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

The connection between the names and scopes in the static structure of a program and memory and memory access during its execution is not systematically made in specifications of programming language semantics. 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 binding structures and run-time memory layout. The approach scales to a range of binding features, makes type soundness proofs straightforward, and provides the basis for a language-independent specification of sound reachability-based garbage collection.

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 memory and memory access during its execution. However, this connection is not made systematically in specifications of programming language semantics. There is a proliferation of different ways to deal with name binding and memory management in semantics specifications. For example, in type soundness proofs for Java-like languages, formalization of name binding and memory ranges from simple states mapping identifiers and references to values [3], to untyped frames relying on traditional environments for typing [23], to use of ad-hoc lookup functions (or visibility predicates) uniquely defined to resolve a specific kind of identifiers such as classes or fields [5, 8]. Dynamic semantic specification frameworks (redex [7], Ott [20], K [19], funcons [2]) provide little guidance in formalizing the connection between static names and dynamic memory.

In this paper, we introduce a systematic approach to aligning names in static semantics with memory in dynamic semantics. The approach builds on the scope graph framework for name resolution of Neron et al. [14], which provides a language-independent model for the representation of static name binding of programs. To match that representation, 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 binding structures and run-time memory layout.

The main contributions of this paper are:

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

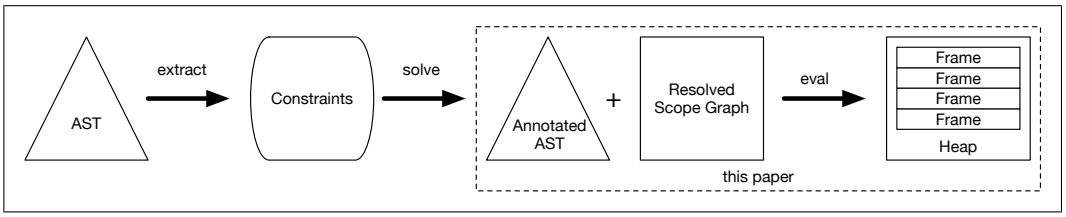

Figure 1 Architecture of the approach: static analysis of a program abstract syntax tree via constraints leads to a representation with explicit name and type information, which is the input for evaluation.

- We introduce the frames-as-scopes 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.
- We show how phrasing the consistency between frames and scopes as an invariant gives a *language-independent principle* that supports straightforward *type soundness* proofs.
- We demonstrate that the approach scales by applying it to a range of binding features.
- We discuss a high-level *language-independent* specification of *sound garbage collection* of frames, and verify that several standard GC strategies refine this specification.

In the next section we introduce the approach by means of an example. In Section 3 we define a complete formalization of a small language with first-class functions using the approach, including a discussion of its type soundness proof. 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 and ideas for future work. We plan to submit the Coq development incorporating these ideas to the artifact evaluations.

2 Linking Static Names to Dynamic Slots

We introduce a systematic approach to the alignment of names in static and dynamic semantics. The approach builds on the theory of name resolution of Neron et al. [14], which provides the foundation for a language-independent representation of name binding in programs based 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 [24]. Third, an interpreter evaluates this data structure using a run-time heap consisting of frames to represent the memory of the program. The shape of the run-time frames is determined by the scopes in the static scope graph. This relation is systematic and provides structure to the type soundness proof. In this paper we focus on the third stage. Our starting point is a well-bound and well-typed program represented by means of an annotated AST and a resolved scope graph produced by name and type resolution.

In the next sections we develop the approach formally using a series of model languages. Before we dive into the formal details, we describe the connection between static scope graphs and dynamic heaps by means of an example. Figure 2 (left) shows an example program defining the `Lst` (list) data structure as a record with a `hd` and a `tl` field (the empty list being represented by `null`). The `ft(x,y)` (from-to) function generates a list with the numbers

