Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics

Casper Bach Poulsen[1], Pierre Néron[2], Andrew Tolmach[3], and Eelco Visser[4]

1 **Delft University of Technology**
`c.b.poulsen@tudelft.nl`
2 **French Network and Information Security Agency (ANSSI)**
`pierre.neron@ssi.gouv.fr`
3 **Portland State University**
`tolmach@pdx.edu`
4 **Delft University of Technology**
`visser@acm.org`

Abstract

Semantic specifications do not make a systematic connection between the names and scopes in the static structure of a program and memory layout, and access during its execution. In this paper, we introduce a systematic approach to the alignment of names in static semantics and memory in dynamic semantics, building on the scope graph framework for name resolution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachability-based garbage collection.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Dynamic semantics, scope graphs, memory layout, type soundness, operational semantics

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Name binding and memory management are pervasive concerns in programming language design. There is clearly a connection between the names and scopes in the static structure of a program and patterns of memory allocation, access, and deallocation during its execution. However, existing semantic specifications of programming languages do not treat this connection systematically, and take a wide variety of approaches to handling name binding and memory management. For example, different type soundness proofs for Java-like languages might model types and memory using simple states that map identifiers and references to values [3], untyped frames that rely on traditional environments for typing [24], or ad-hoc lookup functions (or visibility predicates) designed to resolve particular kinds of identifiers such as classes or fields [6, 9]. Existing dynamic semantic specification frameworks (Redex [8], Ott [21], K [20], funcons [2]) provide little guidance in formalizing the connection between static names and dynamic memory.

In the *language designer's workbench* project [27], we are pursuing high-level declarative language specifications that are amenable to both verification and implementation using

language-independent techniques. A particular goal is to separate the specifications into distinct aspects so far as possible, such as syntax, binding, typing, run-time behavior, etc. In this context we have developed *scope graphs* [14, 25] to provide a language-independent framework for static name binding and name resolution. The framework holds promise for dealing with name binding in a uniform way that scales to many binding-aware tools, such as editor services, refactoring transformations, type checkers, interpreters, and compilers.

In this paper, we address the run-time aspect of specification, and develop the relationship between static scope graphs and the *dynamic* semantics of name binding and run-time memory. Specifically, we formalize a language-independent correspondence between static scopes and *heap-allocated frames*. The approach is designed to support both verification of meta-theoretic properties and generation of realistic interpreters using fairly low-level memory operations. Our framework is *language-independent* in the sense of providing a generic set of facilities for describing and implementing binding and memory operations for a wide range of languages. We make the following contributions:

- We introduce the *scopes-as-frames* paradigm as a language-independent approach to dealing with name binding for both static and dynamic programming language semantics, by organizing frames in a run-time heap in correspondence to the organization of scopes in a scope graph, and relating static resolution paths to run-time memory access paths.
- We show how frame operations can be used in formulating big-step dynamic semantics, which resemble realistic interpreters. (The generation of high-performance frame-based interpreters from specifications in the DynSem operational semantics DSL [26] is work in progress.)
- We show how phrasing the consistency between frames and scopes as an invariant gives a language-independent principle that helps structure *type soundness* proofs by providing generic lemmas about preservation of the invariant under various memory operations.
- We demonstrate that scopes-as-frames is independent of particular language features by applying it to a functional language (Section 3), an imperative language with records (Section 4), and a class-based language with subtyping (Section 4.4). The same approach (scopes describing frames) is applied systematically to describe call frames, records, and objects instantiating a class.
- We give a high-level, language-independent specification of *sound garbage collection* of frames, and verify that several standard GC strategies refine this specification.
- We provide a Coq development[1] containing mechanized type soundness proofs for (supersets of) all languages in the paper. The type soundness proofs rely on a language-independent library formalizing scope graphs and frame heaps.

In the next section we introduce the approach by means of an example. In Section 3 we use the approach to specify and prove type soundness for a small language with first-class functions. In Section 4 we extend the formalization to a language with records. In Section 5 we discuss the formalization of garbage collection. In Section 6 we discuss related work. Section 7 summarizes this work and outlines ideas for future work.

2 Static Scopes Describe Dynamic Frames

Our approach builds on the theory of name resolution of Néron et al. [14, 25], which provides the foundation for a language-independent representation of static name binding in programs

[1] `https://github.com/metaborg/scopes-describe-frames/`

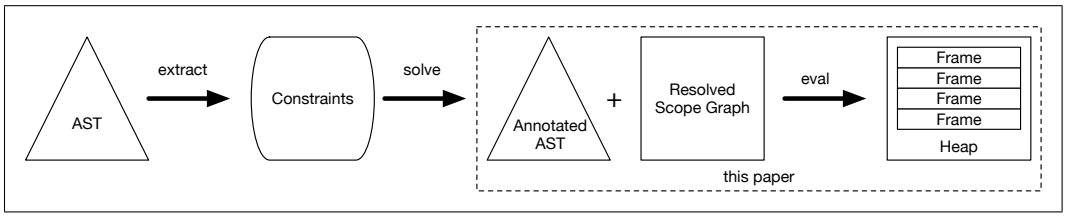

Figure 1 Architecture of the approach: static analysis of a program abstract syntax tree via constraints leads to a representation with explicit name and type information, which is the input for evaluation.

based on *scope graphs*. This section recalls the basics of scope graphs, and introduces a new paradigm for name binding at run time based on scope graphs.

2.1 Architecture

We first discuss how this work ties in with previous work on scope graphs. Figure 1 shows the overall architecture of the approach. We assume a program is represented by an abstract syntax tree (AST) produced by a parser. First, a set of name and type constraints is extracted from the AST. Second, a (language-independent) name and type resolution algorithm resolves the constraints and produces an annotated abstract syntax tree and a resolved scope graph [25]. Third, an interpreter evaluates this data structure using a run-time heap consisting of frames to represent the memory of the program.

This paper focuses on the third stage in Figure 1. Our starting point is a *well-bound* and *well-typed* program, represented as an *annotated AST* (where each AST node is annotated with *scope*- and *type annotations*) and a *resolved scope graph* (e.g., a scope graph produced by name and type resolution as described in [25]). Section 3 defines these notions formally. Before diving into the formal details, we illustrate the concepts of scope graphs and frame heaps by example.

2.2 Scope Graphs

Scope graphs [14, 25] provide a language-independent framework for describing static name binding and performing name resolution, which can support a wide variety of binding constructs. We recall the basic concepts of scope graphs here, starting with a simple but illustrative example.

Figure 2 presents a program that implements the factorial function and computes `fac(2)`. The figure also presents the scope graph of the program. The nodes of the scope graph represent three basic notions derived from the program abstract syntax tree: *scopes*, *declarations*, and *references*:

- A *scope* is an abstraction of a set of AST nodes that behave uniformly with respect to name binding.
- A *declaration* is an occurrence of an identifier that introduces a name.
- A *reference* is an occurrence of an identifier referring to a declaration.

A declaration is only visible in the scope in which it is declared and in other scopes connected to the declaration scope by a sequence of inter-scope edges. These edges represent visibility idioms such as nesting in lexical scoping and imports in module systems. A key idea is that scope graphs provide a means of resolving which declaration each reference in the AST refers

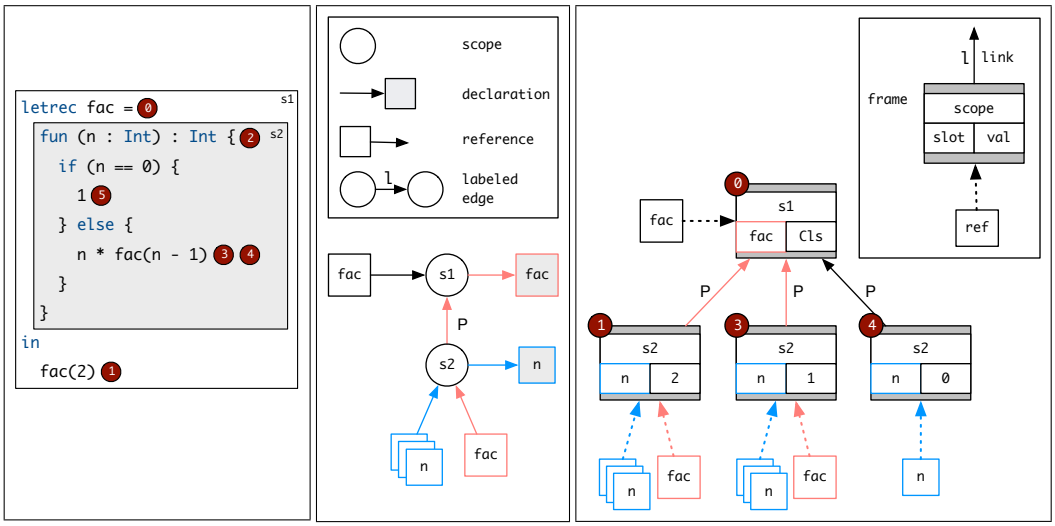

Figure 2 The left box contains an example program defining the factorial function. The boxes surrounding code blocks represent scopes. The call-outs refer to points in the execution of the program. The middle box shows the scope graph for this program; the graphical notation is explained in the legend. Distinct occurrences of similarly named references are stacked for conciseness (e.g., scope `s2` contains three distinct `n` references). The right box shows the evolution of the heap as the program executes: there is one top-level frame and three call-frames for the `fac` function.

to, by finding a *resolution path* from a reference to a declaration. Thus, a *reference* represents the use of a name in a scope and is *resolved* via a path along the edges of the scope graph.

In the example in Figure 2, `n` is both declared in and referenced from scope `s2`, which is associated with the function body; the blue resolution paths involve only this scope. Lexical nesting of `s2` within the outer scope `s1` is modeled by an edge labeled P (for lexical **p**arent); the reference `fac` in scope `s2` resolves to declaration `fac` in scope `s1` via the pink path.

If a given identifier is declared in more than one place, there may be multiple possible resolution paths for references to that identifier. In such cases, we choose the preferred path using a language-dependent *specificity ordering* and *path well-formedness* predicate. The resolution calculus introduced in [14] and refined in [25] defines the semantics of resolution and describes algorithms for finding most-specific resolution paths. In this paper, we assume that we already have a *resolved scope graph* in which each reference has a unique path to a corresponding declaration within the graph.

Records and field access. A key benefit of scope graphs is that they can describe many different language binding constructs in a uniform way. As a small illustration of this, we consider a language with records and named fields. Figure 3 shows an example program that defines a record type `A` with a single field `n`. The program declares two local variables, `x` and `y`. Each of these variables is first assigned new record instances of `A`; subsequently, the fields of the records stored in the variables are initialized.

