \,\&\, \mathbf{f}\end{array}}{\Gamma \vdash \llbracket \mathbf{use}\ r \rrbracket : t'} \qquad \text{[Dup]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : t \\ vec(t) \\ t' = t \,\&\, \mathbf{f}\end{array}}{\Gamma \vdash \llbracket \mathbf{dup}\ e \rrbracket : t'} \qquad \text{[Ass]}\ \frac{\begin{array}{c}t = \Gamma(v) \qquad \Gamma \vdash \llbracket e \rrbracket : t' \\ match(t', t) \\ \neg borrowed(t)\end{array}}{\Gamma \vdash \llbracket v = e \rrbracket : t}$$

$$\text{[Wri]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : \mathbf{v(I)o!} \\ \Gamma \vdash \llbracket e' \rrbracket : t \\ \Gamma \vdash \llbracket e'' \rrbracket : t \\ t = \mathbf{Is!}\end{array}}{\Gamma \vdash \llbracket e[e'] = e'' \rrbracket : t} \qquad \text{[Ref]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : t \\ prom(t)\end{array}}{\Gamma \vdash \llbracket e\$x \rrbracket : \mathbf{*s?}} \qquad \text{[RWr]}\ \frac{\begin{array}{c}\Gamma \vdash \llbracket e \rrbracket : t \qquad prom(t) \\ \Gamma \vdash \llbracket e' \rrbracket : t' \qquad value(t') \qquad shared(t')\end{array}}{\Gamma \vdash \llbracket e\$x = e' \rrbracket : t'}$$

Fig. 7. Types

- An abstraction is reflective if annotated as such. An abstraction 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 effect analysis and the type system mutually recursive, though this definition is well-founded as recursive calls only apply on strict sub-expressions.
- An application is reflective if the abstraction being applied is, and a call is reflective if the abstraction being called is.
- Any expression containing reflective sub-expressions is reflective.

We define the effect union $fx \vee fx'$ to be $\mathbf{+}$ if either fx or fx' is, otherwise $\mathbf{-}$:

$$\begin{array}{rcl} \mathbf{-} \vee \mathbf{-} & = & \mathbf{-} \\ \mathbf{-} \vee \mathbf{+} & = & \mathbf{+} \\ \mathbf{+} \vee fx & = & \mathbf{+} \end{array}$$

Flow Analysis. A program is well-flowed if register variables are initialized before their first access and never accessed after a use. The challenge here is related to promises, as the point in the program where they are evaluated may be difficult (or even, in the general case, impossible) to know statically. A precise solution would involve a data-flow analysis that would model the flow of promise 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 use (in the sense of **use** r).
- 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.

$$\text{[EAbs]}\ \frac{\begin{array}{c}\Gamma = \mathtt{rdefs}, \mathtt{defs} \\ \Gamma \vdash \mathsf{E}[\![\mathtt{e}]\!] = \mathtt{fx}' \qquad \mathtt{fx}' \le_f \mathtt{fx}\end{array}}{\mathsf{E}[\![(\mathtt{rdefs}) \xrightarrow{\mathtt{fx}} \mathtt{t}\{\mathtt{defs};\ \mathtt{e}\}]\!] = \mathtt{fx}} \qquad \text{[ELit]}\ \frac{}{\Gamma \vdash \mathsf{E}[\![\mathtt{i}]\!] = -} \qquad \text{[EVar]}\ \frac{}{\Gamma \vdash \mathsf{E}[\![\mathtt{v}]\!] = -}$$

$$\text{[ESeq]}\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{E}[\![\mathtt{e}]\!] = \mathtt{fx} \\ \Gamma \vdash \mathsf{E}[\![\mathtt{e}']\!] = \mathtt{fx}' \\ \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'\end{array}}{\Gamma \vdash \mathsf{E}[\![\mathtt{e}; \mathtt{e}']\!] = \mathtt{fx}''} \qquad \text{[EVec]}\ \frac{\Gamma \vdash \mathsf{E}[\![\mathtt{e}_0; \dots \mathtt{e}_n]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{E}[\![\mathbf{vec}(\mathtt{e}_1 \dots \mathtt{e}_n)]\!] = \mathtt{fx}} \qquad \text{[EApp]}\ \frac{\begin{array}{c}\mathsf{E}[\![\mathtt{abs}]\!] = \mathtt{fx} \\ \Gamma \vdash \mathsf{E}[\![\mathtt{e}_0; \dots \mathtt{e}_n]\!] = \mathtt{fx}' \\ \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'\end{array}}{\Gamma \vdash \mathsf{E}[\![\mathtt{abs} \leftarrow (\mathtt{e}_0 \dots \mathtt{e}_n)]\!] = \mathtt{fx}''}$$

$$\text{[ECal]}\ \frac{\begin{array}{c}\mathtt{sig}(F(\mathtt{f})(\mathtt{i})) = \mathtt{t}_0 \dots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \\ \Gamma \vdash \mathsf{E}[\![\mathtt{e}_0; \dots \mathtt{e}_n]\!] = \mathtt{fx}' \\ \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'\end{array}}{\Gamma \vdash \mathsf{E}[\![\mathtt{f.i}(\mathtt{e}_0 \dots \mathtt{e}_n)]\!] = \mathtt{fx}''} \qquad \text{[EDis]}\ \frac{\begin{array}{c}\mathtt{sig} = \mathtt{t}_0 \dots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \\ \Gamma \vdash \mathsf{E}[\![\mathtt{e}_0; \dots \mathtt{e}_n]\!] = \mathtt{fx}' \\ \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'\end{array}}{\Gamma \vdash \mathsf{E}[\![\mathtt{f}\, \texttt{<sig>}\, (\mathtt{e}_0 \dots \mathtt{e}_n)]\!] = \mathtt{fx}''}$$

$$\text{[EFor]}\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{E}[\![\mathtt{e}]\!] = \mathtt{fx} \\ \Gamma \vdash [\![\mathtt{e}]\!] : \mathbf{p}^{\mathtt{fx}'}(\mathtt{t}) \\ \mathtt{fx}'' = \mathtt{fx} \vee \mathtt{fx}'\end{array}}{\Gamma \vdash \mathsf{E}[\![\mathbf{force}\ \mathtt{e}]\!] = \mathtt{fx}''} \qquad \text{[ECas]}\ \frac{\Gamma \vdash \mathsf{E}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{E}[\![\mathtt{e}\ \mathbf{as}\ \mathtt{t}]\!] = \mathtt{fx}} \qquad \text{[ERefl]}\ \Gamma \vdash \mathsf{E}[\![\mathtt{r\$x}]\!] = +$$

$$\text{[ERea]}\ \frac{\Gamma \vdash \mathsf{E}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{E}[\![\mathtt{r}[\mathtt{e}]]\!] = \mathtt{fx}} \qquad \text{[EDup]}\ \frac{\Gamma \vdash \mathsf{E}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{E}[\![\mathbf{dup}\ \mathtt{e}]\!] = \mathtt{fx}} \qquad \text{[EUse]}\ \frac{\Gamma \vdash \mathsf{E}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{E}[\![\mathbf{use}\ \mathtt{e}]\!] = \mathtt{fx}}$$

$$\text{[EPro]}\ \frac{}{\Gamma \vdash \mathsf{E}[\![\mathbf{prom}^{\mathtt{fx}}\texttt{<t>}\{\mathtt{e}\}]\!] = -} \qquad \text{[EAss]}\ \frac{\Gamma \vdash \mathsf{E}[\![\mathtt{e}]\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{E}[\![\mathtt{v} = \mathtt{e}]\!] = \mathtt{fx}} \qquad \text{[EWri]}\ \frac{\Gamma \vdash \mathsf{E}[\![\mathtt{e}; \mathtt{e}']\!] = \mathtt{fx}}{\Gamma \vdash \mathsf{E}[\![\mathtt{r}[\mathtt{e}] = \mathtt{e}']\!] = \mathtt{fx}}$$

$$\text{[ERwr]}\ \Gamma \vdash \mathsf{E}[\![\mathtt{r\$x} = \mathtt{e}]\!] = +$$

Fig. 8. Effect Analysis

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.