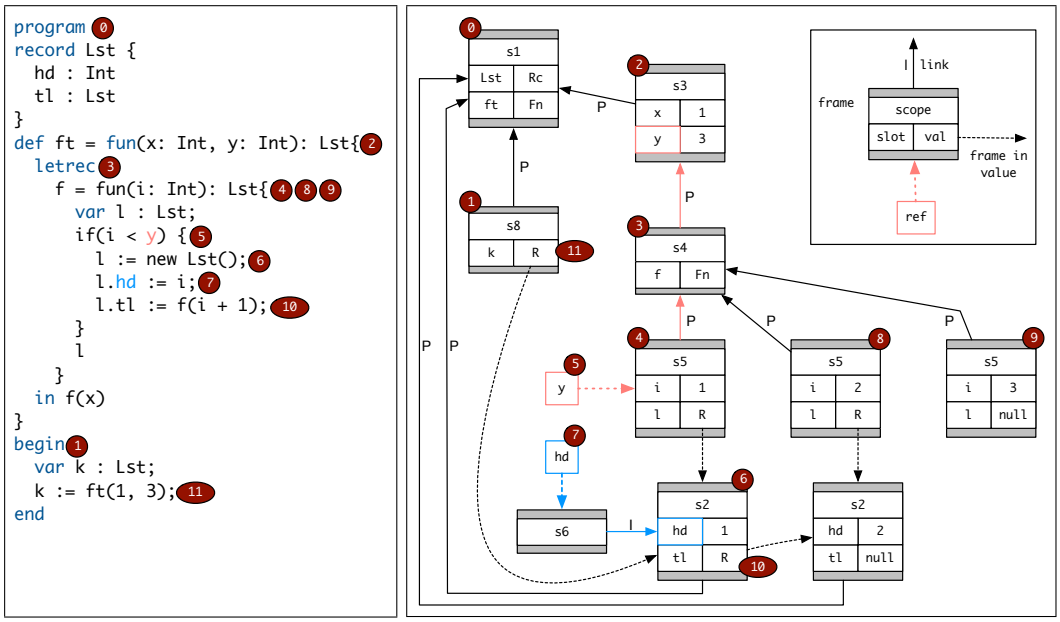

Figure 2 Example program (left) and snapshot of heap with frames for execution of that program (right). The callouts refer to points in the execution of the program. The dotted edges are frame pointers embedded in values. The colored references (`y` and `hd`) illustrate in which frames these references are evaluated; the references are not part of the heap.

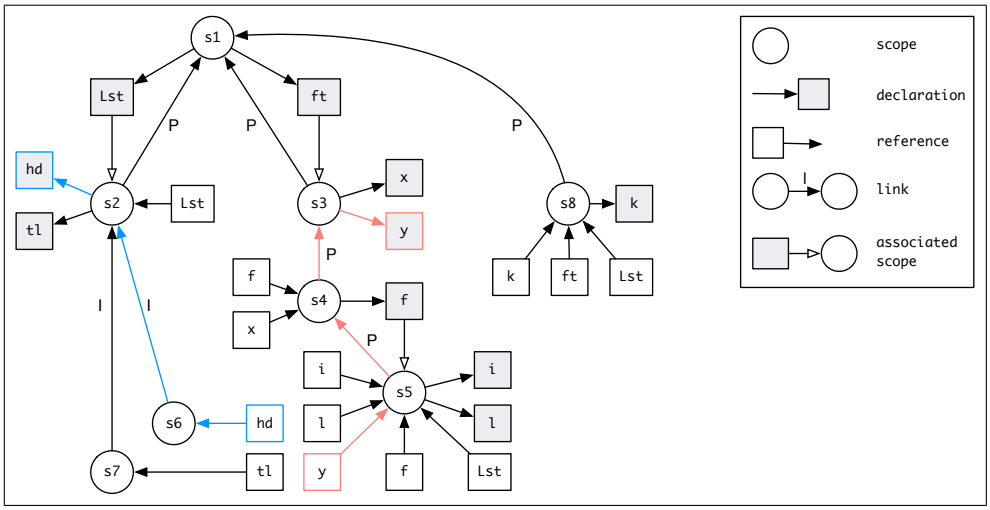

Figure 3 Static scope graph for example program from Figure 2. The legend explains the graphical notation. The colored references, edges, and declarations illustrate paths from references to declarations.

between `x` and `y` (non-inclusive). The function is defined using a nested recursive function `f` that does the real work, relying on the value of `y` in its closure. Function `ft` is called in the main body of the program, assigning the resulting value to local variable `k`. (Note that this example uses an extension of the language with records formally defined in Section 4 with `if` and `letrec` expressions, n-ary instead of unary functions, top-level function definitions, and local variables. However, the scope graph and frame representations are used similarly as for that language.)

Scope Graphs. Figure 3 defines the *static scope graph* for the program of Figure 2 (left). The legend explains the graphical notation. A scope graph is an abstraction of an abstract syntax tree representing the name binding relations of a program. A *scope* is an abstraction over a group of nodes in the abstract syntax tree that behave uniformly with respect to name resolution. For example, scope `s1` corresponds to the whole program, scope `s3` corresponds to the function `ft`, scope `s4` corresponds to the `letrec`, and scope `s5` corresponds to the function `f`. A *declaration* introduces a name in a scope. For example, function `ft` is declared in scope `s1` and variable `l` is declared in scope `s5`. A declaration is only visible in the scope in which it is declared, and in scopes that can reach that scope through edges between scopes. These *links* represent visibility idioms such as nesting in lexical scoping and imports in module systems. A *reference* represents the use of a name in a scope and is *resolved* via a path along the edges of the scope graph. For example, the reference `y` in scope `s5` is resolved to the declaration `y` in scope `s3`.

Resolution of a scope graph consists in finding a path to a declaration for each reference and possibly disambiguating between multiple paths for a reference, for example by applying shadowing rules. In this paper, we are not concerned with resolution. We start with a resolved scope graph in which each reference is resolved and in which we can consider names of declarations to be distinct. That is, each declaration is uniquely identified and each reference is characterized by a unique path in the scope graph. Our use of concrete names in examples is for the convenience of the reader.