The scope graph of the program in Figure 3 contains five scopes. The program scope `s1` contains the record type declaration `A`, where `A` has an *associated scope* `s2`, which is the record scope containing the record field declaration. Scope `s3` declares the local variables `x` and `y`. The two scopes `s4` and `s5` are *import scopes* containing the field references `x` and `y`, respectively. These scopes do not contain any declarations, but instead provide access to the fields of records via I-labeled edges. (Building these edges requires resolving the *types* of `x`

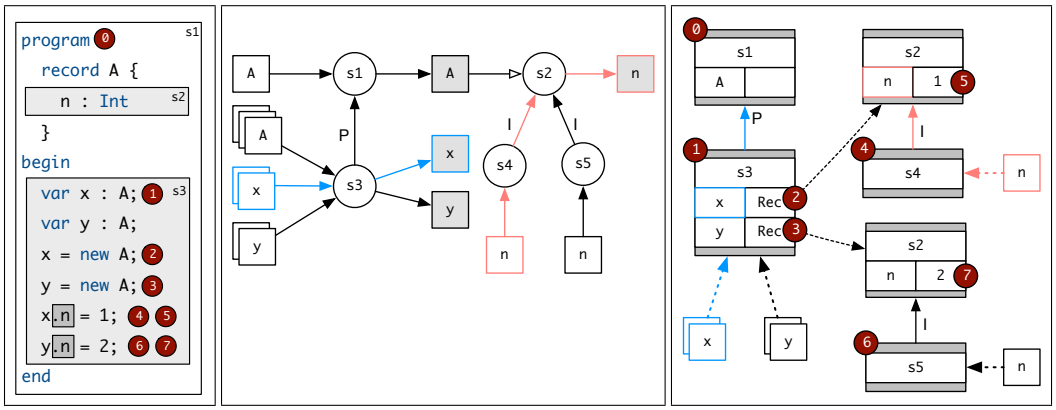

Figure 3 An example program using records (left) with its scope graph (center) and a picture of the evolution of the heap during execution (right).

and `y`; again, in this paper we assume that this resolution has already occurred, e.g. using the methods of [25].) Resolution of field names uses the same paradigm as for variables. For example, the reference to `n` in `s4` is resolved to the declaration in `s2` via the pink path.

2.3 Dynamic Frames and Heaps

We have illustrated how scope graphs provide a uniform and language-independent model for *static* name binding. In this paper, we propose a uniform and language-independent model for the layout of memory and the binding of values to names at run time. In this model, the static scope graph provides a blueprint for the layout of memory. The model is based on the notions of *frames*, *slots*, and *heaps*. Frames and heaps provide building blocks for structuring memory that are inspired by how realistic implementations of programming languages organize memory using, e.g., *stack frames* (for function calls), and *heap blocks* (for structured memory objects).

- The memory of a program at run time is a *heap* consisting of *frames*.
- A *frame* is a unit of memory consisting of *slots* mapping names to values, and *links* to other frames.
- A frame is described by a scope in the sense that each slot corresponds to a scope declaration and each link corresponds to an outgoing scope edge.
- Frames may be allocated when control passes to a program point in a new lexical scope (e.g. on function entry) or when executing an explicit memory allocation command (e.g. **new**). Frames can be garbage-collected when no longer referenced.
- The path for a reference in the static scope graph can be applied directly to fetch the referenced value at run time by interpreting it relative to the "current" frame.

The connection between scopes and frames enables us to define invariants that formally guarantee the absence of run-time errors, as Sections 3 to 5 of this paper demonstrate. In the rest of this section we illustrate how scopes-as-frames support a wide range of name binding phenomena, including functions, let bindings, blocks with local variables, and records; all in a systematic way that follows the same pattern.

Frames and heaps for lexical scoping. Returning to the factorial function example from earlier, the rightmost illustration in Figure 2 shows the evolution of the heap as the example

program executes. In this example, all the frames contain lexically-scoped variables, and correspond to stack frames in conventional language implementations. We examine the execution steps of the program to illustrate how frames and heaps are used to structure memory during run time, and how these correspond to the static scope graph.

- The program frame for scope `s1` is created at point (0); and the function closure is assigned to the `fac` slot. This is the initial current frame.
- The execution of the body of the `letrec` proceeds to initialize a new frame at point (1), which has the same structure as the scope (`s2`) that it instantiates: it is linked to a lexical parent frame that instantiates `s1`; and it has a single slot for `n`, the sole declaration of scope `s2`, which is initialized to contain `2`.
- At point (2), the newly created frame becomes the current frame. Then the statically computed resolution path for the `n` reference (colored blue in Figure 2) can be applied, starting at the current frame, to fetch the value `n` = `2`, whence a recursive call to `fac` is made in the `else` branch of the example program. Again, `fac` is dereferenced using the statically computed resolution path (highlighted in pink in Figure 2).
- The recursive call at point (3) creates another frame, and results in another recursive call and frame at point (4). At this point, `n` evaluates to `0`, and the recursive calls return at point (5).

At each step of execution, references are dereferenced relative to the current frame. The run-time memory access path for each reference is the same as its static resolution path.

Records and field access. Just as scopes provide a static model for many kinds of binding, frames support a wide range of patterns for binding at run time (not just stack-like binding). In particular, frames can be used to model structured memory objects, such as the records in our second example. These frames are allocated by an explicit **`new`** operator in the language.

The rightmost illustration in Figure 3 shows the evolution of the heap during program evaluation. Step-by-step:

- The program frame is created at point (0). This frame has a single slot for `A`, matching the declaration in scope `s1`.
- At point (1), the frame for the local variables `x` and `y` is created.
- Points (2) and (3) populate the slots for `x` and `y` with record values which point to freshly allocated record frames (indicated by the dotted lines at points (2) and (3) in Figure 3). These record frames instantiate the scope (`s2`) associated to the declaration of record type `A`.
- Next, the field slots of the record frames are filled in. The slot names are declared in record scope `s2`, but references to them are placed in intermediate scopes `s4` and `s5`, which import `s2`. To maintain a consistent relationship between static and run-time paths, we create an intermediate frame corresponding to `s4` at point (4). Then at point (5) we use the static path for `n` to locate the relevant slot in the record frame for `x` and set the value of `n` to `1`. Points (6) and (7) proceed similarly to (4) and (5).

Once more, the static and run-time resolution paths coincide, as illustrated by the colored paths in the scope graph and heap frame in Figure 3. Note that maintaining this correspondence leads to some infelicities in the dynamic frame layout. The slot created for `A` in the program frame is actually unused and could have been omitted; declarations of purely static things, like the type name `A`, do not need a run-time correlative. Similarly, the frames corresponding to the declaration-free scopes `s4` and `s5` are empty, and having to traverse

them to reach the record frame is inefficient. We could avoid both of these problems in a real system by using a more nuanced definition of correspondence between static and run-time paths; in this paper, we have elected to keep the scopes-as-frames correspondence as simple as possible.

Invariants. We have sketched how the layout of memory in frames follows the scope graph, and how this applies uniformly to a range of binding patterns. We can phrase the systematic correspondence between static scopes and dynamic frames as an invariant. Doing so gives a basis for statically checking and ruling out run-time errors. In a nutshell, the scopes and frames considered in this section all satisfy invariants that can be summarized as:

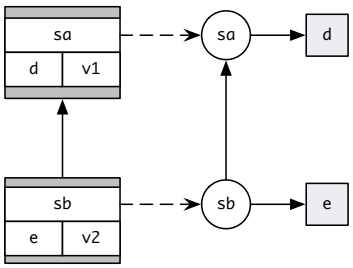

Figure 4 Frames instantiate scopes.

- *Binding invariant:* (a) the slots of a frame are in one-to-one correspondence with the declarations in the frame's scope; (b) the links of a frame are in one-to-one correspondence with the labeled edges of the frame's scope, and the scope of each link target matches the corresponding edge target. Figure 4 illustrates the resulting commuting diagram.
- *Typing invariant:* assuming declarations and values are typed, each slot value has the same type as its corresponding declaration.

These invariants extend to heaps by requiring that all frames in a heap satisfy the invariant. A consequence of these invariants is that each static resolution path can be used to perform corresponding run-time lookups, and (again assuming everything is typed) the values obtained from such lookups will be correctly typed.

In the next sections we formalize the framework and invariants outlined above. Concretely, we show how to define the static and dynamic semantics of a number of model languages with typical programming language binding patterns, and demonstrate and discuss how the scopes-as-frames invariant provides a basis for straightforward type soundness proofs and sound garbage collection.

3 Dynamic Frames for a Simple Functional Language

In this section we formalize our approach using L1, a language with arithmetic expressions and first-class, simply-typed functions, as a running example (Figure 8). We first present the language-independent framework of scope graphs to represent the (resolved) name binding and type facts of programs, and we formalize well-bound and well-typed L1 programs in terms of scope graphs. Then we introduce a language-independent framework of frames and heaps for modeling memory layout, and formalize the dynamic semantics of L1 in terms of it. Finally, we will see how phrasing the consistency between frames and scopes as an invariant gives a language-independent principle for proving type soundness, which we apply to prove the type soundness of L1.

3.1 Scope Graphs

We use scope graphs [14] extended with types [25] to provide a uniform (language-independent) treatment of name binding and type assignment in the definition of the static semantics of

Scope graph

$$
\begin{array}{rcl}
\mathit{ScopeId} \ni s & & \\
\mathit{Vertex} \ni v & ::= & s \mid x_i^{\mathsf{D}} \mid x_i^{\mathsf{R}} \\
\mathit{Edge} \ni e & ::= & s \xrightarrow{l} s \mid s \longrightarrow x_i^{\mathsf{D}} \\
& \mid & x_i^{\mathsf{R}} \longrightarrow s \mid x_i^{\mathsf{D}} \longrightarrow\!\triangleright s \\
\mathit{Label} \ni l & ::= & \mathbf{P} \mid \mathbf{I} \\
\mathit{TypeAnn} \ni ta & ::= & x_i^{\mathsf{D}} : t \\
\mathcal{G} \in \mathit{ScopeGraph} & \triangleq & \wp(\mathit{Vertex}) \times \wp(\mathit{Edge}) \times \\
& & \wp(\mathit{TypeAnn})
\end{array}
$$

Projection functions

$$
\begin{array}{l}
\mathcal{K}(s) = \{l \mapsto \{s' \mid s \xrightarrow{l} s'\}\} \\
\mathcal{D}(s) = \{x_i^{\mathsf{D}} \mid s \longrightarrow x_i^{\mathsf{D}}\} \\
\mathcal{R}(s) = \{x_i^{\mathsf{R}} \mid x_i^{\mathsf{R}} \longrightarrow s\}
\end{array}
$$

Figure 5 Scope graphs

Resolution paths

$$\mathit{Path} \ni p ::= \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{E}(l, s) \cdot p$$

Path consistency

$$\frac{\vdash_{\mathcal{G}} s \longrightarrow x_i^{\mathsf{D}}}{\vdash_{\mathcal{G}} \mathbf{D}(x_i^{\mathsf{D}}) : s \overset{\mathsf{S}}{\longmapsto} (s, x_i^{\mathsf{D}})} \quad [\mathsf{ResD}]$$

$$\frac{\vdash_{\mathcal{G}} s \xrightarrow{l} s' \quad \vdash_{\mathcal{G}} p : s' \overset{\mathsf{S}}{\longmapsto} (s'', x_i^{\mathsf{D}})}{\vdash_{\mathcal{G}} \mathbf{E}(l, s') \cdot p : s \overset{\mathsf{S}}{\longmapsto} (s'', x_i^{\mathsf{D}})} \quad [\mathsf{ResE}]$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s \quad \vdash_{\mathcal{G}} p : s \overset{\mathsf{S}}{\longmapsto} (s', x_j^{\mathsf{D}})}{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathsf{R}}{\longmapsto} (s', x_j^{\mathsf{D}})} \quad [\mathsf{ResR}]$$

Figure 6 Resolution paths

programming languages. We introduced scope graphs by example, including their graphical notation, in the previous section. Here we formalize scope graphs and resolution.

Vertices and edges. Figure 5 formally defines what we mean by a *scope graph*, consisting of vertices and edges. A scope graph vertex is either a scope (s), a declaration (x_i^{D}), or a reference (x_i^{R}). We assume that each scope identifier, declaration, and reference is *unique* in the graph. For a declaration x_i^{D} there is exactly one scope s which contains the declaration, represented as an edge $s \longrightarrow x_i^{\mathsf{D}}$ in the scope graph; similarly, for a reference x_i^{R} there is exactly one scope s which contains the reference, represented as an edge $x_i^{\mathsf{R}} \longrightarrow s$.