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

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

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

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

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.

$$\text{[FLit]}\ \frac{\mathsf{A} = (\emptyset, \emptyset, \emptyset, \emptyset)}{\mathsf{F}[\![\mathsf{i}]\!] = \mathsf{A}} \qquad \text{[FVar]}\ \frac{\mathsf{A} = (\emptyset, \emptyset, \emptyset, \emptyset)}{\mathsf{F}[\![\mathsf{x}]\!] = \mathsf{A}} \qquad \text{[FSeq]}\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{F}[\![\mathsf{e}']\!] = \mathsf{A}' \qquad \mathsf{A}'' = \mathsf{A} \mathbin{;;} \mathsf{A}'}{\mathsf{F}[\![\mathsf{e}; \mathsf{e}']\!] = \mathsf{A}''}$$

$$\text{[FMul]}\ \frac{\begin{array}{c}\mathsf{F}[\![\mathsf{e}_0]\!] = \mathsf{A}_0 \dots \mathsf{F}[\![\mathsf{e}_n]\!] = \mathsf{A}_n\\ \mathsf{A}' = (\emptyset, \emptyset, \emptyset, \emptyset) \mathbin{;;} \mathsf{A}_0 \mathbin{;;} \dots \mathsf{A}_n\end{array}}{\mathsf{F}[\![\mathsf{e}_0 \dots \mathsf{e}_n]\!] = \mathsf{A}'} \qquad \text{[FVec]}\ \frac{\mathsf{F}[\![\mathsf{e}_1 \dots \mathsf{e}_n]\!] = \mathsf{A}}{\mathsf{F}[\![\mathbf{vec}(\mathsf{e}_1 \dots \mathsf{e}_n)]\!] = \mathsf{A}} \qquad \text{[FApp]}\ \frac{\begin{array}{c}\mathsf{F}[\![\mathsf{abs}]\!]\\ \mathsf{F}[\![\mathsf{e}_0 \dots \mathsf{e}_n]\!] = \mathsf{A}\end{array}}{\mathsf{F}[\![\mathsf{abs} \longleftarrow (\mathsf{e}_0 \dots \mathsf{e}_n)]\!] = \mathsf{A}}$$

$$\text{[FCal]}\ \frac{\mathsf{F}[\![\mathsf{e}_0 \dots \mathsf{e}_n]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{f.i}(\mathsf{e}_0 \dots \mathsf{e}_n)]\!] = \mathsf{A}} \qquad \text{[FDis]}\ \frac{\mathsf{F}[\![\mathsf{e}_0 \dots \mathsf{e}_n]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{f} \mathbin{<}\mathsf{sig}\mathbin{>} (\mathsf{e}_0 \dots \mathsf{e}_n)]\!] = \mathsf{A}} \qquad \text{[FCas]}\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{e}\ \mathbf{as}\ \mathsf{t}]\!] = \mathsf{A}}$$

$$\text{[FFor]}\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}}{\mathsf{F}[\![\mathbf{force}\ \mathsf{e}]\!] = \mathsf{A}} \qquad \text{[FReg]}\ \frac{\mathsf{A} = \mathsf{read}\ \mathsf{r}}{\mathsf{F}[\![\mathsf{r}]\!] = \mathsf{A}} \qquad \text{[FRefl]}\ \frac{\mathsf{A} = \mathsf{read}\ \mathsf{r}}{\mathsf{F}[\![\mathsf{r\$x}]\!] = \mathsf{A}} \qquad \text{[FRea]}\ \frac{\begin{array}{c}\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}\\ \mathsf{A}' = \mathsf{read}\ \mathsf{r} \mathbin{;;} \mathsf{A}\end{array}}{\mathsf{F}[\![\mathsf{r}[\mathsf{e}]]\!] = \mathsf{A}'}$$

$$\text{[FDup]}\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}}{\mathsf{F}[\![\mathbf{dup}\ \mathsf{e}]\!] = \mathsf{A}} \qquad \text{[FUse]}\ \frac{\mathsf{A} = \mathsf{read}\ \mathsf{r} \mathbin{;;} \mathsf{use}\ \mathsf{r}}{\mathsf{F}[\![\mathbf{use}\ \mathsf{r}]\!] = \mathsf{A}} \qquad \text{[FPro]}\ \frac{\begin{array}{c}\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{A} = (R, W, U, C)\\ C' = R \cup W \cup U \cup C\end{array}}{\mathsf{F}[\![\mathbf{prom}^{\mathsf{fx}}\mathbin{<}\mathsf{t}\mathbin{>}\{\mathsf{e}\}]\!] = (R, \emptyset, U, C')}$$

$$\text{[FAss1]}\ \frac{\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}}{\mathsf{F}[\![\mathsf{x} = \mathsf{e}]\!] = \mathsf{A}} \qquad \text{[FAss2]}\ \frac{\begin{array}{c}\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}\\ \mathsf{A}' = \mathsf{A} \mathbin{;;} \mathsf{write}\ \mathsf{r}\end{array}}{\mathsf{F}[\![\mathsf{r} = \mathsf{e}]\!] = \mathsf{A}'} \qquad \text{[FWri]}\ \frac{\begin{array}{c}\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A} \qquad \mathsf{F}[\![\mathsf{e}']\!] = \mathsf{A}'\\ \mathsf{A}'' = \mathsf{read}\ \mathsf{r} \mathbin{;;} \mathsf{A} \mathbin{;;} \mathsf{A}' \mathbin{;;} \mathsf{write}\ \mathsf{r}\end{array}}{\mathsf{F}[\![\mathsf{r}[\mathsf{e}] = \mathsf{e}']\!] = \mathsf{A}''}$$

$$\text{[FRwr]}\ \frac{\begin{array}{c}\mathsf{F}[\![\mathsf{e}]\!] = \mathsf{A}\\ \mathsf{A}' = \mathsf{read}\ \mathsf{r} \mathbin{;;} \mathsf{A} \mathbin{;;} \mathsf{write}\ \mathsf{r}\end{array}}{\mathsf{F}[\![\mathsf{r\$x} = \mathsf{e}]\!] = \mathsf{A}'}$$

Fig. 9. Flow Analysis

$$\frac{\begin{array}{c}\mathsf{A} = (R, W, U, C)\\ \mathsf{A}' = (R', W', U', C')\\ U \cap (R' \cup W' \cup U' \cup C') = \emptyset\\ C \cap U' = \emptyset\end{array}}{\mathsf{A} \;;;\; \mathsf{A}' = (R \cup (R' - W), W \cup W', U \cup U', C \cup C')}$$

Our flow analysis runs on each subexpression of the body of an abstraction and composes their actions. Analysis fails if this action cannot be computed, or if its R set is not a subset of the abstraction parameters. The former means there is a use invariant violation (read/assign/capture after use, or capture before use), and the latter means there is a read before assign.