Heaps and Frames. The key idea of this paper is that the layout of memory at run-time is determined by the static scope graph. The memory of a running program is represented by a heap consisting of frames. A frame is a region of memory consisting of a scope identifier, a set of slots mapping slot identifiers to values, and a set of links to other frames. The links of a frame connect it to other frames, i.e, the heap is a frame graph. A link identifies a frame through its frame identifier (pointer), depicted as an arrow between frames in diagrams. In addition to a direct link, a frame can also refer to other frames through frame identifiers stored as part of values assigned to slots. Given a frame pointer, the value of a slot can be retrieved by following the path to the frame it is contained in. The structure of a frame is determined by its scope in a static scope graph as illustrated in Figure 4. A frame instantiates the scope corresponding to its scope identifier. A frame has a slot for each declaration in the scope (and declarations are used as slot identifiers). For each link from a frame to another, there is a corresponding link from its scope to another scope, such that the source and target frames instantiate the source and target scopes; i.e. the diagram in Figure 4 commutes.

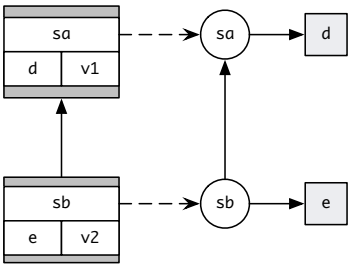

Figure 4 Frames instantiate scopes.

Example. Figure 2 (right) illustrates the approach with a snapshot of the heap created by the execution of the example program. The snapshot corresponds to the state of the heap at point (11) in the execution of the program, after the assignment to variable `k` (and before any non-reachable frames have been garbage collected). We examine the execution steps of the program to see how scopes describe frames.

- The invocation of the program leads to the instantiation of a frame for scope `s1`.

- The execution of the body of the program at point (1) leads to the instantiation of scope `s8` with a slot for local variable `k` and a link to the `s1` frame as its lexical context.
- The call to function `ft` from the program body leads to the instantiation of scope `s3` at point (2) with initialization of slots `x` and `y` with the values of the actual arguments and the `s1` frame as its lexical context.
- The execution of the `letrec` in the body of `ft` leads to the instantiation of scope `s4` with the `s3` frame as its lexical context. The `f` slot is filled with the function value and has the `s4` frame as its static link.
- The call `f(x)` in the body of the `letrec` is evaluated in the context of the `s4` frame. The call leads to the instantiation of scope `s5` for the function value of `f` with slots for declarations `i` (the formal parameter) and `l` (the local variable). The parameter slot `i` is assigned the value `1` of the actual argument `x`. The local variable slot `l` is initialized to the default value `null`. Next, the body of the function is evaluated in the context of this new frame.
- At point (5) the value of the variable `y` is retrieved by interpreting its path in the static scope graph as path in the frame graph relative to the frame of execution of the function. Note the correspondence between the paths from reference `y` to declaration `y` in the scope graph and the heap.
- At point (6) the evaluation of `new Lst()` creates a frame that instantiates the associated scope `s2` of the `Lst` record type. The `hd` and `tl` slots are initialized with default values `0` and `null`, respectively. The resulting value, a record value (`R`) whose only argument is a pointer to a record frame, is assigned to the slot representing local variable `l`.
- At point (7) the value of the slot `i` is assigned to the slot `hd` of the new record frame. (We will discuss in Section 4 the structure of the scope graph for field access expressions.)
- Recursive calls at (8) and (9) to function `f` then lead to a repetition of this pattern, with creation of new frames instantiating the `s5` scope of the function, and the `s2` scope of the record type.
- In the third invocation of `f` at point (9) the recursion terminates and the function `f` returns the default value `null` in slot `l`. The second invocation at point (8) returns a record value, and at point (10) assigns that to the `tl` slot of the first record frame.
- The first record value is returned from the invocation of `f`, the `letrec`, and the invocation of `ft`, and eventually assigned to the local variable slot `k` at (11).

A Uniform Model. This example illustrates how name binding and memory layout is treated uniformly. The evaluation of functions, let bindings, blocks with local variables, and records all follow the same pattern. In the static scope graph these binding constructs are represented by means of a scope with declarations for each of the names they introduce. Invocation of these constructs at run-time leads to frames that instantiate their scopes. Evaluating a reference to a value is achieved by taking its path in the static scope graph and interpreting it relative to the frame of interpretation in the runtime heap.

Types and Type Soundness. In the example above, we have sketched how the layout of memory in frames follows the scope graph. We can phrase this as an invariant that should be observed by the dynamic semantics. This gives a basis for statically checking and ruling out runtime binding errors, such as trying to read uninitialized slots from memory.

But we can go further than the shape of memory by using types to classify the kinds of values that program expressions compute and that memory locations contain. Associating each declaration in a scope graph with a type, the invariant to be observed by dynamic

semantics can be strengthened to insist that each slot contains a value of the same type as the corresponding declaration. This gives a basis for statically checking and ruling out both binding errors and type errors.