In addition to edges relating scope identifiers to references or declarations, there are labeled edges connecting two scopes. A labeled edge $s \xrightarrow{\mathbf{P}} s'$ means that s has s' as its *lexical parent*, and an edge $s \xrightarrow{\mathbf{I}} s'$ means that s *imports* s'. For language L1, we only need lexical scoping, but in Section 4 we consider a language with records which relies on imports.

The bottom of Figure 5 defines some useful projection functions for a given scope graph: $\mathcal{K}(s)$ is a map from labels l to the set of scopes to which s is connected via label l; $\mathcal{D}(s)$ is the set of all declarations contained in a scope; and $\mathcal{R}(s)$ is the set of all references contained in a scope.

The last kind of edge defined in Figure 5 is an *associated scope* edge: $x_i^{\mathsf{D}} \longrightarrow\!\triangleright s$ connects the declaration x_i^{D} of a named collection of names (e.g., a module or a record) to the scope s which declares the constituent names (e.g., the body of a module or a record). Again, language L1 does not rely on associated scopes, but the language in Section 4 does.

Finally, in a *typed* scope graph, declarations are annotated with a type ($x_i^{\mathsf{D}} : t$).

Resolution paths. Figure 6 defines *resolution paths*. We use the $\vdash_{\mathcal{G}}$ turnstile for predicates and relations that query an underlying scope graph $\mathcal{G}$. A path p is interpreted relative to an initial scope, and is given by a sequence of resolution steps. The resolution step $\mathbf{D}(x_i^{\mathsf{D}})$ checks that the declaration x_i^{D} is available in the initial scope, while $\mathbf{E}(l, s)$ checks that resolution continues by following the edge labeled by l from the initial scope to scope s. The path consistency relation $\vdash_{\mathcal{G}} p : s \overset{\mathsf{S}}{\longmapsto} (s', x_i^{\mathsf{D}})$ checks that a path p is consistent with the underlying scope graph, i.e. that there is an actual path in the graph from scope s to declaration x_i^{D} in scope s'. The relation $\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathsf{R}}{\longmapsto} (s', x_j^{\mathsf{D}})$ checks that there is a path p from the initial scope that contains x_i^{R} to a scope s' containing x_j^{D}.

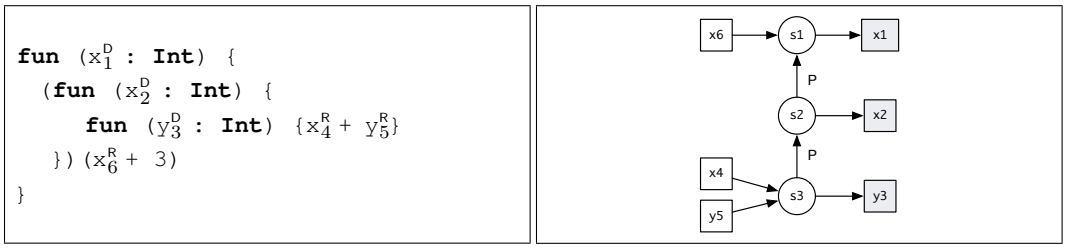

Figure 7 Example program with nested functions and its scope graph

In this paper, we assume that scope graphs have been resolved, following the resolution calculus of [14, 25], so that each reference is associated with a *unique and consistent* path to a declaration. In other words, a *resolved* scope graph includes a mapping from each reference x_i^{R} to a path p which corresponds to an actual resolution, i.e. there exist s' and x_j^{D} such that $\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathsf{R}}{\longmapsto} (s', x_j^{\mathsf{D}})$. When there are multiple possible resolution paths for a reference, the resolution calculus uses language-specific well-formedness and specificity ordering rules to choose which path to prefer. Since we are assuming resolution has already occurred, we ignore the details of these rules in this paper.

An example scope graph. Figure 7 defines the scope graph for an L1 expression which nests three functions (where we have replaced identifiers with unique references and declarations). Here, the $\xrightarrow{\mathbf{P}}$ edges connect the inner scope of each function to its enclosing (lexical) scope. For example, the reference $\mathtt{x}_6^{\mathsf{R}}$ has a trivial path in the scope graph. We write $p_6 = \mathbf{D}(\mathtt{x}_1^{\mathsf{D}}) : \mathtt{x}_6^{\mathsf{R}} \longmapsto \mathtt{x}_1^{\mathsf{D}}$ for this resolution path, which says that the declaration $\mathtt{x}_1^{\mathsf{D}}$ is found in the same scope as the reference $\mathtt{x}_6^{\mathsf{R}}$. Similarly, the resolution path for $\mathtt{y}_5^{\mathsf{R}}$ is $p_5 = \mathbf{D}(\mathtt{y}_3^{\mathsf{D}}) : \mathtt{y}_5^{\mathsf{R}} \longmapsto \mathtt{y}_3^{\mathsf{D}}$. In contrast, there are two possible paths from reference $\mathtt{x}_4^{\mathsf{R}}$ to matching declarations: $p_4 = \mathbf{E}(\mathbf{P}, s_2) \cdot \mathbf{D}(\mathtt{x}_2^{\mathsf{D}})$ and $p_4' = \mathbf{E}(\mathbf{P}, s_2) \cdot \mathbf{E}(\mathbf{P}, s_1) \cdot \mathbf{D}(\mathtt{x}_2^{\mathsf{D}})$. In this case, the shorter path p_4 would be preferred over the longer path p_4' according to L1's path specificity ordering rules, which encode lexical scoping (shadowing). Thus, the complete resolution mapping for this graph is $\{\mathtt{x}_6^{\mathsf{R}} \mapsto p_6, \mathtt{y}_5^{\mathsf{R}} \mapsto p_5, \mathtt{x}_4^{\mathsf{R}} \mapsto p_4\}$.

3.2 Well-Bound and Well-Typed Terms

Scope graphs provide a language-independent data structure for representing name binding and typing facts about the declarations and references in a program. To reason about the relation of this model to the program that it represents, we define a well-boundness and a well-typedness predicate on abstract syntax trees annotated with scope and type information. Figure 9 defines annotated terms for L1, ranged over by a, where expressions are annotated by their scope s and their type t. Identifiers have been replaced with unique references and declarations. We define well-boundness and well-typedness as separate predicates that check the consistency of (the annotations of) a term with respect to its scope graph. In definitions of these predicates, we omit the annotation that is not relevant for the property checked by the rule; that is, we leave out the type annotation in the well-boundness rules, and the scope annotation in the well-typedness rules. All rules are implicitly parameterized by a common underlying scope graph $\mathcal{G}$; all occurrences of $\vdash_{\mathcal{G}}$ query this $\mathcal{G}$.

The *well-boundness* rules in Figure 10 check that a term is *well-bound* relative to a resolved scope graph $\mathcal{G}$. That is, the scope graph is consistent with the scoping patterns of the term, and each reference is bound, i.e., resolves to a declaration with a path that is consistent with the scope graph. Rules [WbArithOp, WbApp] check that the child terms

$$
\begin{array}{l}
t ::= \texttt{Int} \mid t \to t \\
e ::= z \mid x \mid e \oplus e \mid \texttt{fun}(x\ \texttt{:}\ t)\{\ e\ \} \mid e(e) \\
z \in \mathbb{Z}
\end{array}
$$

Figure 8 Syntax of L1

$$
\begin{array}{l}
t ::= \texttt{Int} \mid t \to t \\
e ::= z \mid x_i^{\mathsf{R}} \mid a \oplus a \mid \texttt{fun}(\,x_i^{\mathsf{D}}\ \texttt{:}\ t)\{\ a\ \} \mid a(a) \\
a ::= e^{s,t}
\end{array}
$$

Figure 9 Annotated syntax of L1

Well-boundness $\boxed{\vdash^{\mathsf{B}} a}$

$$\vdash^{\mathsf{B}} z^{s} \quad [\mathsf{WbInt}]$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s}{\vdash^{\mathsf{B}} (x_i^{\mathsf{R}})^{s}} \quad [\mathsf{WbRef}]$$

$$\frac{\vdash^{\mathsf{B}} e_1^{s} \quad \vdash^{\mathsf{B}} e_2^{s}}{\vdash^{\mathsf{B}} (e_1^{s} \oplus e_2^{s})^{s}} \quad [\mathsf{WbArithOp}]$$

$$\frac{\vdash^{\mathsf{B}} e^{s_1} \quad \vdash_{\mathcal{G}} s_1 \xrightarrow{\mathbf{P}} s_2 \quad \vdash_{\mathcal{G}} \mathcal{D}(s_1) = \{x_i^{\mathsf{D}}\}}{\vdash^{\mathsf{B}} (\texttt{fun}\ (x_i^{\mathsf{D}}\ \texttt{:}\ t)\ \{\ e^{s_1}\ \})^{s_2}} \quad [\mathsf{WbFun}]$$

$$\frac{\vdash^{\mathsf{B}} e_1^{s} \quad \vdash^{\mathsf{B}} e_2^{s}}{\vdash^{\mathsf{B}} (e_1^{s}(e_2^{s}))^{s}} \quad [\mathsf{WbApp}]$$

Figure 10 Well-boundness of L1 terms

Well-typedness $\boxed{\vdash^{\mathsf{T}} a}$

$$\vdash^{\mathsf{T}} z^{\texttt{Int}} \quad [\mathsf{WtInt}]$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \xmapsto{\mathsf{R}} (s', x_j^{\mathsf{D}}) \quad \vdash_{\mathcal{G}} x_j^{\mathsf{D}} : t}{\vdash^{\mathsf{T}} (x_i^{\mathsf{R}})^{t}} \quad [\mathsf{WtRef}]$$

$$\frac{\vdash^{\mathsf{T}} e_1^{\texttt{Int}} \quad \vdash^{\mathsf{T}} e_2^{\texttt{Int}}}{\vdash^{\mathsf{T}} \left(e_1^{\texttt{Int}} \oplus e_2^{\texttt{Int}}\right)^{\texttt{Int}}} \quad [\mathsf{WtArithOp}]$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathsf{D}} : t_1 \quad \vdash^{\mathsf{T}} e^{t_2}}{\vdash^{\mathsf{T}} (\texttt{fun}(x_i^{\mathsf{D}}\ \texttt{:}\ t_1)\{\ e^{t_2}\ \})^{t_1 \to t_2}} \quad [\mathsf{WtFun}]$$

$$\frac{\vdash^{\mathsf{T}} e_1^{t_1 \to t_2} \quad \vdash^{\mathsf{T}} e_2^{t_1}}{\vdash^{\mathsf{T}} \left(e_1^{t_1 \to t_2}(e_2^{t_1})\right)^{t_2}} \quad [\mathsf{WtApp}]$$

Figure 11 Well-typedness of L1 terms

of arithmetic operator applications and function applications reside in the same scope as the parent term. Rule [WbRef] checks that the reference x_i^{R} is declared in the scope s of its annotation ($\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s$). Rule [WbFun] checks that the scope s_1 of the body of a function is a lexical child of the scope s_2 of the function expression, and that the set of declarations of the function scope is exactly the singleton set containing the formal parameter of the function.

The *well-typedness* predicate on annotated terms checks that (1) expressions are consistently typed according to the rules of the language, and (2) that each reference is typed consistently with its declaration. Figure 11 specifies well-typedness rules for L1. The Rules [WtInt, WtArithOp, WtFun, WtApp] check that the type of an expression is consistent with the types of its direct sub-expressions. Note the absence of environment threading in the well-typedness rules; scope management is taken care of by the well-boundness rules.