5.3 Dynamic Semantics

We endow FIŘ with a small-step operational semantics with evaluation contexts [7]. The program state is captured by a *heap* (H), mapping *references* (o) to values (O), written $\mathsf{o}_0 \mapsto \mathsf{O}_0 \,\ldots\, \mathsf{o}_n \mapsto \mathsf{O}_n$. References include `undef`, which does not occur on the left of a mapping.

A FIŘ expression that reduces to `undef` is analogous to an expression of the source language that raises an error or otherwise diverts control-flow. Hence, `undef` is considered an instance of every type, because code where it shows up would not be evaluated at all. This allows us to type failable operations, like indexing and casting, the same as if they cannot fail.

Values are defined as follows:

$$\begin{array}{lcll} \mathsf{O} & ::= & \mathsf{i} & \text{integers}\\ & | & \mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_n) & \text{mutable vectors}\\ & | & \langle \mathsf{e}, \mathsf{k}, \mathsf{o}\rangle & \text{unevaluated promises}\\ & | & \langle \mathsf{o}_1, \mathsf{k}, \mathsf{o}_2\rangle & \text{cached promises}\\ & | & (\mathsf{v}_0 \mapsto \mathsf{o}_0 \ldots \mathsf{v}_n \mapsto \mathsf{o}_n) & \text{environments} \end{array}$$

The syntax is extended with the following terms:

$$\begin{array}{lcll} \mathsf{e} & ::= & \ldots & \\ & | & \mathsf{o} & \text{references}\\ & | & \{\mathsf{e}\}_{\mathsf{o}} & \text{abstraction scope}\\ & | & \{\mathsf{e}\}_{\mathsf{o}}^{\mathsf{o}'} & \text{promise scope} \end{array}$$

Evaluation contexts are defined as follows:

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

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

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

The main judgments appear in Fig. 10. The semantics deal with out of bound arrays 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:

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

The above rule is prioritized over the other judgments. This treatment of `undef` simplifies the proofs of type safety: as usual, our type system focuses on preventing stuck states, and thus failable operations that are not caught by our type system should not lead to a stuck state, but instead should evaluate to `undef`.

A literal (DLit), promise constructor (DPro), or vector constructor (DVec) creates a fresh reference within the heap, then reduces to that reference. The other expressions that create fresh references are `dup` expressions (DDup) and applications (DApp1); a `dup` expression copies its argument into a fresh reference, while an application references a new environment (corresponding to the scope of the abstraction being called), initialized with the parameter values. This environment is unreferenced when the application returns (DApp2).

An assignment (DAss) maps a variable to a reference in the enclosing environment, while a reflective write (DRwr) maps a variable to a reference in the provided promise's environment. Notably, a vector write (DWri) does not remap the variable containing the vector reference, instead it mutates the vector in-place. This vector value must be referenced at most once in the heap, otherwise the rule does not apply and the reduction is stuck.

A variable expression (DVar) and `use` expression (DUse) looks up its variable in the enclosing environment, and a `use` expression additionally removes the variable mapping (since the used register variable cannot be accessed again, its mapping is redundant). A reflective read (DRrd) looks up its variable in the provided promise's environment.

A force expression whose argument is a cached promise (DFor1) reduces to the cached value. A force expression whose argument is an unevaluated promise (DFor2 and DFor3), reduces to the result of evaluating the promise's body in the promise's environment, and caches the result for future forces.

A static call reduces to an application, where the abstraction is looked up by function and version, and the application arguments are the call arguments. A dispatch call reduces to a static call, where the version to call is selected according to the signature annotation at the call site and the argument types; specifically, the arguments are evaluated to get their runtime types, and the selected version in the function is the first whose signature is a subtype of the annotated signature, and whose parameter types are satisfied by all of said argument types.

A cast expression (DCas) evaluates to its argument if the argument is of the correct kind, otherwise it reduces to `undef`. A sequence (DSeq) evaluates each expression in order and reduces to the last one. Lastly, DCtx1 and DCtx2 are both mechanical rules to evaluate promise or abstraction code in the appropriate context.

Some additional notation is described. o `fresh` denotes a reference o that does not occur in the heap; the env function extracts a reference to the environment 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 a value is an environment then $H(\mathsf{o})(\mathsf{v})$ 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 a mapping; $\mathtt{sig}(\mathsf{abs})$ returns the signature of abs; $F(\mathtt{f})(\mathtt{i})$ returns the version $\mathtt{i}$ of function $\mathtt{f}$; $\mathtt{refs}\ (H, \mathsf{o})$ returns the number of occurrences of o on the right hand-side of a mapping in the environments in H. Lastly, the $H \vdash \mathsf{o} \leq_{dyn} \mathtt{t}$ relation is defined as follows:

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

5.4 Type Safety

The static semantics (in particular the type checker and flow analysis described in Section 5.2) ensure the safety of the execution for the dynamic semantics: if a program is well-typed and

$$\text{[DLit]}\ \frac{\begin{array}{c}\mathtt{o}'\ \mathtt{fresh}\\ H' = H, \mathtt{o}' \mapsto \mathtt{i}\end{array}}{H\llbracket \mathtt{i} \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DPro]}\ \frac{\begin{array}{c}\mathtt{o}'\ \mathtt{fresh}\\ H' = H, \mathtt{o}' \mapsto \langle \mathtt{e}, \mathbf{p}(\mathtt{t}\ \mathtt{fx}), \mathtt{o}\rangle\end{array}}{H\llbracket \mathbf{prom}^{\mathtt{fx}}\texttt{<t>\{e\}} \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DVec]}\ \frac{\begin{array}{c}\mathtt{o}'\ \mathtt{fresh}\\ H(\mathtt{o}_1) = \mathtt{i}_1 \ldots H(\mathtt{o}_n) = \mathtt{i}_n\\ H' = H, \mathtt{o}' \mapsto \mathbf{vec}(\mathtt{i}_1 \ldots \mathtt{i}_n)\end{array}}{H\llbracket \mathbf{vec}(\mathtt{o}_1 \ldots \mathtt{o}_n) \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DVar]}\ \frac{\mathtt{o}' = H(\mathtt{o})(\mathtt{v})}{H\llbracket \mathtt{v} \rrbracket_{\mathtt{o}} \implies H\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DAss]}\ \frac{\begin{array}{c}\mathtt{O} = H(\mathtt{o}), \mathtt{v} \mapsto \mathtt{o}'\\ H' = H, \mathtt{o} \mapsto \mathtt{O}\end{array}}{H\llbracket \mathtt{v} = \mathtt{o}' \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DSeq]}\ \frac{}{H\llbracket \mathtt{o}'\ ;\ \mathtt{e} \rrbracket_{\mathtt{o}} \implies H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}}}$$

$$\text{[DDup]}\ \frac{\begin{array}{c}\mathtt{o}''\ \mathtt{fresh}\\ H' = H, \mathtt{o}'' \mapsto H(\mathtt{o}')\end{array}}{H\llbracket \mathbf{dup}\ \mathtt{o}' \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}'' \rrbracket_{\mathtt{o}}}$$