In a nutshell, the binding and typing invariants for frames are summarized as:

- Binding invariant: the links of a frame should correspond to the links of the scope, and a frame should provide a slot for each declaration of the scope.
- Typing invariant: the values in the slots of a frame should correspond to the types of the corresponding declarations of the scope.

These invariants extend to heaps by requiring that all frames in a heap satisfy this invariant. We argue that this provides a foundation for proving type soundness in a uniform manner that scales to a wide range of language features and binding patterns, and that using scopes to describe frames supports a style of operational dynamic semantics that is concise, precise, and reminiscent of realistic implementations of programming languages.

In the next sections we make these claims and ideas precise by formalizing the framework of scope graphs and heaps with frames, using it to define the static and dynamic semantics of a series of model languages representing typical binding patterns of programming languages, discussing the verification of type soundness for these languages, and showing how we can extend this framework to safe 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 [24] to provide a uniform (language-independent) treatment of name binding and type assignment in the definition of the static semantics of 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. Hence, 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 labelled edges for *linking* two scopes. A labelled link $s \xrightarrow{\mathbf{P}} s'$ means that s has s' is its *lexical parent*. On the other hand, a labelled link $s \xrightarrow{\mathbf{I}} s'$ means that s *imports* s'. For language L1,

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

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

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

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

Figure 6 Resolution paths

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 which s links to 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*. A path p is interpreted relative to a reference scope, and is given by a sequence of resolution steps. The resolution step $\mathbf{D}(x_i^{\mathsf{D}})$ states that the declaration x_i^{D} is available in the reference scope, while $\mathbf{E}(l, s)$ states that resolution continues by following the edge labeled by l from the reference scope to scope s. The path consistency relation $\mathcal{G} \vdash^{\mathsf{G}} p : s \longmapsto (s', x_i^{\mathsf{D}})$ checks that a path p is consistent with the scope graph, i.e. that it is valid in scope s and leads to a scope s' with declaration x_i^{D}.

In this paper, we assume that scope graphs have been resolved, following the resolution calculus of [14, 24], 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 such that $\mathcal{G} \vdash^{\mathsf{G}} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$. When there are multiple possible resolution paths for a reference, the resolution calculus uses language-dependent 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. Here, the $\xrightarrow{\mathbf{P}}$ edges connect the inner scope of each function to its enclosing (lexical) scope. For example, the reference $\mathrm{x}_6^{\mathsf{R}}$ has a trivial path in the scope graph. We write $p_6 = \mathbf{D}(\mathrm{x}_1^{\mathsf{D}}) : \mathrm{x}_6^{\mathsf{R}} \longmapsto \mathrm{x}_1^{\mathsf{D}}$ for this resolution path, which says that the declaration

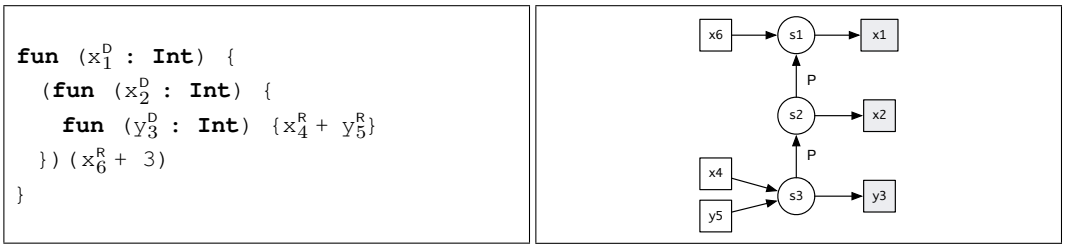

Figure 7 Example program with nested functions and its scope graph.

$\mathtt{x}_1^{\mathsf{D}}$ 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 $p_5 = \mathbf{D}(\mathtt{y}_3^{\mathsf{D}}) : \mathtt{y}_5^{\mathsf{R}} \longmapsto \mathtt{x}_3^{\mathsf{D}}$. 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 $\{\mathtt{x}_6^{\mathsf{D}} \mapsto p_6, \mathtt{y}_5^{\mathsf{D}} \mapsto p_5, \mathtt{x}_4^{\mathsf{D}} \mapsto p_4\}$.

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. Furthermore, identifiers have been replaced with unique references and declarations. We define well-boundness and well-typedness as complementary 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.