3.3 Frames and Heaps

We now develop the formalization of the language-independent system of heaps and frames to represent memory at run-time. Figure 12 defines frames, heaps, and some operations on them that we will use in the dynamic semantics of L1. The framework is parameterized with a domain *Val* of values.

A heap h is a finite map from frame identifiers (pointers) to frames. A frame consists of a scope identifier, a collection of dynamic links, and a collection of slots. The scope identifier refers to the scope that the frame instantiates. The dynamic links are defined as a two-dimensional mapping from labels and scopes to frame identifiers. These links are the dynamic counterparts of labeled edges in the scope graph and implement the link from a

Frames and heaps

$$f \in \mathit{FrameId} = \{f_1, f_2, \ldots\}$$
$$ks \in \mathit{DynLinks} = \mathit{Label} \xrightarrow{\text{fin}} \mathit{ScopeId} \xrightarrow{\text{fin}} \mathit{FrameId}$$
$$\sigma \in \mathit{Slots} = \mathit{Decl} \xrightarrow{\text{fin}} \mathit{Val}$$
$$\langle s, ks, \sigma \rangle \in \mathit{Frame} = \mathit{ScopeId} \times \mathit{DynLinks} \times \mathit{Slots}$$
$$h \in \mathit{Heap} = \mathit{FrameId} \xrightarrow{\text{fin}} \mathit{Frame}$$

Projection functions

$$\mathcal{S}_h(f) = s \quad \text{where } h(f) = \langle s, ks, \sigma \rangle$$
$$\mathcal{K}_h(f) = ks \text{ where } h(f) = \langle s, ks, \sigma \rangle$$
$$\mathcal{D}_h(f) = \sigma \quad \text{where } h(f) = \langle s, ks, \sigma \rangle$$

Operations on frames

$\boxed{\mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h'}$

$$\frac{f' \notin \mathit{Dom}(h)}{\mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h[f' \mapsto \langle s, ks, \sigma \rangle]} \quad [\text{InitFrame}]$$

Dynamic address lookup

$\boxed{\mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_i^{\mathrm{D}})}$

$$\frac{x_i^{\mathrm{D}} \in \mathit{Dom}(\mathcal{D}_h(f))}{\mathsf{lookup}(f, h, \mathbf{D}(x_i^{\mathrm{D}})) \Rightarrow \mathsf{Addr}(f, x_i^{\mathrm{D}})} \quad [\text{DLookupD}]$$

$$\frac{\mathcal{K}_h(f)(l)(s) = f' \quad \mathsf{lookup}(f', h, p) \Rightarrow \mathsf{Addr}(f'', x_i^{\mathrm{D}})}{\mathsf{lookup}(f, h, \mathbf{E}(l, s) \cdot p) \Rightarrow \mathsf{Addr}(f'', x_i^{\mathrm{D}})} \quad [\text{DLookupE}]$$

Fetching and mutating slot values

$\boxed{\mathsf{get}(f, h, x_i^{\mathrm{D}}) \Rightarrow v}$ $\boxed{\mathsf{set}(f, h, x_i^{\mathrm{D}}, v) \Rightarrow h'}$

$$\frac{\mathcal{D}_h(f) = \sigma \quad \sigma(x_i^{\mathrm{D}}) = v}{\mathsf{get}(f, h, x_i^{\mathrm{D}}) \Rightarrow v} \quad [\text{GetSlot}]$$

$$\frac{x_i^{\mathrm{D}} \in \mathit{Dom}(\mathcal{D}_h(f))}{\mathsf{set}(f, h, x_i^{\mathrm{D}}, v) \Rightarrow h[f \mapsto (\mathcal{D}_h(f)[x_i^{\mathrm{D}} \mapsto v])]} \quad [\text{SetSlot}]$$

Figure 12 A language-independent formalization of frames and heaps

Values

$$\mathit{Val} \ni v ::= \mathsf{NumV}(z) \mid \mathsf{ClosV}(x_i^{\mathrm{D}}, a, f)$$

Annotation projections

$$\mathcal{S}(e^{s,t}) = s \qquad \mathcal{T}(e^{s,t}) = t$$

Dynamic semantics

$\boxed{f \vdash a/h \Rightarrow v/h'}$

$$f \vdash z/h \Rightarrow \mathsf{NumV}(z)/h \quad [\text{EvInt}]$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \xrightarrow{\mathrm{R}} (s', x_j^{\mathrm{D}}) \quad \mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_j^{\mathrm{D}}) \quad \mathsf{get}(f', h, x_j^{\mathrm{D}}) \Rightarrow v}{f \vdash x_i^{\mathrm{R}}/h \Rightarrow v/h} \quad [\text{EvRef}]$$

$$\frac{f \vdash a_1/h_1 \Rightarrow \mathsf{NumV}(z_1)/h_2 \quad f \vdash a_2/h_2 \Rightarrow \mathsf{NumV}(z_2)/h_3}{f \vdash a_1 \oplus a_2/h_1 \Rightarrow \mathsf{NumV}(\oplus(z_1, z_2))/h_3} \quad [\text{EvArithOp}]$$

$$f \vdash \texttt{fun}(x_i^{\mathrm{D}} : t)\{\, a \,\}/h \Rightarrow \mathsf{ClosV}(x_i^{\mathrm{D}}, a, f)/h \quad [\text{EvFun}]$$

$$\frac{\begin{array}{l} f \vdash a_1/h_1 \Rightarrow \mathsf{ClosV}(x_i^{\mathrm{D}}, a', f')/h_2 \quad f \vdash a_2/h_2 \Rightarrow v/h_3 \\ \mathsf{initFrame}(\mathcal{S}(a'), \{\mathbf{P} \mapsto \{\mathcal{S}_{h_3}(f') \mapsto f'\}\}, \{x_i^{\mathrm{D}} \mapsto v\})/h_3 \Rightarrow f''/h_4 \\ f'' \vdash a'/h_4 \Rightarrow v'/h_5 \end{array}}{f \vdash a_1(a_2)/h_1 \Rightarrow v'/h_5} \quad [\text{EvApp}]$$

Figure 13 Dynamic semantics of L1

frame to the frame that represents its lexical context or an imported module. The slots of a frame are defined as a finite map from declarations to values.

We define operations on heaps to construct new frames and retrieve slot values. Note that all operations will refer to frames using their frame identifier (f). Accessing frames is done via a lookup in the heap ($h(f)$) and using the projection functions $\mathcal{S}\_(\_)$, $\mathcal{K}\_(\_)$, or $\mathcal{D}\_(\_)$. We will write 'frame f' as shorthand for 'the frame referred to by frame identifier f'. The $\mathsf{initFrame}(s, ks, \sigma)$ operation adds a new frame with scope identifier s, links ks, and slots σ, to the heap at a fresh frame identifier. The operation is defined as a relation $\mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h'$, using the notation t/h to represent the pair of a term (or value) t and a heap h. We use this notation throughout to explicitly represent heap threading. (We could use a state monad to implicitly thread the heap, but we prefer to make semantic rules explicit for this presentation.)

The lookup operation $\mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_i^{\mathsf{D}})$ dereferences a path p relative to frame f, resulting in an *address*, consisting of a pair of a frame f' and a slot x_i^{D} in that frame. The operation $\mathsf{get}(f, h, x_i^{\mathsf{D}}) \Rightarrow v$ retrieves the value v of slot x_i^{D} from frame f; and $\mathsf{set}(f, h, x_i^{\mathsf{D}}, v) \Rightarrow h'$ sets the value of slot x_i^{D} of frame f to v, resulting in a new heap h'.

3.4 Dynamic Semantics

Figure 13 specifies the big-step dynamic semantics for L1 using a heap to represent the memory during execution. The value domain *Val* consists of numbers $\mathsf{NumV}(z)$ and closures $\mathsf{ClosV}(x_i^{\mathsf{D}}, a, f)$. The judgment $f \vdash a/h \Rightarrow v/h'$ specifies that evaluating an annotated expression a in the context of frame f in heap h gives a value v and heap h'. We discuss the rules and use the example in Figure 14 as illustration:

- The Rules [EvInt,EvArithOp] for arithmetic are standard and do not affect memory other than passing the current frame to evaluation of arguments and threading the heap.
- Rule [EvFun] constructs a *closure* which records the formal parameter, the body of the function, and the lexical context at the point of function definition, which is exactly represented by the current frame f. In Figure 14, the result of evaluating the function in the body of the function at (3) leads to a closure with frame `s3` as lexical context.
- Rule [EvApp] defines the evaluation of a function application. The function argument a_1 should evaluate to a closure. Using initFrame a new frame instantiating the scope of the function body ($\mathcal{S}(a')$) is constructed to represent the call frame of the function call. The frame from the closure (f') is used as the lexical parent of the call frame. The argument value (v) is assigned to the slot for the formal parameter of the function (x_i^{D}). The body of the function (a') is evaluated in the context of the new frame. In Figure 14, in the application `f(1)`, `f` evaluates to a closure with a pointer to the `s3` frame. The call frame for the body then constructs a frame for `s4` with the `s3` frame as its parent.
- Rule [EvRef] defines the evaluation of a variable x_i^{R}. The frame f represents the lexical context of the variable and the static scope graph path of the (reference representing the) variable is the offset into that context. The lookup happens in multiple steps: first, we find the path associated with x_i^{R} in the resolved scope graph; second, we dynamically follow the path to compute the dynamic memory address for the reference; lastly, we get the value of the dynamic memory address. In Figure 14, the evaluation at (5) of variable `x` in the `s4` frame follows the path defined for `x` in the scope graph. (Note how the call-time context of `f` is not accessible to the evaluation of the closure in frame `s4`.)

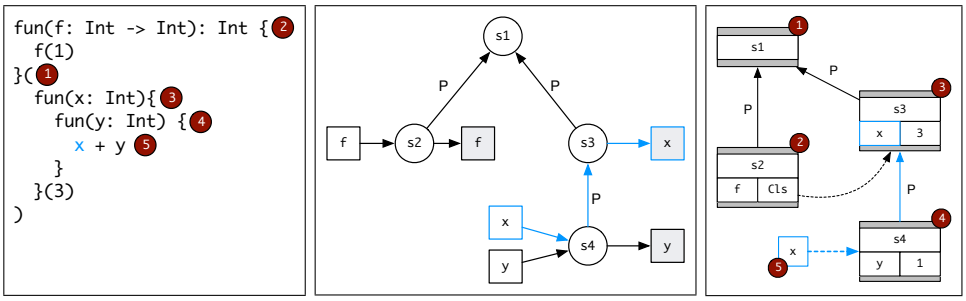

Figure 14 Example L1 program with scope graph and heap.

3.5 Intermezzo

What have we learned from this exercise so far? (1) *There is a systematic correspondence between static name binding and binding at run-time.* Static scopes are collections of declarations. Dynamic frames are units of memory allocation, holding values for the declarations in a scope. Static name binding is about visibility of declarations following a path from reference through the scope graph. Binding at run-time is about paths from evaluation frame to storage frame. This correspondence provides a guiding principle for the definition of the dynamic semantics. Evaluating a binding construct requires creation of a frame that instantiates the corresponding static scope. Evaluating a reference requires looking up its value in the heap using its static path as offset. (2) *We can mostly separate the specification of binding and typing rules.* Binding rules are concerned with describing the structure of and the access to memory. Typing rules are concerned with the types of the arguments and results of operations. In the dynamic semantics this corresponds to factoring out (and making systematic and uniform) the treatment of memory from other 'domain-specific' operations (such as arithmetic). We have seen that typing depends on binding in order to establish the types of references. In the next section we will see that binding depends on typing exactly when types are used to describe memory in data type definitions.