$$\text{[DUse]}\ \frac{\begin{array}{c}\mathtt{o}' = H(\mathtt{o})(\mathtt{r})\\ \mathtt{O} = H(\mathtt{o}) \setminus \mathtt{r}\\ H' = H, \mathtt{o} \mapsto \mathtt{O}\end{array}}{H\llbracket \mathbf{use}\ \mathtt{r} \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DCas]}\ \frac{\begin{array}{c}H \vdash \mathtt{o} \leq_{dyn} \mathtt{t} \Rightarrow \mathtt{o}' = \mathtt{o}\\ H \vdash \mathtt{o} \not\leq_{dyn} \mathtt{t} \Rightarrow \mathtt{o}' = \mathtt{undef}\end{array}}{H\llbracket \mathtt{o}'\ \mathbf{as}\ \mathtt{t} \rrbracket_{\mathtt{o}} \implies H\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DFor1]}\ \frac{H(\mathtt{o}') = \langle \mathtt{o}'', \mathtt{k}, \mathtt{o}''' \rangle}{H\llbracket \mathbf{force}\ \mathtt{o}' \rrbracket_{\mathtt{o}} \implies H\llbracket \mathtt{o}'' \rrbracket_{\mathtt{o}}}$$

$$\text{[DFor2]}\ \frac{\begin{array}{c}H(\mathtt{o}') = \langle \mathtt{e}, \mathtt{k}, \mathtt{o}'' \rangle\\ H' = H, \mathtt{o}' \mapsto \langle \mathtt{undef}, \mathtt{k}, \mathtt{o}'' \rangle\end{array}}{H\llbracket \mathbf{force}\ \mathtt{o}' \rrbracket_{\mathtt{o}} \implies H'\llbracket \{\mathtt{e}\}^{\mathtt{o}'}_{\mathtt{o}''} \rrbracket_{\mathtt{o}}}$$

$$\text{[DFor3]}\ \frac{H' = H, \mathtt{o}''' \mapsto \langle \mathtt{o}', \mathtt{k}, \mathtt{o}'' \rangle}{H\llbracket \{\mathtt{o}'\}^{\mathtt{o}'''}_{\mathtt{o}''} \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DDis]}\ \frac{\begin{array}{c}\text{smallest i such that}\\ \mathtt{sig}(F(\mathtt{f})(\mathtt{i})) = \mathtt{t}_0 \ldots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t}\\ \mathtt{t}_0 \ldots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \leq_{sig} \mathtt{sig}\\ H \vdash \mathtt{o}_0 \leq_{dyn} \mathtt{t}_0 \ldots H \vdash \mathtt{o}_n \leq_{dyn} \mathtt{t}_n\\ \mathtt{e} = \mathtt{f.i}(\mathtt{o}_0 \ldots \mathtt{o}_n)\end{array}}{H\llbracket \mathtt{f}\ \texttt{<sig>}\ (\mathtt{o}_0 \ldots \mathtt{o}_n) \rrbracket_{\mathtt{o}} \implies H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}}}$$

$$\text{[DApp1]}\ \frac{\begin{array}{c}\mathtt{o}'\ \mathtt{fresh}\\ \mathtt{abs} = (\mathtt{r}_0 : \mathtt{t}_0 \ldots \mathtt{r}_n : \mathtt{t}_n) \xrightarrow{\mathtt{fx}} \mathtt{t}\{\mathtt{defs};\ \mathtt{e}\}\\ \mathtt{O} = \mathtt{r}_0 \mapsto \mathtt{o}_0 \ldots \mathtt{r}_n \mapsto \mathtt{o}_n\\ H' = H, \mathtt{o}' \mapsto \mathtt{O}\end{array}}{H\llbracket \mathtt{abs} \leftarrow (\mathtt{o}_0 \ldots \mathtt{o}_n) \rrbracket_{\mathtt{o}} \implies H'\llbracket \{\mathtt{e}\}_{\mathtt{o}'} \rrbracket_{\mathtt{o}}}$$

$$\text{[DApp2]}\ \frac{H' = H \setminus \mathtt{o}''}{H\llbracket \{\mathtt{o}'\}_{\mathtt{o}''} \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\text{[DCal]}\ \frac{\begin{array}{c}F(\mathtt{f})(\mathtt{i}) = \mathtt{abs}\\ \mathtt{e} = \mathtt{abs} \leftarrow (\mathtt{o}_0 \ldots \mathtt{o}_n)\end{array}}{H\llbracket \mathtt{f.i}(\mathtt{o}_0 \ldots \mathtt{o}_n) \rrbracket_{\mathtt{o}} \implies H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}}}$$

$$\text{[DWri]}\ \frac{\begin{array}{c}H(\mathtt{o}') = \mathbf{vec}(\mathtt{i}_0 \ldots \mathtt{i}_n)\\ \mathtt{refs}(H, \mathtt{o}') \leq 1\\ H(\mathtt{o}'') = \mathtt{j} \qquad H(\mathtt{o}''') = \mathtt{i}\\ \mathtt{O} = \mathbf{vec}(\mathtt{i}_0 \ldots \mathtt{i}_{j-1}, \mathtt{i}, \mathtt{i}_{j+1} \ldots \mathtt{i}_n)\\ H' = H, \mathtt{o}' \mapsto \mathtt{O}\end{array}}{H\llbracket \mathtt{o}'[\mathtt{o}''] = \mathtt{o}''' \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}''' \rrbracket_{\mathtt{o}}}$$

$$\text{[DRea]}\ \frac{\begin{array}{c}H(\mathtt{o}') = \mathbf{vec}(\mathtt{i}_0 \ldots \mathtt{i}_n)\\ \mathtt{o}'''\ \mathtt{fresh} \qquad H(\mathtt{o}'') = \mathtt{j}\\ H' = H, \mathtt{o}''' \mapsto \mathtt{i}_\mathtt{j}\end{array}}{H\llbracket \mathtt{o}'[\mathtt{o}''] \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}''' \rrbracket_{\mathtt{o}}}$$

$$\text{[DRrd]}\ \frac{\begin{array}{c}\mathtt{o}'' = \mathtt{env}(H(\mathtt{o}'))\\ \mathtt{o}''' = H(\mathtt{o}'')(\mathtt{x})\end{array}}{H\llbracket \mathtt{o}'\$\mathtt{x} \rrbracket_{\mathtt{o}} \implies H\llbracket \mathtt{o}''' \rrbracket_{\mathtt{o}}}$$

$$\text{[DRwr]}\ \frac{\begin{array}{c}\mathtt{o}''' = \mathtt{env}(H, \mathtt{o}') \qquad \mathtt{O} = H(\mathtt{o}'), \mathtt{x} \mapsto \mathtt{o}''\\ H' = H, \mathtt{o}''' \mapsto \mathtt{O}\end{array}}{H\llbracket \mathtt{o}'\$\mathtt{x} = \mathtt{o}'' \rrbracket_{\mathtt{o}} \implies H'\llbracket \mathtt{o}'' \rrbracket_{\mathtt{o}}}$$

$$\text{[DCtx1]}\ \frac{H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}'} \implies H'\llbracket \mathtt{e}' \rrbracket_{\mathtt{o}'}}{H\llbracket \{\mathtt{e}\}_{\mathtt{o}'} \rrbracket_{\mathtt{o}} \implies H'\llbracket \{\mathtt{e}'\}_{\mathtt{o}'} \rrbracket_{\mathtt{o}}}$$

$$\text{[DCtx2]}\ \frac{H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}'} \implies H'\llbracket \mathtt{e}' \rrbracket_{\mathtt{o}'}}{H\llbracket \{\mathtt{e}\}^{\mathtt{o}''}_{\mathtt{o}'} \rrbracket_{\mathtt{o}} \implies H'\llbracket \{\mathtt{e}'\}^{\mathtt{o}''}_{\mathtt{o}'} \rrbracket_{\mathtt{o}}}$$

Fig. 10. Dynamic Semantics

well-flowed, then its reduction cannot get stuck (although it can reduce to undef). In this section, we provide an overview of the type safety proofs and the invariants that are preserved throughout execution. A more detailed proof is available in Appendix A (Theorem A.29).