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 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 ($\mathcal{G} \overset{\mathsf{G}}{\vdash} x_i^{\mathsf{R}} \longrightarrow s$), and that the reference resolves to some declaration x_i^{D} in the scope graph. 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 references are consistently typed with their 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. However, note the absence of environment threading in the well-typedness rules; scope management is taken care of by the well-boundness rules. The only rule where well-typedness and well-boundness overlap is Rule [WtRef], which refers to the resolution of a reference (variable) in the scope graph in order to check the consistency with the type of its declaration.

$$
\begin{array}{l}
t ::= \mathbf{Int} \mid t \to t \\
e ::= z \mid id \mid e \oplus e \mid \mathbf{fun}(id\ :\ t)\{\ e\ \} \mid e(e)
\end{array}
$$

Figure 8 Syntax of L1

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

Figure 9 Annotated syntax of L1

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

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

$$\frac{\mathcal{G} \vdash^{\mathrm{G}} x_i^{\mathrm{R}} \longrightarrow s \quad \mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_j^{\mathrm{D}}}{\vdash^{\mathrm{B}} (x_i^{\mathrm{R}})^{s}} \quad [\mathsf{WbRef}]$$

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

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

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

Figure 10 Well-boundness of L1 terms

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

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

$$\frac{\mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_j^{\mathrm{D}} \quad \mathcal{G} \vdash^{\mathrm{G}} x_j^{\mathrm{D}} : t}{\vdash^{\mathrm{T}} (x_i^{\mathrm{R}})^{t}} \quad [\mathsf{WtRef}]$$

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

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

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

Figure 11 Well-typedness of L1 terms

3.3 Frames and Heaps

We now develop the formalization of the language-independent framework of heaps and frames to represent memory at run-time. Figure 12 defines frames, heaps, and some operations on frames 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 by means of 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 frame to the frame that represents its lexical context or an imported module. The slots of a frame are defined by means of a finite map from declarations to values.

We define operations on heaps to construct new frames and retrieve slot values. Note that all operation 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. Note that the operation is defined as the relation $\vdash^{\mathrm{F}} \mathsf{initFrame}(s, ks, \sigma)/h \Rightarrow f'/h'$, 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 $f, h \models p \Rightarrow (f', x_i^{\mathrm{D}})$ dereferences a path p relative to frame f,

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

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

Operations on frames

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

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

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

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

Figure 12 A language independent formalization of frames and heaps

Values

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

Annotation projections

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

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

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

$$\frac{\mathcal{G} \vdash^{\mathsf{G}} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}} \quad f, h \models p \Rightarrow (f', x_j^{\mathsf{D}}) \quad f', h \models \mathsf{get}(x_j^{\mathsf{D}}) \Rightarrow v}{f \vdash x_i^{\mathsf{R}}/h \Rightarrow v/h} \quad [\mathsf{EvRef}]$$

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

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

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

Figure 13 Dynamic semantics of L1

resulting in an *address* consisting of a pair of a frame f' and a slot x_i^{D} in that frame. The operation $f, h \models \mathsf{get}(x_i^{\mathsf{D}}) \Rightarrow v$ retrieves the value of slot x_i^{D} from frame f.

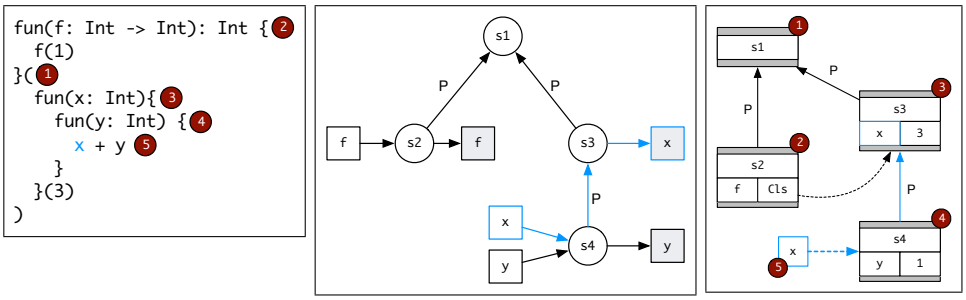

Figure 14 Example L1 program with scope graph and heap.

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 context.
- 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`.)

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 Appendix C shows 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, 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 (Figure 4). Now we make that precise and formally define the correspondence between frames and scopes, and discuss how that correspondence supports a language-independent formulation and systematic proof of type soundness.

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, that a frame f provides exactly one slot for each declaration in the corresponding scope s, and a set of dynamic links, such that for each link $\{l \mapsto s'\} \in \mathcal{K}(s)$, there is a dynamic link $\mathcal{K}_h(f)(l)(s') = f'$ such that f' instantiates 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, it holds that 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: (1) the lexical frame of the closure is an actual frame in the heap; (2) that the body of the closure scopes exactly the formal parameter of the closure; (3) that the formal parameter has the right type; (4) that the scope

Good frame

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

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

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

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

Good heap

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

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

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

$$\frac{f \in \mathit{Dom}(h)\quad \mathcal{G} \overset{\mathrm{G}}{\vdash} \mathcal{D}(s) = \{x_i^{\mathrm{D}}\}\quad \mathcal{G} \overset{\mathrm{G}}{\vdash} x_i^{\mathrm{D}} : t_1 \qquad \mathcal{K}(s) = \{\mathbf{P} \mapsto \{\mathcal{S}_h(f)\}\}\quad \overset{\mathrm{B}}{\vdash} e^{s}\quad \overset{\mathrm{T}}{\vdash} e^{t_2}}{h \overset{\mathrm{V}}{\vdash} \mathsf{ClosV}(x_i^{\mathrm{D}}, e^{s,t_2}, f) : t_1 \to t_2} \qquad [\mathsf{VtClo}]$$

Figure 15 Good frames, good heaps, and value typing

of the body of the closure is dominated by the scope of the lexical frame; and (5) that the body of the function is well-bound and well-typed.