In conclusion: defining name binding in terms of scope graphs provides a uniform methodology (an API if you want) for the treatment of memory in static and dynamic semantics. This scales to a range of language features: in Section 4 we show that the approach scales to imperative features and records, and in Section 4.4 we discuss how to deal with classes and subtyping.

What other benefits does the approach provide? The design of this uniform memory model is just the start of a study of making language-independent those aspects of programming language design that are usually treated as language-specific. In Section 3.6 we will see how phrasing the consistency between frames and scopes as an invariant gives a *language-independent principle* for proving *type soundness*, i.e. that well-bound and well-typed programs cannot go wrong. Later, in Section 5 we discuss a high-level *language-independent* specification of *sound reachability-based garbage collection* of frames, and verify that several standard GC strategies refine this specification.

3.6 Type Soundness

We have argued that the structure of frames follows the structure of scopes. Now we make this argument precise by defining the correspondence between frames and scopes, and discuss how that supports a language-independent formulation and systematic proof of type soundness.

Good frame

For any definition of types t, values v, and value typing relation $\overset{\mathrm{V}}{\vdash}$:

$\mathsf{wellBound}(h, f) \triangleq$
$\quad \exists s.\ s = \mathcal{S}_h(f)\ \wedge\ \mathcal{D}(s) = Dom(\mathcal{D}_h(f)) \wedge$
$\quad\quad (\forall l\ s'.\ s' \in \mathcal{K}(s)(l)\ \iff\ \mathcal{S}_h(\mathcal{K}_h(f)(l)(s')) = s')$

$\mathsf{wellTyped}(h, f) \triangleq$
$\quad \forall t\ x_i^{\mathrm{D}} \in \mathcal{D}(\mathcal{S}_h(f)).\ \vdash_{\mathcal{G}} x_i^{\mathrm{D}} : t\ \implies\ \forall v.\ \mathcal{D}_h(f)(x_i^{\mathrm{D}}) = v\ \implies\ h \overset{\mathrm{V}}{\vdash} v : t$

$\mathsf{goodFrame}(h, f) \triangleq$
$\quad \mathsf{wellBound}(h, f)\ \wedge\ \mathsf{wellTyped}(h, f)$

Good heap

$\mathsf{goodHeap}(h) \triangleq (f \in Dom(h)\ \implies\ \mathsf{goodFrame}(h, f))$

Value typing $\boxed{h \overset{\mathrm{V}}{\vdash} v : t}$

$$h \overset{\mathrm{V}}{\vdash} \mathsf{NumV}(z) : \texttt{Int} \qquad [\mathsf{VtInt}]$$

$$\frac{\overset{\mathrm{B}}{\vdash} \texttt{fun}(x_i^{\mathrm{D}} : t_1)\{\ a\ \}^{\mathcal{S}_h(f)} \quad \overset{\mathrm{T}}{\vdash} \texttt{fun}(x_i^{\mathrm{D}} : t_1)\{\ a\ \}^{t_1 \to t_2}}{h \overset{\mathrm{V}}{\vdash} \mathsf{ClosV}(x_i^{\mathrm{D}}, a, f) : t_1 \to t_2} \qquad [\mathsf{VtClo}]$$

Figure 15 Good frames, good heaps, and value typing

Good frames. The goodFrame property defined in Figure 15 formally captures the correspondence between frames and scopes. The definition relies on two auxiliary properties:

1. wellBound: Maintains the binding invariant that frames have the same structure as their scopes. Specifically, the slots $\mathcal{D}_h(f)$ of each frame f are in one-to-one correspondence with the declarations of scope $s = \mathcal{S}_h(f)$, and the links $\mathcal{K}_h(f)$ of f are in one-to-one correspondence with the edges $\{l \mapsto s'\} \in \mathcal{K}(s)$, such that $\mathcal{K}_h(f)(l)(s') = f'$ iff $\mathcal{S}_h(f') = s'$.
2. wellTyped: Maintains the typing invariant that slots contain values of the type that their declarations expect. Specifically, for each declaration and its corresponding slot (if it exists) in the frame, the value in the slot is of the same type as the declaration.

These properties are language-independent, i.e. define a generic correspondence between scopes and frames, but are parameterized by a language-specific value typing judgment $h \overset{\mathrm{V}}{\vdash} v : t$ (also defined in Figure 15) which asserts that the value v has type t in the heap h. The value typing of closures (Rule [VtClo]) checks that the closure corresponds to an actual well-typed and well-bound function relative to the scope of the lexical frame recorded in the closure, using the rules from Figure 10 and Figure 11.

Good heaps. The goodHeap property also defined in Figure 15 generalizes the goodFrame property to the entire heap; that is, a heap h is good iff every frame in h satisfies the goodFrame property.

Language-independent lemmas. The scopes-as-frames approach provides structure to type soundness proofs by using a small suite of language-independent helper lemmas about dynamic memory operations, such as looking up a path, initializing a new frame, etc. For example, Lemma 1 formalizes the property that each static resolution path has a corresponding run-time memory access path.

▶ Lemma 1 (Good paths). Suppose $\mathsf{goodHeap}(h)$ and $\mathcal{S}_h(f) = s$. For any reference x_i^{R} where $\vdash_{\mathcal{G}} x_i^{\mathrm{R}} \longrightarrow s$, it holds for any p, s', and x_i^{D} that $\vdash_{\mathcal{G}} p : x_i^{\mathrm{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathrm{D}})$ implies that there exists a frame f' such that $\mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_j^{\mathrm{D}})$ and $\mathcal{S}_h(f') = s'$.

This, along with a number of other lemmas, provides structure to the type preservation proof that we discuss next. We refer the reader to our Coq development for the full details.

Type preservation. Theorem 2 proves type preservation, i.e., that if evaluation of a well-bound and well-typed program in a frame that is in a good heap succeeds, then the resulting value is of the expected type, and the resulting heap is good.

▶ Theorem 2 (Type preservation for L1). Suppose $\mathsf{goodHeap}(h_1)$ and $\mathcal{S}_{h_1}(f) = s$, as well as $\overset{\mathrm{B}}{\vdash} e^s$ and $\overset{\mathrm{T}}{\vdash} e^t$ hold. Then, for any v and h_2, $f \vdash e^{s,t}/h_1 \Rightarrow v/h_2$ implies $\mathsf{goodHeap}(h_2)$ and $h_2 \overset{\mathrm{V}}{\vdash} v : t$.

There are several ways to extend type preservation proofs to type soundness proofs. A traditional approach [11] to proving type soundness using big-step semantics is to add explicit "wrong" rules to a big-step semantics anywhere evaluation can get stuck. This allows distinguishing between getting stuck and diverging, which is otherwise impossible using the inductively defined big-step rules in Figure 13. Our accompanying technical report summarizes the rules that need to be added to the language. The rules and full type soundness proof can also be found in our Coq development. Adding wrong rules to a language does not alter the structure of the cases for the preservation proof. Section 6 recalls and discusses alternative means of proving type soundness using big-step semantics.

4 Dynamic Frames for Records

In this section we apply our approach to L2, an extension of L1 with records and imperative features. This shows that frames-as-scopes scales to model the memory layout for records, and that the invariants about frames and heaps given in the previous section also provide a basis for type soundness proofs for languages with such features.

Figure 16 defines the well-formed terms of L2 with changes from L1 highlighted. A program in L2 consists of a sequence of *record type definitions* d, and a program expression e, enclosed in a **`begin` ... `end`** block. A record type definition declares the type name and the record's field names. Records are constructed using the **`new`** construct, which takes a reference that resolves to a record type name declaration. L2 also adds an assignment operation (`_ := _`), which assigns a value to a memory address. Memory addresses (or *L-values* [23]) consist of frame-declaration pairs (f, x_i^{D}), and are computed by left-hand side (lhs) expressions, namely variables (x_i^{R}) and record field access $(a.x_i^{\mathrm{R}})$. Finally, L2 has a sequential composition operator (`_; _`).

4.1 Well-Bound and Well-Typed Terms

Figure 17 specifies the well-boundness rules for the newly introduced L2 terms. The well-boundness of L1 expressions from Figure 10 carry over to L2. Well-boundness checking in L2 uses three new kinds of judgments: $\overset{\mathrm{B}}{\underset{\mathrm{P}}{\vdash}}, \overset{\mathrm{T}}{\underset{\mathrm{P}}{\vdash}}$ for programs, $\overset{\mathrm{B}}{\underset{\mathrm{D}}{\vdash}}, \overset{\mathrm{T}}{\underset{\mathrm{D}}{\vdash}}$ for declarations, and $\overset{\mathrm{B}}{\underset{\mathrm{L}}{\vdash}}, \overset{\mathrm{T}}{\underset{\mathrm{L}}{\vdash}}$ for lhs expressions. The well-typedness of terms in L1 also carries over to L2. Figure 18 defines rules for checking the well-typedness of the new programs and terms in L2.

- Rule [WbProg] defines the well-boundness of programs and checks that the record definitions ds are well-bound, and that the expression e of a program is well-bound.

$$
\begin{array}{rcl}
t & ::= & \texttt{Int} \mid t \to t \mid \texttt{Rec}\,(x_i^{\mathsf{D}}) \\
p & ::= & d^*\ \texttt{begin}\ e\ \texttt{end} \\
d & ::= & \texttt{record}\ x_i^{\mathsf{D}}\ \{\ \ (x_i^{\mathsf{D}}\ \texttt{:}\ t)^*\ \} \\
e & ::= & z \mid \mathit{lhs} \mid a \oplus a \mid \texttt{fun}\,(x_i^{\mathsf{D}}\texttt{:}\ t)\{\ a\ \} \mid a(a) \mid \texttt{new}\ x_i^{\mathsf{R}} \mid \mathit{lhs}\ \texttt{:=}\ a \mid a\texttt{;}\ a \\
\mathit{lhs} & ::= & x_i^{\mathsf{R}} \mid a\,.\,x_i^{\mathsf{R}} \\
a & ::= & e^{s,t}
\end{array}
$$

Figure 16 Annotated syntax of L2 with distinguished references and declarations

Well-bound programs $\boxed{\vdash^{\mathrm{B}}_{\mathrm{P}} p}$

$$\frac{(\forall d \in ds.\ \vdash^{\mathrm{B}}_{\mathrm{D}} d^s) \quad \vdash^{\mathrm{B}} e^s}{\vdash^{\mathrm{B}}_{\mathrm{P}} ds\ \texttt{begin}\ e^s\ \texttt{end}}\ [\mathsf{WbProg}]$$

Well-bound declarations $\boxed{\vdash^{\mathrm{B}}_{\mathrm{D}} d^s}$

$$\frac{\vdash_{\mathcal{G}} s \longrightarrow x_i^{\mathsf{D}} \quad \vdash_{\mathcal{G}} x_i^{\mathsf{D}} \longrightarrow\!\triangleright s' \quad \forall (x_j^{\mathsf{D}}\ \texttt{:}\ t) \in \mathit{flds}.\ \vdash_{\mathcal{G}} s' \longrightarrow x_j^{\mathsf{D}}}{\vdash^{\mathrm{B}}_{\mathrm{D}} (\texttt{record}\ x_i^{\mathsf{D}}\ \{\ \mathit{flds}\ \})^s}\ [\mathsf{WbRD}]$$