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

The proof is decomposed in two parts. The first part (Appendix A.2) focuses on the type checker, while the second part (Appendix A.3) focuses on the flow analysis.

Safety of the type checker. The type checker alone does not guarantee type safety for the dynamic semantics described in Section 5.3. In particular, it does not ensure that registers are not read or **use**d before having been initialized (or after having been **use**d). Consequently, the type safety theorem we prove for the type checker applies to a modified version of the dynamic semantics, in which those two reduction rules are added:

$$\text{[DRegUndef]}\ \frac{\texttt{r} \notin \text{dom}(H(\texttt{o}))}{H[\![\texttt{r}]\!]_{\texttt{o}} \implies H[\![\texttt{undef}]\!]_{\texttt{o}}} \qquad \text{[DUseUndef]}\ \frac{\texttt{r} \notin \text{dom}(H(\texttt{o}))}{H[\![\textbf{use}\ \texttt{r}]\!]_{\texttt{o}} \implies H'[\![\texttt{undef}]\!]_{\texttt{o}}}$$

As undef propagates to the top level, the purpose of these two rules is to treat an invalid register read or **use** as an unchecked runtime error instead of a stuck reduction.

The proof of type safety is then decomposed into two lemmas: *type preservation*, that states that reduction steps preserve typeability, and *progress*, that states that a well-typed expression can be reduced (unless it is already a final result, that is, a reference o).

Proving these two lemmas requires extending our type system so that it can not only type expressions of the source language, but also transient expressions that we obtain during reduction and that may feature references o and context switches $\{\texttt{e}\}_{\texttt{o}'}$ and $\{\texttt{e}\}_{\texttt{o}'}^{\texttt{o}''}$. The idea is to give the current heap H as an additional input to the type system, and to type references according to this heap. Additionally, in order to type context switches, the type environment Γ for the current scope is replaced by a mapping Δ from scopes o to type environments Γ.

The type preservation can then be proved, relying on several invariants that are preserved throughout execution and are required for the type preservation lemma to be inductive:

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

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

Valid borrowing Any reference in H that has an owned type according to Δ cannot be used anywhere else in H, except in a more recent environment (i.e. an environment above in the call stack) in which it has a borrowed type according to Δ, and

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

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

A formal description of these invariants is given in Appendix A.2 (Definition A.7), as well as a proof of type preservation and progress.

Safety of the flow analysis. The flow analysis guarantees that registers are not read or **use**d before having been initialized or after having been **use**d. In other terms, it guarantees that, for any well-flowed expression, the two reduction rules DRegUndef and DUseUndef are never used.

Consequently, combining the guarantees of the type checker with those of the flow analysis gives type safety for the dynamic semantics of Section 5.3.

Similarly to the proofs for the type checker, we decompose the correctness of the flow analysis into two lemmas: *preservation of well-flowedness* and *correctness of well-flowedness*. The first one ensures that a well-flowed expression stays well-flowed after a reduction step, and the second ensures that DRegUndef and DUseUndef rules cannot apply on well-flowed expressions.

As for the type-checker, in order to satisfy these lemmas, the flow analysis has to be extended to support transient expressions that we obtain during reduction. For that, we reuse similar ideas, replacing actions A by mappings $\mathbb{A}$ from scopes o to actions A, and adding rules to handle the case of references and context switches. These additional rules are formalized in Appendix A.3, together with the associated proofs.

6 Conclusion

Dynamic languages pose many challenges for compiler writers due to features like dynamic typing, reflection, late binding, copy-on-write, and delayed evaluation. These features make it difficult to reason statically about program behavior.

This paper introduces FIř, a statically typed high-level intermediate representation designed to address these challenges. By exposing key dynamic behaviors explicitly (reflective reads and writes, dispatched calls, inlining, promise creations and evaluations, vector copies) and providing gradual types that combine information about the structure of values and their ownership, FIř simplifies the implementation of classical compiler optimizations such as specialization, inlining, scope elision, and copy elimination. It allows compiler writers to reason locally about code transformations without compromising global semantic correctness.

We have formalized the operational semantics of FIř, as well as a static semantics composed of a type system, a flow analysis, and an effect analysis. We proved that together, they ensure the safety of the execution (in the usual sense of type safety). Although the design of FIř is informed by the semantics of R, we believe it generalizes to a wide range of dynamic languages that exhibit similar behaviors.

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] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

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

[14] 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

[15] 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

[16] 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/

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

[18] 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

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

[20] 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

[21] 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

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

[23] 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

[24] 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

We start by giving the full set of rules for a small step semantics. This semantics is the same as the one given in Section 5.3, except for the two extra rules [DRegUndef] and [DUseUndef] (in blue). Reduction steps for this semantics are noted $H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{e}' \rrbracket_{\mathtt{o}}$. We note $H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}} \leadsto^{*} H'\llbracket \mathtt{e}' \rrbracket_{\mathtt{o}}$ a reduction of any number of steps, and $H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}} \leadsto^{\infty}$ a diverging reduction.