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.

Good Paths. Lemma 1 formalizes the property that, for each consistent static path, looking up the path dynamically results in a corresponding runtime address (i.e., a pair of a frame identifier and a declaration).

▶ Lemma 1 (Good paths).

$$\forall h\ s\ f\ s'\ p.\ \mathsf{goodHeap}(h)\ \implies\ \mathcal{S}_h(f) = s\ \implies\ \mathcal{G} \overset{\mathrm{G}}{\vdash} p : s \longmapsto (s', x_i^{\mathrm{D}})\ \implies\ \exists f'.\ f, h \models p \Rightarrow (f', x_i^{\mathrm{D}})\ \wedge\ \mathcal{S}_h(f') = s'$$

Type Preservation. Theorem 1 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 1** (Type preservation for L1).

$$\forall f\ s\ t\ e\ h_1\ h_2\ v.\ f \vdash e^{s,t}/h_1 \Rightarrow v/h_2\ \implies\ \mathsf{goodHeap}(h_1)\ \implies\ f \in \mathit{Dom}(h_1)\ \implies\ \mathcal{S}_{h_1}(f) = s\ \implies \overset{\mathrm{B}}{\vdash} e^{s}\ \implies \overset{\mathrm{T}}{\vdash} e^{t}\ \implies\ \mathsf{goodHeap}(h_2)\ \wedge\ h_2 \overset{\mathrm{V}}{\vdash} v : t$$

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 an explicit "wrong" value to a big-step semantics anywhere evaluation can get stuck. This allows distinguishing between getting stuck and divergence, which is otherwise impossible using the inductive big-step rules in Figure 13, but which is necessary in order to prove type soundness. Appendix A presents the rules that are necessary to add to the language. 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 extend our approach to L2, an extension of L1 with records and imperative features, and show 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 can also provide a basis for giving and verifying a sound semantics for L2.

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 fields that a record consists of at runtime. Records are constructed using the **`new`** construct, which takes a reference that refers to a record type definition. L2 also adds an assignment operation (_ `:=` _), which assigns a value to a memory address. Memory addresses (or *L-values* [22]) 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 (_`;` _). The example program in Section 2 is an extension of L2, i.e. the language has `if` and `letrec` expressions, top-level function definitions, and local variables. However, the discussion of the structure of the scope graph and the corresponding heap provides a good illustration of the formal definitions in this section.

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: $\vdash^{\mathrm{B}}_{\mathrm{P}}, \vdash^{\mathrm{T}}_{\mathrm{P}}$ for programs, $\vdash^{\mathrm{B}}_{\mathrm{D}}, \vdash^{\mathrm{T}}_{\mathrm{D}}$ for declarations, and $\vdash^{\mathrm{B}}_{\mathrm{L}}, \vdash^{\mathrm{T}}_{\mathrm{L}}$ 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.
- 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 in a declaration in scope s'.
- Rules [WbFAcc] and [WtFAcc] describe the static semantics of field access. In a field access expression such as $a.x_i^{\mathrm{R}}$, the annotated expression a computes a record, and the reference x_i^{R} refers to a declaration in this record. The auxiliary scope s' makes the declarations of the record accessible by importing the scope s_{rec} associated with the record type $\mathbf{Rec}\,(x_j^{\mathrm{D}})$, and declaring the field x_i^{R} as a reference in s'.
- Rule [WtAsgn] uses the projection function $\mathcal{R}(lhs)$ for projecting the reference part of an lhs expression and checking its type. This precludes the need for $\vdash^{\mathrm{T}}_{\mathrm{L}}$ judgments to

constrain types, and makes Rule [WtLVar] very simple (any variable is a well-typed lhs expression).

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 the additions.

Rule [EvNew] specifies that evaluating a **new** expression constructs a frame via initDefault(s, ks), which takes as a scope identifier s and a map of dynamic links that the frame should be initialized with, and instantiates all slots of the frame with *default values*. Here, the $\vdash^{\mathrm{D}}_{\mathrm{V}}$ judgment in Figure 20 defines the default value for all types in L2. We introduce a special default function (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. Rule [LhsFAccNull] handles the case of dereferencing a null value through a field access expression by raising a null-pointer exception. Not shown in Figure 19 are rules for propagating such exceptions. The omitted rules are summarized in Appendix B and are the obvious ones for propagating exceptions. There are several reasons why we choose to let frames contain null values by default: 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. This import frame could effectively have been replaced by an operation that strips away the first import step of the path for the reference of the lhs expression and accesses the record frame resulting from evaluating the receiver expression. The semantics of Rule [LhsFAcc] is, however, more straightforwardly proven to preserve the well-boundness and well-typedness invariants of the heap.