Well-bound expressions $\boxed{\vdash^{\mathrm{B}} e^s}$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s \quad \vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathsf{D}}) \quad \vdash_{\mathcal{G}} x_j^{\mathsf{D}} \longrightarrow\!\triangleright s_{rec} \quad \vdash_{\mathcal{G}} \mathcal{K}(s_{rec}) = \emptyset}{\vdash^{\mathrm{B}} (\texttt{new}\ x_i^{\mathsf{R}})^s}\ [\mathsf{WbNew}]$$

$$\frac{\vdash^{\mathrm{B}} \mathit{lhs}^s \quad \vdash^{\mathrm{B}} e^s}{\vdash^{\mathrm{B}} (\mathit{lhs}\ \texttt{:=}\ e^s)^s}\ [\mathsf{WbAsgn}]$$

$$\frac{\vdash^{\mathrm{B}} e_1^s \quad \vdash^{\mathrm{B}} e_2^s}{\vdash^{\mathrm{B}} (e_1^s\texttt{;}\ e_2^s)^s}\ [\mathsf{WbSeq}]$$

$$\frac{\vdash^{\mathrm{B}}_{\mathrm{L}} \mathit{lhs}^s}{\vdash^{\mathrm{B}} \mathit{lhs}^s}\ [\mathsf{WbLhs}]$$

Well-bound lhs expressions $\boxed{\vdash^{\mathrm{B}}_{\mathrm{L}} \mathit{lhs}^s}$

$$\frac{\vdash^{\mathrm{B}} e^s \quad \vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s' \quad \vdash_{\mathcal{G}} s' \xrightarrow{\mathbf{I}} s_{rec} \quad \vdash_{\mathcal{G}} \mathcal{D}(s') = \emptyset}{\vdash^{\mathrm{B}}_{\mathrm{L}} (e^s\,.\,x_i^{\mathsf{R}})^s}\ [\mathsf{WbLFAcc}]$$

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s}{\vdash^{\mathrm{B}}_{\mathrm{L}} (x_i^{\mathsf{R}})^s}\ [\mathsf{WbLVar}]$$

Figure 17 Well-boundness in L2

Well-typed programs $\boxed{\vdash^{\mathrm{T}}_{\mathrm{P}} p}$

$$\frac{(\forall d \in ds.\ \vdash^{\mathrm{T}}_{\mathrm{D}} d) \quad \vdash^{\mathrm{T}} e^t}{\vdash^{\mathrm{T}}_{\mathrm{P}} ds\ \texttt{begin}\ e^t\ \texttt{end}}\ [\mathsf{WtProg}]$$

Well-typed declarations $\boxed{\vdash^{\mathrm{T}}_{\mathrm{D}} d^t}$

$$\frac{\forall (x_j^{\mathsf{D}}\ \texttt{:}\ t) \in \mathit{flds}.\ \vdash_{\mathcal{G}} x_j^{\mathsf{D}} : t}{\vdash^{\mathrm{T}}_{\mathrm{D}} \texttt{record}\ x_i^{\mathsf{D}}\ \{\ \mathit{flds}\ \}}\ [\mathsf{WtRD}]$$

Well-typed expressions $\boxed{\vdash^{\mathrm{T}} e^t}$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathsf{D}})}{\vdash^{\mathrm{T}} (\texttt{new}\ x_i^{\mathsf{R}})^{\texttt{Rec}\,(x_j^{\mathsf{D}})}}\ [\mathsf{WtNew}]$$

$$\frac{\vdash^{\mathrm{T}}_{\mathrm{L}} \mathit{lhs}^t \quad \vdash^{\mathrm{T}} e^t}{\vdash^{\mathrm{T}} (\mathit{lhs}\ \texttt{:=}\ e^t)^t}\ [\mathsf{WtAsgn}]$$

$$\frac{\vdash^{\mathrm{T}} e_1^{t_1} \quad \vdash^{\mathrm{T}} e_2^{t_2}}{\vdash^{\mathrm{T}} (e_1^{t_1}\texttt{;}\ e_2^{t_2})^{t_2}}\ [\mathsf{WtSeq}]$$

$$\frac{\vdash^{\mathrm{T}}_{\mathrm{L}} \mathit{lhs}^t}{\vdash^{\mathrm{T}} \mathit{lhs}^t}\ [\mathsf{WtLhs}]$$

Well-typed lhs expressions $\boxed{\vdash^{\mathrm{T}}_{\mathrm{L}} \mathit{lhs}^t}$

$$\frac{\vdash^{\mathrm{T}} e^{\texttt{Rec}\,(x_j^{\mathsf{D}})} \quad \vdash_{\mathcal{G}} x_j^{\mathsf{D}} \longrightarrow\!\triangleright s_{rec} \quad \vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s' \quad \vdash_{\mathcal{G}} s' \xrightarrow{\mathbf{I}} s_{rec} \quad \vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathrm{R}}{\longmapsto} (s'', x_i^{\mathsf{D}}) \quad \vdash_{\mathcal{G}} x_i^{\mathsf{D}} : t}{\vdash^{\mathrm{T}}_{\mathrm{L}} \left(e^{\texttt{Rec}\,(x_j^{\mathsf{D}})}\,.\,x_i^{\mathsf{R}}\right)^t}\ [\mathsf{WtLFAcc}]$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_i^{\mathsf{D}}) \quad \vdash_{\mathcal{G}} x_i^{\mathsf{D}} : t}{\vdash^{\mathrm{T}}_{\mathrm{L}} (x_i^{\mathsf{R}})^t}\ [\mathsf{WtLVar}]$$

Figure 18 Well-typedness in L2

- Rule [WbRD] in Figure 17 checks that the type name x_i^{D} is declared in the surrounding scope s, that it is associated with scope s', which has the surrounding scope s as its lexical parent, and that each field of the record is declared in scope s'.
- Rule [WbNew] checks that the record name resolves to a record declaration. Also, the associated record scope does not have any edges, a fact which is needed when initializing a matching record frame.
- Rules [WbAsgn] and [WtAsgn] simply check that both the lhs expression and the value expression are well-bound and well-typed. Rules [WbLhs,WtLhs] in turn rely on a separate judgment for checking that an lhs expression (either a field access or a variable) is well-bound and well-typed.
- Rules [WbLFAcc] and [WtLFAcc] describe the static semantics of field access. In a field access expression such as $a.x_i^{\mathsf{R}}$, the annotated expression a computes a record, and the reference x_i^{R} refers to a field declared in this record. The field reference is installed in an auxiliary scope s' that imports the scope s_{rec} associated with the record type $\mathbf{Rec}(x_j^{\mathsf{D}})$.

4.2 Dynamic Semantics

The dynamic semantics of most existing constructs from L1 is unchanged in L2. Figure 19 gives the dynamic semantics of the new constructs in the L2 extension and replaces the rule for variables since variables are now lhs expressions. We describe these changes.

Rule [EvNew] says that evaluating a **new** expression constructs a record frame via $\mathsf{initDefault}(s, ks)$. Here, s is a scope identifier and ks is the map of dynamic links for the frame. As specified in Figure 20, initDefault instantiates all slots of the frame with *default values*. The $\overset{\mathsf{D}}{\underset{\mathsf{V}}{\vdash}}$ judgment in Figure 20 defines default values for all types in L2. We introduce a special default function ($\mathsf{DFun}(v)$) which, when applied to anything, returns the value v. Rule [EvAppDef] specifies the semantics for applications involving this value.

We also introduce the null value, NullV, as the default value for record types. Trying to access a NullV value as a field results in a null-pointer exception being raised (Rule [LhsFAccNull]). Not shown in Figure 19 are the straightforward rules for propagating such exceptions, which can be found in the accompanying technical report [17]. We choose to let frames contain null values by default for several reasons: firstly, null values (or a variant thereof) are required if we want to construct self-referential records. Secondly, using default values and null values ensures that frames are always well-typed, which makes reasoning about type soundness easier.

The rules in Fig. 19 closely follow the static semantics for binding and typing. For example, Rule [LhsFAcc] initializes an import frame which is used as the basis for dynamic resolution. In order to evaluate lhs expressions, a new judgment $f \overset{\mathsf{LHS}}{\vdash} lhs/h \Rightarrow u/h'$ is introduced that evaluates a *lhs* to an address, or throws a null-pointer exception if the lhs expression attempts to access a field of a null value. Lhs expressions compute auxiliary values ranged over by u, as defined in Figure 19.

Rule [EvAsgn] uses $\mathsf{set}(f, h, x_i^{\mathsf{D}}, v) \Rightarrow h'$ for updating frame f in heap h by assigning v to slot x_i^{D} to produce a new frame h'. Setting a slot fails if the slot has not been allocated. Note that Rule [EvAsgn] defines assignment not just for record fields but also for variables, which makes variables bound by functions mutable. Our framework currently does not distinguish between mutable and immutable variables, but we expect it would be straightforward to introduce such a distinction, e.g. by classifying declarations into separate namespaces.

Values

$Val \ni v ::= \ldots \mid \mathsf{RecordV}(f) \mid \mathsf{NullV} \mid \mathsf{ExcV}(X) \mid \mathsf{DFun}(v)$

$Exc \ni X ::= \mathsf{NullPointer}$

Auxiliary values

$AuxVal \ni u ::= \mathsf{Addr}(f, x_i^{\mathsf{D}}) \mid \mathsf{ExcV}(X)$

Program evaluation $\boxed{\vdash^{\mathrm{P}} p \Rightarrow v/h'}$

$$\frac{\mathsf{initDefault}(s, \emptyset)/\emptyset \Rightarrow f_{root}/h \quad f_{root} \vdash e/h \Rightarrow v/h'}{\vdash^{\mathrm{P}} ds\ \mathtt{begin}\ e^s\ \mathtt{end} \Rightarrow v/h'}\ [\mathsf{EvProg}]$$

Expression evaluation $\boxed{f \vdash a/h \Rightarrow v/h'}$

$$\frac{f \vdash^{\mathrm{LHS}} lhs/h_1 \Rightarrow \mathsf{Addr}(f', x_i^{\mathsf{D}})/h_2 \quad \mathsf{get}(f', h_2, x_i^{\mathsf{D}}) \Rightarrow v}{f \vdash lhs/h_1 \Rightarrow v/h_2}\ [\mathsf{EvLhs}]$$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathsf{D}}) \quad \vdash_{\mathcal{G}} x_j^{\mathsf{D}} \longrightarrow\!\!\triangleright s \quad \mathsf{initDefault}(s, \emptyset)/h_1 \Rightarrow f'/h_2}{f \vdash \mathtt{new}\ x_i^{\mathsf{R}}/h_1 \Rightarrow \mathsf{RecordV}(f')/h_2}\ [\mathsf{EvNew}]$$

$$\frac{f \vdash^{\mathrm{LHS}} lhs/h_1 \Rightarrow \mathsf{Addr}(f', x_i^{\mathsf{D}})/h_2 \quad f \vdash e/h_2 \Rightarrow v/h_3 \quad \mathsf{set}(f', h_3, x_i^{\mathsf{D}}, v) \Rightarrow h_4}{f \vdash lhs\ \mathtt{:=}\ e/h_1 \Rightarrow v/h_4}\ [\mathsf{EvAsgn}]$$

$$\frac{f \vdash a_1/h_1 \Rightarrow \mathsf{DFun}(v_1)/h_2 \quad f \vdash a_2/h_2 \Rightarrow v_2/h_3}{f \vdash a_1(a_2)/h_1 \Rightarrow v_1/h_3}\ [\mathsf{EvAppDef}]$$