$$\textsc{[DLit]}\ \dfrac{\mathtt{o}' \text{ fresh} \qquad H' = H, \mathtt{o}' \mapsto \mathtt{i}}{H\llbracket \mathtt{i} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}} \qquad \textsc{[DPro]}\ \dfrac{\mathtt{o}' \text{ fresh} \qquad H' = H, \mathtt{o}' \mapsto \langle \mathtt{e}, \mathbf{p}(\mathtt{t\ fx}), \mathtt{o}\rangle}{H\llbracket \mathbf{prom}^{\mathtt{fx}}\texttt{<t>\{e\}} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DVec]}\ \dfrac{\mathtt{o}' \text{ fresh} \qquad H(\mathtt{o}_1) = \mathtt{i}_1 \dots H(\mathtt{o}_n) = \mathtt{i}_n \qquad H' = H, \mathtt{o}' \mapsto \mathbf{vec}(\mathtt{i}_1 \dots \mathtt{i}_n)}{H\llbracket \mathbf{vec}(\mathtt{o}_1 \dots \mathtt{o}_n) \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DVar]}\ \dfrac{\mathtt{o}' = H(\mathtt{o})(\mathtt{v})}{H\llbracket \mathtt{v} \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}} \qquad \textsc{[DNVarUndef]}\ \dfrac{\mathtt{x} \notin \mathrm{dom}(H(\mathtt{o}))}{H\llbracket \mathtt{x} \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{undef} \rrbracket_{\mathtt{o}}} \qquad \textsc{[DRegUndef]}\ \dfrac{\mathtt{r} \notin \mathrm{dom}(H(\mathtt{o}))}{H\llbracket \mathtt{r} \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{undef} \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DDup]}\ \dfrac{\mathtt{o}'' \text{ fresh} \qquad H' = H, \mathtt{o}'' \mapsto H(\mathtt{o}')}{H\llbracket \mathbf{dup}\ \mathtt{o}' \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}'' \rrbracket_{\mathtt{o}}} \qquad \textsc{[DUse]}\ \dfrac{\mathtt{o}' = H(\mathtt{o})(\mathtt{r}) \qquad H' = H, \mathtt{o} \mapsto H(\mathtt{o}) \setminus \mathtt{r}}{H\llbracket \mathbf{use}\ \mathtt{r} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}} \qquad \textsc{[DUseUndef]}\ \dfrac{\mathtt{r} \notin \mathrm{dom}(H(\mathtt{o}))}{H\llbracket \mathbf{use}\ \mathtt{r} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{undef} \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DFor1]}\ \dfrac{H(\mathtt{o}') = \langle \mathtt{o}'', \mathtt{k}, \mathtt{o}''' \rangle}{H\llbracket \mathbf{force}\ \mathtt{o}' \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{o}'' \rrbracket_{\mathtt{o}}} \qquad \textsc{[DFor2]}\ \dfrac{H(\mathtt{o}') = \langle \mathtt{e}, \mathtt{k}, \mathtt{o}'' \rangle \qquad H' = H, \mathtt{o}' \mapsto \langle \mathtt{undef}, \mathtt{k}, \mathtt{o}'' \rangle}{H\llbracket \mathbf{force}\ \mathtt{o}' \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \{\mathtt{e}\}^{\mathtt{o}'}_{\mathtt{o}''} \rrbracket_{\mathtt{o}}} \qquad \textsc{[DFor3]}\ \dfrac{H' = H, \mathtt{o}''' \mapsto \langle \mathtt{o}', \mathtt{k}, \mathtt{o}'' \rangle}{H\llbracket \{\mathtt{o}'\}^{\mathtt{o}'''}_{\mathtt{o}''} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DApp1]}\ \dfrac{\mathtt{o}' \text{ fresh} \qquad \mathtt{abs} = (\mathtt{r}_0 : \mathtt{t}_0 \dots \mathtt{r}_n : \mathtt{t}_n) \xrightarrow{\mathtt{fx}} \mathtt{t}\{\mathtt{defs};\ \mathtt{e}\} \qquad O = \mathtt{r}_0 \mapsto \mathtt{o}_0 \dots \mathtt{r}_n \mapsto \mathtt{o}_n \qquad H' = H, \mathtt{o}' \mapsto O}{H\llbracket \mathtt{abs} \leftarrow (\mathtt{o}_0 \dots \mathtt{o}_n) \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \{\mathtt{e}\}_{\mathtt{o}'} \rrbracket_{\mathtt{o}}} \qquad \textsc{[DApp2]}\ \dfrac{H' = H \setminus \mathtt{o}''}{H\llbracket \{\mathtt{o}'\}_{\mathtt{o}''} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DCal]}\ \dfrac{F(\mathtt{f})(\mathtt{i}) = \mathtt{abs} \qquad \mathtt{e} = \mathtt{abs} \leftarrow (\mathtt{o}_0 \dots \mathtt{o}_n)}{H\llbracket \mathtt{f.i}(\mathtt{o}_0 \dots \mathtt{o}_n) \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DDis]}\ \dfrac{\begin{array}{c}\text{smallest i such that} \\ \mathtt{sig}(F(\mathtt{f})(\mathtt{i})) = \mathtt{t}_0 \dots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \qquad \mathtt{t}_0 \dots \mathtt{t}_n \xrightarrow{\mathtt{fx}} \mathtt{t} \leq_{sig} \mathtt{sig} \\ H \vdash \mathtt{o}_0 \leq_{dyn} \mathtt{t}_0 \dots H \vdash \mathtt{o}_n \leq_{dyn} \mathtt{t}_n \qquad \mathtt{e} = \mathtt{f.i}(\mathtt{o}_0 \dots \mathtt{o}_n)\end{array}}{H\llbracket \mathtt{f}\ \texttt{<sig>}\ (\mathtt{o}_0 \dots \mathtt{o}_n) \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DWriUndef]}\ \dfrac{H(\mathtt{o}_1) = \mathbf{vec}(\mathtt{i}_0 \dots \mathtt{i}_n) \qquad H(\mathtt{o}_2) = \mathtt{j} \qquad \mathtt{j} < 0 \text{ or } \mathtt{j} > \mathtt{n}}{H\llbracket \mathtt{o}_1[\mathtt{o}_2] = \mathtt{o}_3 \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{undef} \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DWri]}\ \dfrac{\begin{array}{c} H(\mathtt{o}') = \mathbf{vec}(\mathtt{i}_0 \dots \mathtt{i}_n) \qquad \mathtt{refs}(H, \mathtt{o}') \leq 1 \\ H(\mathtt{o}'') = \mathtt{j} \qquad H(\mathtt{o}''') = \mathtt{i} \\ O = \mathbf{vec}(\mathtt{i}_0 \dots \mathtt{i}_{\mathtt{j}-1}, \mathtt{i}, \mathtt{i}_{\mathtt{j}+1} \dots \mathtt{i}_n) \qquad H' = H, \mathtt{o}' \mapsto O \end{array}}{H\llbracket \mathtt{o}'[\mathtt{o}''] = \mathtt{o}''' \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}''' \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DReaUndef]}\ \dfrac{H(\mathtt{o}_1) = \mathbf{vec}(\mathtt{i}_0 \dots \mathtt{i}_n) \qquad H(\mathtt{o}_2) = \mathtt{j} \qquad \mathtt{j} < 0 \text{ or } \mathtt{j} > \mathtt{n}}{H\llbracket \mathtt{o}_1[\mathtt{o}_2] \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{undef} \rrbracket_{\mathtt{o}}} \qquad \textsc{[DRea]}\ \dfrac{H(\mathtt{o}') = \mathbf{vec}(\mathtt{i}_0 \dots \mathtt{i}_n) \qquad \mathtt{o}''' \text{ fresh} \qquad H(\mathtt{o}'') = \mathtt{j} \qquad H' = H, \mathtt{o}''' \mapsto \mathtt{i}_{\mathtt{j}}}{H\llbracket \mathtt{o}'[\mathtt{o}''] \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}''' \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DAss]}\ \dfrac{O = H(\mathtt{o}), \mathtt{v} \mapsto \mathtt{o}' \qquad H' = H, \mathtt{o} \mapsto O}{H\llbracket \mathtt{v} = \mathtt{o}' \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}} \qquad \textsc{[DRrd]}\ \dfrac{\mathtt{o}'' = \mathrm{env}(H(\mathtt{o}')) \qquad \mathtt{o}''' = H(\mathtt{o}'')(\mathtt{x})}{H\llbracket \mathtt{o}'\$\mathtt{x} \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{o}''' \rrbracket_{\mathtt{o}}} \qquad \textsc{[DRwr]}\ \dfrac{\mathtt{o}''' = \mathrm{env}(H, \mathtt{o}') \qquad O = H(\mathtt{o}'), \mathtt{x} \mapsto \mathtt{o}'' \qquad H' = H, \mathtt{o}''' \mapsto O}{H\llbracket \mathtt{o}'\$\mathtt{x} = \mathtt{o}'' \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{o}'' \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DCas]}\ \dfrac{H \vdash \mathtt{o} \leq_{dyn} \mathtt{t} \Rightarrow \mathtt{o}' = \mathtt{o} \qquad H \vdash \mathtt{o} \not\leq_{dyn} \mathtt{t} \Rightarrow \mathtt{o}' = \mathtt{undef}}{H\llbracket \mathtt{o}'\ \mathbf{as}\ \mathtt{t} \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{o}' \rrbracket_{\mathtt{o}}} \qquad \textsc{[DSeq]}\ \dfrac{}{H\llbracket \mathtt{o}'\ ;\ \mathtt{e} \rrbracket_{\mathtt{o}} \leadsto H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}}} \qquad \textsc{[DCtx1]}\ \dfrac{H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}'} \leadsto H'\llbracket \mathtt{e}' \rrbracket_{\mathtt{o}'}}{H\llbracket \{\mathtt{e}\}_{\mathtt{o}'} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \{\mathtt{e}'\}_{\mathtt{o}'} \rrbracket_{\mathtt{o}}}$$