In order to evaluate lhs expressions, a new judgment $f \vdash^{\mathrm{LHS}} lhs/h \Rightarrow u/h'$ 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, and also defined in Figure 19.

Note that Rule [EvAsgn] defines assignment not just for record fields but also for variables, which makes variables bound by functions mutable. For the invariants and the type soundness proof, this is completely unproblematic.

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 admits such exceptions to occur anywhere in well-typed programs. Theorem 2 states type preservation for the newly introduced constructs between L1 and L2 and does not involve nasty surprises.

▶ **Theorem 2** (Type preservation for L2)**.**

$$\begin{aligned}\forall f\ s\ t\ e\ h_1\ h_2\ v.\ f \vdash e^{s,t}/h_1 \Rightarrow v/h_2 \implies \mathsf{goodHeap}(h_1) \implies \\ f \in Dom(h_1) \implies \mathcal{S}(h_1)f = s \implies \vdash^{\mathrm{B}} e^s \implies \vdash^{\mathrm{T}} e^t \implies \\ \mathsf{goodHeap}(h_2) \ \wedge\ h_2 \vdash^{\mathrm{V}} v : t\end{aligned}$$

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

Figure 16 Annotated syntax of L2 with distinguished references and declarations

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

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

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

$$\frac{\begin{array}{l}\mathcal{G} \vdash^{\mathrm{G}} s \longrightarrow x_i^{\mathrm{D}} \quad \mathcal{G} \vdash^{\mathrm{G}} x_i^{\mathrm{D}} \longrightarrow\!\triangleright s' \\ \mathcal{G} \vdash^{\mathrm{G}} s' \xrightarrow{\mathbf{P}} s \\ \forall (x_j^{\mathrm{D}}\ \mathbf{:}\ t) \in \mathit{flds}.\ \mathcal{G} \vdash^{\mathrm{G}} s' \longrightarrow x_j^{\mathrm{D}}\end{array}}{\vdash^{\mathrm{B}}_{\mathrm{D}} (\mathbf{record}\ x_i^{\mathrm{D}}\ \{\ \mathit{flds}\ \})^s}\ [\mathsf{WbRD}]$$

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

$$\frac{\mathcal{G} \vdash^{\mathrm{G}} x_i^{\mathrm{R}} \longrightarrow s \quad \mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_j^{\mathrm{D}} \quad \mathcal{G} \vdash^{\mathrm{G}} x_j^{\mathrm{D}} \longrightarrow\!\triangleright s_{rec}}{\vdash^{\mathrm{B}} (\mathbf{new}\ x_i^{\mathrm{R}})^s}\ [\mathsf{WbNew}]$$

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

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

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

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

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

$$\frac{\mathcal{G} \vdash^{\mathrm{G}} x_i^{\mathrm{R}} \longrightarrow s \quad \mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_j^{\mathrm{D}}}{\vdash^{\mathrm{B}}_{\mathrm{L}} (x_i^{\mathrm{R}})^s}\ [\mathsf{WbLVar}]$$

Figure 17 Well-boundness in L2

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

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

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

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

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

$$\frac{\mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_j^{\mathrm{D}}}{\vdash^{\mathrm{T}} (\mathbf{new}\ x_i^{\mathrm{R}})^{\mathbf{Rec}\,(x_j^{\mathrm{D}})}}\ [\mathsf{WtNew}]$$

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

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

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

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

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

$$\frac{\mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_i^{\mathrm{D}} \quad \mathcal{G} \vdash^{\mathrm{G}} x_i^{\mathrm{D}} : t}{\vdash^{\mathrm{T}}_{\mathrm{L}} (x_i^{\mathrm{R}})^t}\ [\mathsf{WtLVar}]$$

Figure 18 Well-typedness in L2

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 19 Dynamic semantics of L2

Frame operations

$$f, h \models \mathsf{set}(x_i^{\mathrm{D}}, v) \Rightarrow h[f \mapsto (\mathcal{D}_h(f)[x_i^{\mathrm{D}} \mapsto v])] \ [\mathsf{SetSlot}]$$

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

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

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

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

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

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

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

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

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

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

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

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

Figure 21 Value-typing of records and null-values

As argued in connection with the proof of Theorem 1 in Section 3.6, we can extend this proof to a proof of type soundness by adding cases for "going wrong". The wrong cases follow along the same lines as the cases in the preservation proof.

4.4 From Records to Classes

In Appendix C (provided as an anonymized appendix with this submission) we present L3, an extension of L2 which replaces records with classes, which can inherit from other classes, and can override declarations from super classes. The appendix is not essential for understanding the approach of this paper, but it does demonstrate that the techniques we have presented, scope graphs to represent binding, frames to represent memory at run-time, and their correspondence to prove type soundness, scale to classes, objects, and subtyping.

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 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 sound (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 22.

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 2. 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 22 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 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 22, 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 2.