Lhs evaluation $\boxed{f \vdash^{\mathrm{LHS}} lhs/h \Rightarrow u/h'}$

$$\frac{\vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathsf{D}}) \quad \mathsf{lookup}(f, h, p) \Rightarrow \mathsf{Addr}(f', x_j^{\mathsf{D}})}{f \vdash^{\mathrm{LHS}} x_i^{\mathsf{R}}/h \Rightarrow \mathsf{Addr}(f', x_j^{\mathsf{D}})/h}\ [\mathsf{LhsVar}]$$

$$\frac{\begin{array}{l} f \vdash e/h_1 \Rightarrow \mathsf{RecordV}(f_{rec})/h_2 \quad \vdash_{\mathcal{G}} x_i^{\mathsf{R}} \longrightarrow s \\ \mathsf{initDefault}(s, \{\mathbf{I} \mapsto \{\mathcal{S}_{h_2}(f_{rec}) \mapsto f_{rec}\}\})/h_2 \Rightarrow f'/h_3 \\ \vdash_{\mathcal{G}} p : x_i^{\mathsf{R}} \overset{\mathrm{R}}{\longmapsto} (s', x_j^{\mathsf{D}}) \quad \mathsf{lookup}(f', h_3, p) \Rightarrow \mathsf{Addr}(f'', x_j^{\mathsf{D}}) \end{array}}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathsf{R}}/h_1 \Rightarrow \mathsf{Addr}(f'', x_j^{\mathsf{D}})/h_3}\ [\mathsf{LhsFAcc}]$$

$$\frac{f \vdash e/h_1 \Rightarrow \mathsf{NullV}/h_2}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathsf{R}}/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{NullPointer})/h_2}\ [\mathsf{LhsFAccNull}]$$

Figure 19 Dynamic semantics of L2

Frame operations $\boxed{\mathsf{initDefault}(s, ks)/h \Rightarrow f'/h'}$

$$\frac{\mathsf{defaults}(\mathcal{D}(s)) \Rightarrow \sigma \quad \mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h'}{\mathsf{initDefault}(s, ks)/h \Rightarrow f'/h'}\ [\mathsf{InitDefault}]$$

Default slots $\boxed{\mathsf{defaults}(ds) \Rightarrow \sigma}$

$$\mathsf{defaults}(\emptyset) \Rightarrow \emptyset\ [\mathsf{DNil}]$$

$$\frac{x_i^{\mathsf{D}} \in ds \quad \vdash_{\mathcal{G}} x_i^{\mathsf{D}} : t \quad \vdash^{\mathrm{DV}} v : t \quad \mathsf{defaults}(ds - \{x_i^{\mathsf{D}}\}) \Rightarrow \sigma}{\mathsf{defaults}(ds) \Rightarrow \sigma[x_i^{\mathsf{D}} \mapsto v]}\ [\mathsf{DCons}]$$

Default values $\boxed{\vdash^{\mathrm{DV}} t : v}$

$$\vdash^{\mathrm{DV}} 0 : \mathtt{Int}\ [\mathsf{DefaultInt}]$$

$$\frac{\vdash^{\mathrm{DV}} v : t_2}{\vdash^{\mathrm{DV}} \mathsf{DFun}(v) : t_1 \to t_2}\ [\mathsf{DefaultFun}]$$

$$\vdash^{\mathrm{DV}} \mathsf{NullV} : \mathtt{Rec}\,(x_i^{\mathsf{D}})\ [\mathsf{DefaultRec}]$$

Figure 20 Frame operations for updating slots and initializing frames with default values

$$\frac{\vdash_{\mathcal{G}} x_i^{\mathrm{D}} \longrightarrow\!\triangleright \mathcal{S}_h(f)}{h \overset{\mathrm{V}}{\vdash} \mathsf{RecordV}(f) : \mathbf{Rec}\,(x_i^{\mathrm{D}})} \quad [\mathsf{VtRec}]$$

$$\frac{h \overset{\mathrm{V}}{\vdash} v : t_2}{h \overset{\mathrm{V}}{\vdash} \mathsf{DFun}(v) : t_1 \to t_2} \quad [\mathsf{VtDefFun}]$$

$$h \overset{\mathrm{V}}{\vdash} \mathsf{NullV} : \mathbf{Rec}\,(x_i^{\mathrm{D}}) \quad [\mathsf{VtNullRec}]$$

$$h \overset{\mathrm{V}}{\vdash} \mathsf{ExcV}(\mathsf{NullPointer}) : t \quad [\mathsf{VtNullPointer}]$$

Figure 21 Value-typing of records and null-values

4.3 Type Soundness

The rules in Figure 21 summarize the value typing judgment for L2. Here, null-pointer exceptions are viewed as well-typed values of any kind, which allows such exceptions to occur anywhere in well-typed programs. Theorem 3 states type preservation for L2.

▶ Theorem 3 (Type preservation for L2). Suppose $\mathsf{goodHeap}(h_1)$ and $\mathcal{S}_{h_1}(f) = s$, as well as $\overset{\mathrm{B}}{\vdash} e^s$ and $\overset{\mathrm{T}}{\vdash} e^t$ hold. Then, for any v and h_2, $f \vdash e^{s,t}/h_1 \Rightarrow v/h_2$ implies $\mathsf{goodHeap}(h_2)$ and $h_2 \overset{\mathrm{V}}{\vdash} v : t$.

As argued in connection with the proof of Theorem 2 in Section 3.6, we can extend this proof to a proof of type soundness by adding cases for "going wrong". The wrong cases, which are given in the accompanying technical report [17] follow along the same lines as the cases in the preservation proof.

4.4 From Records to Classes

Our accompanying technical report [17] describes L3, an extension of L2 which replaces records by classes with inheritance, sub-typing, and overriding. Inheritance and sub-typing are modeled by representing run-time objects as a hierarchy of frames, one frame for each super-class in the hierarchy. Paths to object fields may contain edges that traverse this hierarchy. This poses interesting challenges for looking up fields at run time.

Figure 22 provides a small but illustrative example L3 program, its scope graph, and its object frame structure. Consider the expression: `(fun (a : A) { a.x })(new B)`. The path of the field access expression `a.x` is calculated based on `a` being an object of type `A`; i.e., $\mathbf{E}(\mathbf{I}, \mathtt{s2}) \cdot \mathbf{D}(\mathtt{x})$, where `s2` is the scope for the class `A`. But applying this path to the actual field of the value gets us to a subtype of `A`, namely the scope of the `B` object, which only provides an `x` slot via its import edge! The actual path to the desired `x` slot is thus: $\mathbf{E}(\mathbf{I}, \mathtt{s3}) \cdot \mathbf{E}(\mathbf{I}, \mathtt{s2}) \cdot \mathbf{D}(\mathtt{x})$. In order to deal with this mismatch between the statically computed path and the actual path, we *upcast* the `B` frame (instantiating scope `s3`) bound to `a` before attempting to access fields through it. Upcasting follows a chain of import edges until it arrives at a frame that instantiates the statically expected type, in this case the `A` frame (instantiating scope `s2`). The technical report [17] and Coq development provide the full details.

5 Garbage Collection

So far, we have described how the heap grows by adding frames. Any realistic language implementation also needs to consider how unused frames can be reclaimed. To this end, we

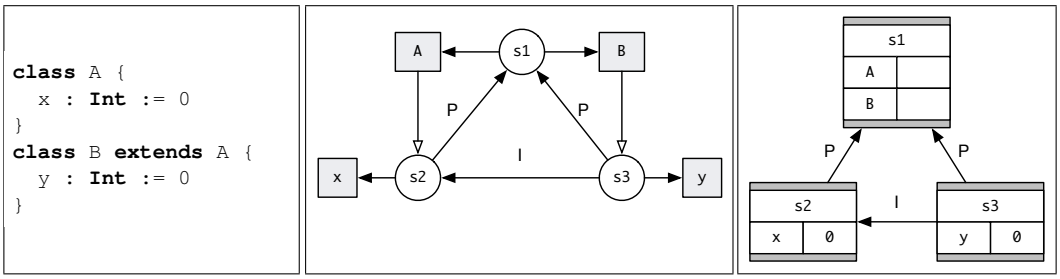

Figure 22 An example program, its scope graph, and its object frame structure

now give a high-level specification of sound reachability-based garbage collection of frames in our setting, and verify that several standard GC strategies refine this specification. We leave actual implementation of concrete collectors as future work.

We model garbage collection as removal of frames from the heap map; the removed frame identifiers then become fresh again for subsequent frame allocations. It is safe to remove a frame exactly when doing so would not change subsequent observable program behavior [12]. Since this is an undecidable criterion, collectors instead use an approximation based on whether there are live pointers to the frame; if not, the frame certainly cannot be accessed again (assuming pointers are unforgeable) and hence may be safely removed.

A frame may be referenced in one of two ways, either (i) from a *root* pointer in the execution state of the program, or (ii) from another frame. We focus first on category (ii), for which we can give a simple and largely language-independent characterization. All our formal definitions are in Figure 23.

Frame-to-frame pointers. We write $h \vdash f \rightsquigarrow f'$ if frame f in heap h makes a *direct reference* to frame f', either through a link or via a slot value v. The latter case, which we write $v \rightsquigarrow f'$, depends on the definition of values in our language. A frame f' is *reachable* from another frame f if there is a sequence of direct references from f to f'; this is just the reflexive transitive closure of the reference relation, so we write it $h \vdash f \rightsquigarrow^* f'$. We write $h \vdash \not\rightsquigarrow f$ if no frame in h references f.

The key observation is that it is safe to remove any set of frames fs from a heap h provided that no frame in fs is referenced from the *resulting* heap h', written $\mathsf{safeRemoval}(h, fs, h')$. In other words, a safe removal is one that does not produce any new dangling pointers. We then have the following easy lemma:

▶ Lemma 4 (Safe removal preserves heap invariants). Suppose $\mathsf{goodHeap}(h)$ holds. Then, for any set of frames fs and heap h', $\mathsf{safeRemoval}(h, fs, h')$ implies $\mathsf{goodHeap}(h')$.

Roots. How to track roots into the heap from the program state depends on the details of the language and the semantic approach. For the big-step semantic style used in this paper, most live frames are reachable from the "current" frame (f in the judgments $f \vdash a/h \Rightarrow v/h'$). However, evaluation of certain big-step rules may introduce other frames, not reachable from the current frame, that must be temporarily registered as roots. To handle these, we add an additional root frame set component rs to the configurations described by the rules. Figure 23 shows the revised rule for application to illustrate how the auxiliary root set is used, here in two different ways: to save the closure frame while evaluating the argument and to save the calling context frame while evaluating the function body.