$$\textsc{[DCtx2]}\ \dfrac{H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}'} \leadsto H'\llbracket \mathtt{e}' \rrbracket_{\mathtt{o}'}}{H\llbracket \{\mathtt{e}\}^{\mathtt{o}''}_{\mathtt{o}'} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \{\mathtt{e}'\}^{\mathtt{o}''}_{\mathtt{o}'} \rrbracket_{\mathtt{o}}} \qquad \textsc{[DCtx3]}\ \dfrac{H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{undef} \rrbracket_{\mathtt{o}}}{H\llbracket \mathcal{E}[\mathtt{e}] \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{undef} \rrbracket_{\mathtt{o}}} \qquad \textsc{[DCtx4]}\ \dfrac{H\llbracket \mathtt{e} \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathtt{e}' \rrbracket_{\mathtt{o}}}{H\llbracket \mathcal{E}[\mathtt{e}] \rrbracket_{\mathtt{o}} \leadsto H'\llbracket \mathcal{E}[\mathtt{e}'] \rrbracket_{\mathtt{o}}}$$

A.2 Proofs for the static semantics

In this section, we prove that the type system defined in Section 5.2 ensures type safety ("well-typed programs cannot go wrong") for the small-step semantics defined above. However, due to the presence of the two reduction rules [DRegUndef] and [DUseUndef], this type safety theorem does not ensure that registers are correctly initialized. Ensuring a correct initialization of the registers is done by the flow analysis, and the associated proofs are available in the next section.

Definition A.1 (Well-typed abstraction). An abstraction abs is well-typed if and only if there exists $\mathsf{t}_0 \ldots \mathsf{t}_n, \mathsf{t}$ such that $[\![\mathsf{abs}]\!] : \mathsf{t}_0 \ldots \mathsf{t}_n \xrightarrow{\mathsf{fx}} \mathsf{t}$ holds.

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

Our type safety theorem will be proved by first proving a type preservation lemma (Lemma A.9), stating that typeability is preserved by reduction, and then a progress lemma (Lemma A.10), stating that a well-typed expression is either a reference o or can be reduced. However, before proving these lemmas, we have to extend the type system so that it can type intermediate results: in particular, we must be able to type references o, abstraction scopes $\{\mathsf{e}\}_\mathsf{o}$, and promise scopes $\{\mathsf{e}\}_\mathsf{o}^{\mathsf{o}'}$. For that, our extended type system needs to take as input the current heap and a heap descriptor:

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 O as follows:

$$\mathsf{dynkind}(i) = \mathbf{I} \qquad \mathsf{dynkind}(\mathbf{vec}(\mathsf{i}_0 \ldots \mathsf{i}_n)) = \mathbf{v}(\mathbf{I}) \qquad \mathsf{dynkind}(\langle \_, \mathsf{k}, \_ \rangle) = \mathsf{k}$$

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

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

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

where $\mathsf{owner}(H, \Delta, \mathsf{o}) = \{\mathsf{o}' \mid \exists \mathsf{r}.\, H(\mathsf{o}')(\mathsf{r}) = \mathsf{o} \text{ and } \mathsf{owned}(\Delta(\mathsf{o}')(\mathsf{r}))\}$ and o' fresh is true if and only if o' does not appear in H except at the left-hand side of a binding, and does not have any other occurrence in the whole expression.

We omit the other typing rules as they can trivially be derived from the typing rules defined in Section 5.2. The additional inputs H and o are just passed recursively through the derivation. Note that the rule [Abs] (of judgement $[\![\mathsf{abs}]\!] : \mathsf{t}_0 \ldots \mathsf{t}_n \xrightarrow{\mathsf{fx}} \mathsf{t}$) does not require any additional parameters: as we do not reduce under abstractions, no reference o, abstraction scope or promise scope can appear in the body of an abstraction, and thus the rule [Abs] defined in Section 5.2 can be reused as is (typing the body with the non-extended version of the type system):

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

Proposition A.4. *If* $\Gamma \vdash [\![\mathtt{e}]\!] : \mathtt{t}$ *(typing judgement from Section 5.2), then for any* $\mathtt{o}$*,* H*,* Δ *such that* $\Delta(\mathtt{o}) = \Gamma$*, we have* $\Delta, H \vdash_{\mathtt{o}} [\![\mathtt{e}]\!] : \mathtt{t}$.

Proof. Straightforward. □

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

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

We define call contexts C as follows:

$$C \quad ::= \quad \dots \quad | \quad \{C\}^{\mathtt{o}}_{\mathtt{o}}$$

where $\dots$ includes all the cases of the grammar of $\mathcal{E}$ (defined in Section 5.3).

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

A call stack (i.e. a sequence of references) can be noted $\vec{\mathtt{o}}$. The concatenation of two call stacks $\vec{\mathtt{o}}$ and $\vec{\mathtt{o}}'$ is noted $\vec{\mathtt{o}}, \vec{\mathtt{o}}'$.

Definition A.7 (Heap validity). We say that the heap H is valid for the heap descriptor Δ and for the call stack $\mathtt{o}_1, \dots, \mathtt{o}_n$ if and only if we have:

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

Valid types for every $\mathtt{i}$, $H \setminus \{\mathtt{o}_{\mathtt{i}+1}, \dots, \mathtt{o}_n\}$ matches $\Delta(\mathtt{o}_{\mathtt{i}})$ at reference $\mathtt{o}_{\mathtt{i}}$, and

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

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

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

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

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

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

$\{\mathsf{e}_1\}^{\mathsf{o}_3}_{\mathsf{o}_2}$ 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 $\mathsf{e}' = \mathsf{undef}$, 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 $\mathsf{dynkind}(H(\mathsf{o}')) \leq \mathsf{kind}(\mathsf{t}')$, 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.

$\mathsf{abs} \leftarrow (\mathsf{o}_0 \ldots \mathsf{o}_n)$ The rule [DApp1] applies. We must type the body of abs, which can easily be done by extracting the subderivation of the body from $[\![\mathsf{abs}]\!] : \mathsf{t}_0 \ldots \mathsf{t}_n \xrightarrow{\mathsf{fx}} \mathsf{t}$ and applying Proposition A.4 to type the body under the new heap descriptor $\Delta, \mathsf{o}' \mapsto \Gamma$ with $\Gamma = \mathsf{rdefs}, \mathsf{defs}$.

$\mathsf{f.i}(\mathsf{o}_0 \ldots \mathsf{o}_n)$ The rule [DCal] applies. We can conclude using the fact that abstractions in F are well-typed, and using Proposition A.4.

$\mathsf{f} \, \mathsf{<sig>} \, (\mathsf{o}_0 \ldots \mathsf{o}_n)$ The rule [DDis] applies. Using the fact that the abstraction selected by the [DDis] rule has a smaller signature than the abstraction used in our typing derivation, this case can be concluded similarly to the previous case.

$\mathsf{o}'\$\mathsf{x} = \mathsf{o}''$ 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* $(H, \Delta, \vec{\mathsf{o}}, \mathsf{e})$ *be a valid quadruplet. If* $\Delta, H \vdash_{\mathsf{o}} [\![\mathsf{e}]\!] : \mathsf{t}$*, then either* e *is a reference* o' *or* $H[\![\mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{e}']\!]_{\mathsf{o}}$ *for some* H' *and* e'.

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

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.

$\mathsf{o}_1[\mathsf{o}_2] = \mathsf{o}_3$ Our typing derivation must end with a [Wri] rule, which requires o_1 to have type $\mathbf{v}(\mathbf{I})\mathbf{o}!$. It yields that $\mathsf{dynkind}(H(\mathsf{o}_1)) \leq \mathbf{v}(\mathbf{I})$ and thus $H(\mathsf{o}_1)$ is a vector. From $\mathsf{owner}(H, \Delta, \mathsf{o}') = \{\mathsf{o}\}$ and the *valid types* property, we can deduce $\mathsf{refs}(H, \mathsf{o}') \leq 1$, which allows to conclude that either the rule [DWri] or [DWriUndef] applies.

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

f <sig> $(o_0 \dots o_n)$ 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 $sig(F(f)(i)) = t_0 \dots t_n \xrightarrow{fx} t'$ 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 $\varnothing \vdash [\![f.i()]\!] : t$, *then either* $\varnothing[\![f.i()]\!]_o \rightsquigarrow^\infty$ *or* $\varnothing[\![f.i()]\!]_o \rightsquigarrow^* H'[\![o']\!]_o$ *for some* H' *and* o′ *such that* $H' \vdash o' \leq_{dyn} t$.

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 abstraction abs in our function table F is well-typed.

A.3 Proofs for the flow analysis

In this section, we prove that the flow analysis defined in Section 5.2 ensures a correct initialization of the registers. More precisely, what we want to show is that if an expression is well-flowed, then the two reduction rules [DRegUndef] and [DUseUndef] cannot apply after any number of steps. By combining this result with the type safety of the previous section, it yields a type safety theorem for the dynamic semantics of the paper (Section 5.3), where an access to an uninitialized register results in the reduction being stuck.

Definition A.12 (Well-flowed abstraction). An abstraction abs is well-flowed if and only if $\mathsf{F}[\![abs]\!]$ holds.

Definition A.13 (Well-defined abstraction). An abstraction abs is well-defined if and only if abs is well-typed and well-flowed.

In the following, we assume that every abstraction abs in our function table F is well-defined.
The correctness of the flow analysis will be proved by first introducing a more general version of it that can be applied not only to the source expression but also to the intermediate results of a reduction. We will then prove that the action returned by this generalized flow analysis is preserved by reduction (Lemma A.25), and that an expression that is well-flowed according to this flow analysis cannot be reduced using the rule [DRegUndef] or [DUseUndef] (Lemma A.27).

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

Proof. Let $A_1 = (R_1, W_1, U_1, C_1)$, $A_2 = (R_2, W_2, U_2, C_2)$, and $A_3 = (R_3, W_3, U_3, C_3)$. The expressions $(A_1 \;;;\; A_2) \;;;\; A_3$ and $A_1 \;;;\; (A_2 \;;;\; A_3)$ 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 respects 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 note $\emptyset$ the heap action mapping every reference to the action $(\varnothing, \varnothing, \varnothing, \varnothing)$. We note $(\mathsf{o} \mapsto \mathsf{A})$ the heap action mapping o to A and every other reference to the action $(\varnothing, \varnothing, \varnothing, \varnothing)$.

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 5.2 is extended. It now takes a reference o as additional input, and returns heap actions:

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

We omit the other rules as they can trivially be derived from the rules defined in Section 5.2: the additional input o is just passed recursively through the derivation, and each action A returned becomes $\mathsf{o} \mapsto \mathsf{A}$, for instance:

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

Note that the rule [FAbs] (of judgement $\mathsf{F}[\![\mathsf{abs}]\!]$) does not require any additional parameters: as we do not reduce under abstractions, no reference o, abstraction scope or promise scope can appear in the body of an abstraction, and thus the rule [FAbs] defined in Section 5.2 can be reused as is (checking the body with the non-extended version of the flow analysis):

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

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. Follow 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:

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

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):

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

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:

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

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 $\mathtt{action}(H, \mathsf{e}, \mathsf{o}) = \mathbb{A}$ *and* $H[\![\mathsf{e}]\!]_{\mathsf{o}} \rightsquigarrow H'[\![\mathsf{e}']\!]_{\mathsf{o}}$, *then* $\mathtt{action}(H', \mathsf{e}', \mathsf{o}) \le \mathbb{A}$.

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

- o' Impossible case as no reduction step can apply on o'.
- $\mathbf{prom}^{\mathsf{fx}}\mathtt{<t>\{e\}}$ 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 $\mathtt{pending}(H)$, but in the same time it gets removed from $\mathsf{F}[\![\mathsf{e}]\!]_{\mathsf{o}}$. We can conclude by using the commutativity of delayed actions (Proposition A.21).
- $\{\mathsf{o}_1\}^{\mathsf{o}_3}_{\mathsf{o}_2}$ 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 trivially conclude as adding a cached promise to the heap has no effect on the pending actions.
- $\{\mathsf{e}_1\}^{\mathsf{o}_3}_{\mathsf{o}_2}$ The reduction rule [DCtx2] applies, we conclude by applying the induction hypothesis on e_1.
- $\mathbf{force}\ \mathsf{o}'$ The rule [DFor1] or [DFor2] applies. In the first case, we can conclude immediately. Otherwise, we have $\mathsf{e}' = \{\mathsf{e}''\}^{\mathsf{o}'}_{\mathsf{o}''}$ and $H' = H, \mathsf{o}' \mapsto \langle \mathsf{undef}, \mathsf{k}, \mathsf{o}'' \rangle$, meaning that the delayed action of e'' gets removed from $\mathtt{pending}(H)$ (as the promise is updated with the cache $\mathtt{undef}$), and the action of e'' is added to $\mathsf{F}[\![\mathsf{e}]\!]_{\mathsf{o}}$ instead. We can conclude using Proposition A.20.
- $\mathtt{f.i}(\mathsf{o}_0 \ldots \mathsf{o}_n)$ The rule [DCal] applies. We can conclude using the fact that abstractions in F are well-flowed, and using Proposition A.18.
- $\mathsf{o}_1\ ;\ \mathsf{e}_2$ The rule [DSeq] applies. We thus have $e' = \mathsf{e}_2$. We can trivially conclude this case.
- $\mathsf{e}_1\ ;\ \mathsf{e}_2$ The rule [DCtx3] or [DCtx4] applies. In the first case, we can immediately conclude. Otherwise, we have $e' = \mathsf{e}'_1\ ;\ \mathsf{e}_2$. We apply the induction hypothesis on e_1, yielding $\mathtt{action}(H', \mathsf{e'}_1, \mathsf{o}) \le \mathtt{action}(H, \mathsf{e}_1, \mathsf{o})$. We conclude by using the monotonicity of the composition operator (Proposition A.16).

□

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

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

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

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

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 abstraction $\mathtt{abs}$ in our function table F is well-defined.

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

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