Several other rules also augment the root set; the need for auxiliary roots is unfortunately a bit ad-hoc and language-specific. We could limit the number of auxiliary roots by requiring

References from values

$\mathsf{ClosV}(x_i^{\mathsf{D}}, a, f) \rightsquigarrow f$ [RefClos]

$\mathsf{RecordV}(f) \rightsquigarrow f$ [RefRecord]

References from frames

$$\frac{\mathsf{get}(f, h, x_i^{\mathsf{D}}) \Rightarrow v \quad v \rightsquigarrow f'}{h \vdash f \rightsquigarrow f'}$$ [RefSlot]

$$\frac{\mathcal{K}_h(f)(l)(s) = f'}{h \vdash f \rightsquigarrow f'}$$ [RefLink]

Unreferenced frames $h \vdash \not\rightsquigarrow f \triangleq \forall f' \in Dom(h).\ h \vdash f' \not\rightsquigarrow f$

Removing frames from heaps

$\mathsf{safeRemoval}(h, fs, h') \triangleq h' = (h - fs) \wedge \forall f \in fs.\ h' \vdash \not\rightsquigarrow f$

$\mathsf{removeUnreferenced}(h, f, h') \triangleq h' = (h - \{f\}) \wedge h \vdash \not\rightsquigarrow f$

$\mathsf{removeAllUnreachable}(h, rs, fs, h') \triangleq h' = (h - fs) \wedge fs = \{f \mid \forall f' \in rs.\ h \vdash f' \not\rightsquigarrow^* f\}$

where $(h - fs)$ is the result of removing all frames in fs from h

Expression evaluation (selected rules) $f, rs \vdash a/h \Rightarrow v/h'$

$$\frac{f, rs \vdash a_1/h_1 \Rightarrow \mathsf{ClosV}(x_i^{\mathsf{D}}, a', f')/h_2 \quad f, \{f'\} \cup rs \vdash a_2/h_2 \Rightarrow v/h_3 \quad \mathsf{initFrame}(\mathcal{S}(a'), \{\mathbf{P} \mapsto \{\mathcal{S}_{h_3}(f') \mapsto f'\}\}, \{x_i^{\mathsf{D}} \mapsto v\})/h_3 \Rightarrow f''/h_4 \quad f'', \{f\} \cup rs \vdash a'/h_4 \Rightarrow v'/h_5}{f, rs \vdash a_1(a_2)/h_1 \Rightarrow v'/h_5}$$ [EvApp]

$$\frac{\mathsf{safeRemoval}(h, fs, h') \quad fs \cap (\{f\} \cup rs) = \emptyset \quad f, rs \vdash a/h' \Rightarrow v/h''}{f, rs \vdash a/h \Rightarrow v/h''}$$ [EvGC]

Figure 23 Definitions for Garbage Collection

source programs to be in A-normal form or continuation-passing style, which both have the effect of putting more rule-internal values into named variables, and hence into the current frame.

Type-safe collection. Finally, we can add Rule [EvGC] for garbage collection shown in Figure 23, which can be applied non-deterministically prior to any of the syntax-directed rules. The resulting system still enjoys type soundness. The proof is straightforward once we strengthen the inductive invariant at each big step to say that $\{f\} \cup rs \subseteq Dom(h)$. The proof case for Rule [EvGC] relies on Lemma 4.

Refinements. To show that safeRemoval is a reasonable specification, we note that it can be refined to specifications of two well-known GC algorithms. removeUnreferenced specifies removal of a single unreferenced frame; it models one step in a reference counting collection. removeAllUnreachable specifies removal of *all* frames unreachable from an arbitrary root set; it models a *complete* tracing-based collector, such as a mark-and-sweep or copying collector.

▶ Lemma 5 (Unreferenced and unreachable frames are safe to remove). It holds that:
(a) $\mathsf{removeUnreferenced}(h, f, h')$ implies $\mathsf{safeRemoval}(h, \{f\}, h')$; and
(b) $\mathsf{removeAllUnreachable}(h, rs, fs, h')$ implies $\mathsf{safeRemoval}(h, fs, h')$.

We have not yet developed a concrete GC implementation, but writing one should be straightforward. The heap-traversal part of an implementation can be language-independent except for determining references from values. Note that, for many languages, an implementation can use the scope labels on frames to detect all references efficiently, without the

need for tagging individual pointers. This follows from the goodFrame property, which states that every frame slot corresponds to a statically known typed declaration in the scope; if the language is sufficiently monomorphic or restricts polymorphism to boxed types, we can consult the scope description to determine whether or not the slot contains a pointer.

6 Discussion

This paper presents a systematic correspondence between static name binding and binding at run time. We have also proposed the correspondence as a guiding principle for dynamic semantic specification, and a language-independent principle for proving type soundness. In this section we discuss our approach and compare with previous work on type soundness and, more generally, on representing and reasoning about semantics and memory layout.

Type soundness. Milner [11] summarized the essence of type soundness in his famous catch-phrase: "well-typed programs cannot go wrong". The common point of type soundness proofs is that they provide this guarantee. Other than that, type soundness proofs come in many different varieties and flavours, depending on both the underlying semantic style (such as big-step vs. small-step) and the underlying language being specified.

Semantic specification and type soundness. Wright and Felleisen's *syntactic approach to type soundness* [28] argues in favor of using small-step reduction semantics and evaluation contexts for type soundness proofs. This avoids some of the shortcomings of big-step semantics, namely that big-step rules do not distinguish getting stuck and diverging. The small-step style is also better suited for formalizing semantics with concurrency and/or interleaving. However, this kind of specification does not correspond to how programming language interpreters are typically implemented.

This paper uses big-step semantics (or *natural semantics* [7]) for type soundness. In order to prove type soundness we have been following in the footsteps of Milner [11] and added explicit rules for going wrong to our language. An inherent danger of this approach is that leaving out such a wrong rule may render the type soundness theorem vacuous. We stress that the approach of using scopes to describe frames is by no means limited to big-step semantics: we expect it to be equally applicable to other styles of semantic specification, including small-step SOS [16], reduction semantics [5], definitional interpreters [18], and abstract machines. We chose big-step semantics because it corresponds to how programming language interpreters are typically implemented. In the future, we plan to use I-MSOS [13] and DynSem [26] to make our big-step rules even more concise. We also plan to investigate ways of alleviating some of the drawbacks of big-step type soundness proofs proposed in the literature, such as:

- using coinductive big-step semantics to represent diverging computations and proving big-step progress, following Leroy and Grall [10];
- checking that wrong rules are correctly defined either by using a *coverage lemma* as proposed by Ernst et al. [4] or using Charguéraud's *pretty-big-step* style [1] with a novel big-step progress predicate, which subsumes explicit wrong rules in a safe way (e.g., failure to add sufficient rules for progress makes it impossible to prove type soundness);
- using a functional definitional interpreter instrumented with a clock (or "fuel") which counts down with each function call [22, 15, 19].

The modularity of the small-step approach in [28] comes from the use of reduction semantics, which supports localizing new patterns of name binding to particular constructs,

such that proofs of existing constructs do not need to change. In contrast, our approach using scopes to describe frames scales to deal with name binding of both functional and imperative features in a uniform manner. Thus, there is no need to use different stores or notions of substitution or to localize these to particular constructs: name binding is uniformly handled both in the dynamic semantics and in the proof.

A selling point of the syntactic approach is that it scales to prove the type soundness of an ML-like language with Hindley-Milner polymorphism such that the structure of proofs change relatively little as new features are added. Indeed, many of the challenges with substitution in [28] stem from dealing with polymorphism and references. We believe that describing scopes as frames will scale to deal with polymorphic type inference, and suspect it might alleviate some of the difficulties inherent to substitution lemmas, but leave it to future work to verify this.

Default values. A shortcoming of our approach compared with, e.g., Wright and Felleisen [28], is our use of default values. Default values make the proofs easy, but introduce the risk of dereferencing null values. In order to deal with function types, we proposed to generate "default functions" too, but this approach is restricted to simple type systems, since the problem of finding inhabitants of a given type quickly becomes undecidable. In the future, we plan to investigate extending our type soundness principle to temporarily allow a frame to be ill-typed, so long as it is initialized when it is accessed. Nipkow et al.'s [9] formalization of Java's definite assignment analysis could provide a useful guide to this end.

Modeling memory in programming language semantics. In addition to the different semantic specification styles, there is a proliferation of ways to deal with name binding and memory management in semantic specifications. For example, consider a type soundness proof for Java-like languages, where formalization of name binding and memory ranges from simple states mapping identifiers and references to values, as used by Drossopoulou and Eisenbach [3], to untyped frames [24] relying on traditional environments for typing, to use of ad-hoc lookup functions (or visibility predicates) uniquely defined to resolve a specific kind of identifiers (e.g. classes or fields) as used in Featherweight Java [6] or Jinja [9].

Similarly, considering the state of affairs in semantic specification frameworks gives a similarly muddled picture. These frameworks can be roughly categorized into two camps: those that deal with binding via substitution (redex [8], Ott [21]), and those that provide support for ad hoc binding via auxiliary entities (K [20], funcons [2]).

None of these pre-existing approaches hits the same sweet spot as our framework: it corresponds to how memory is often organized in real implementations (as also remarked by Syme [24, page 7]); it scales to deal with various models of name binding and memory in a uniform manner; and it provides a language-independent principle for proving the absence of binding and typing errors.

We also remark that our compartmentalization of binding and typing as separate concerns and separate sets of rules in theory allows us to prove the absence of binding errors and typing errors separately. This may be useful for dynamically-typed languages, where proving the absence of typing errors might not be a concern, but the absence of binding errors is.

7 Conclusion

This paper presented a systematic and uniform correspondence between static scopes and dynamic frames. This correspondence provides a mostly separate specification of binding

and typing. This is a novelty compared to traditional approaches, where typing rules usually describe both name binding and typing, occasionally relying on ad hoc predicates (such as those found in, e.g., Featherweight Java). The scopes-as-frames paradigm supports uniform and straightforward type soundness proofs for a number of different language features, as demonstrated by applying it to a simple functional language (Section 3) and a language with first-class functions and records (Section 4). Section 5 shows how the invariants also support verifying the safety of a number of different garbage collection schemes. Section 4.4 discusses, and our accompanying tech report [17] shows, how the approach also scales to a class-based object-oriented language with sub-typing and run-time upcasting.

In future work, we plan to investigate how the approach scales to larger languages including languages with type-level polymorphism; generating Coq infrastructure for mechanizing type soundness proofs using the language-independent framework for well-boundness and well-typedness developed for this paper; and how to derive efficiently executable prototype interpreters based on dynamic semantic specifications in DynSem.

Acknowledgments. We thank Sebastian Erdweg, Peter D. Mosses, and the anonymous reviewers for their feedback on previous versions of this paper. This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206). Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.

References

1 Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *ESOP'13*, volume 7792 of *LNCS*, pages 41–60. Springer, 2013.

2 Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable components of semantic specifications. *Transactions on Aspect-Oriented Software Development*, 12:132–179, 2015.

3 Sophia Drossopoulou and Susan Eisenbach. Java is type safe - probably. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97*, volume 1241 of *LNCS*, pages 389–418. Springer, 1997.

4 Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL'06*, pages 270–282. ACM, 2006.

5 Matthias Felleisen. *The Calculi of λ-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. PhD thesis, Indiana University, 1987.

6 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

7 Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS'87*, volume 247 of *LNCS*, pages 22–39. Springer, 1987.

8 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In John Field and Michael Hicks, editors, *POPL'12*, pages 285–296. ACM, 2012.

9 Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *TOPLAS*, 28(4):619–695, 2006.

10 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

11 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

12 J. Gregory Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA'95*, pages 66–77, 1995.

13 Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *ENTCS*, 229(4):49–66, 2009.

14 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *ESOP'15*, volume 9032 of *LNCS*, pages 205–231. Springer, 2015.

15 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *ESOP'16*, volume 9632 of *LNCS*, pages 589–615. Springer, 2016.

16 Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

17 Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. Technical Report TUD-SERG-2016-010, Delft University of Technology, Programming Languages Research Group, Delft, The Netherlands, 2016.

18 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

19 Tiark Rompf and Nada Amin. From F to DOT: Type soundness proofs with definitional interpreters. 2015. `http://arxiv.org/abs/1510.05216`.

20 Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

21 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

22 Jeremy G. Siek. Type safety in three easy lemmas, May 2013. `http://siek.blogspot.co.uk/2013/05/type-safety-in-three-easy-lemmas.html`.

23 Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

24 Don Syme. Proving Java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 83–118. Springer, 1999.

25 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *PEPM'16*, pages 49–60. ACM, 2016.

26 Vlad A. Vergu, Pierre Néron, and Eelco Visser. DynSem: A DSL for dynamic semantics specification. In Maribel Fernández, editor, *RTA'15*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

27 Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward!'14*, pages 95–111. ACM, 2014.

28 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.