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. We have:

▶ Lemma 3. (a) $\mathsf{removeUnreferenced}(h, f, h') \implies \mathsf{safeRemoval}(h, \{f\}, h')$.
(b) $\forall rs.\ (\mathsf{removeAllUnreachable}(h, rs, fs, h') \implies \mathsf{safeRemoval}(h, fs, h'))$.

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.

References from values

$$\frac{}{\mathsf{ClosV}(x^{\mathrm{D}}_{,}a, f) \rightsquigarrow f}\ [\mathsf{RefClos}]$$

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

References from frames

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

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

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

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) $\quad \boxed{f, rs \vdash a/h \Rightarrow v/h'}$

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

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

Figure 22 Definitions for Garbage Collection

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 semantics 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.

Semantics Specification in Type Soundness. Felleisen and Wright's *syntactic approach to type soundness* [26] argues in favour of using small-step reduction semantics and evaluation contexts for type soundness proofs. This avoids some of the shortcomings inherent to 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* [6]) 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, definitional interpreters [17], 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 [25] 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 [21, 9, 18]. This makes the function expressible in the logic, and, being a total function, this provides a means of distinguishing divergence from going wrong that can be used to prove type soundness.

The modularity of the small-step approach in [26] 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 [26] 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., Felleisen and Wright, 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 the extent to which we could extend 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 [8] 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 semantics specification styles, there is a proliferation of ways to deal with name binding and

memory management in semantics 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 ([23]) 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 [5] or Jinja [8].

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 [7], Ott [20]), and those that provide support for ad hoc binding via auxiliary entities (K [19], 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 [23, 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 certain dynamically-typed languages, where proving the absence of typing errors is not a concern, but proving the absence of binding errors might be.

References

1 A. Charguéraud. Pretty-big-step semantics. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2013.

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

3 S. Drossopoulou and S. Eisenbach. Java is type safe - probably. In M. Aksit and S. Matsuoka, editors, *ECOOP 97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer, 1997.

4 E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In J. G. Morrisett and S. L. P. Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 270–282. ACM, 2006.

5 A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

6 G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

7 C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 285–296. ACM, 2012.

8 G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

9 R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ml. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014.

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

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

12 J. G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *FPCA*, pages 66–77, 1995.

13 P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009.

14 P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015.

15 B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

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

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

18 T. Rompf and N. Amin. From F to DOT: Type soundness proofs with definitional interpreters. abs/1510.05216, 2015. `http://arxiv.org/abs/1510.05216`.

19 G. Rosu and T.-F. Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

20 P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

21 J. G. Siek. Type safety in three easy lemmas, may 2013. `http://siek.blogspot.co.uk/2013/05/type-safety-in-three-easy-lemmas.html`.

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

23 D. Syme. Proving java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118. Springer, 1999.

24 H. van Antwerpen, P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*, pages 49–60, January 2016.

25 V. A. Vergu, P. Neron, and E. Visser. Dynsem: A dsl for dynamic semantics specification. In M. Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

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

A Wrong rules for L1

Values

$v ::= \ldots \mid \mathsf{Stuck}$

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

$$\frac{\forall p\ x_j^{\mathsf{D}}.\ \neg(\mathcal{G} \overset{\mathsf{G}}{\vdash} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}})}{f \vdash x_i^{\mathsf{R}}/h \Rightarrow \mathsf{Stuck}/h} \quad [\mathsf{EvVarB1}]$$

$$\frac{\mathcal{G} \overset{\mathsf{G}}{\vdash} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}} \quad \forall f'\ x_j^{\mathsf{D}}.\ \neg(h, f \models p \Rightarrow (f', x_j^{\mathsf{D}}))}{f \vdash x_i^{\mathsf{R}}/h \Rightarrow \mathsf{Stuck}/h} \quad [\mathsf{EvVarB2}]$$

$$\frac{\mathcal{G} \overset{\mathsf{G}}{\vdash} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}} \quad h, f \models p \Rightarrow (f', x_j^{\mathsf{D}}) \quad \forall v.\ \neg(f', h \models \mathsf{get}(x_j^{\mathsf{D}}) \Rightarrow v)}{f \vdash x_i^{\mathsf{R}}/h \Rightarrow \mathsf{Stuck}/h} \quad [\mathsf{EvVarB3}]$$

$$\frac{f \vdash a_1/h_1 \Rightarrow v_1/h_2 \quad v_1 \neq \mathsf{NumV}(\_)}{f \vdash a_1 \oplus a_2/h_1 \Rightarrow \mathsf{Stuck}/h_2} \quad [\mathsf{EvArithOpB1}]$$

$$\frac{f \vdash a_1/h_1 \Rightarrow z_1/h_2 \quad f \vdash a_2/h_2 \Rightarrow v_2/h_3 \quad v_2 \neq \mathsf{NumV}(\_)}{f \vdash a_1 \oplus a_2/h_1 \Rightarrow \mathsf{Stuck}/h_3} \quad [\mathsf{EvArithOpB2}]$$

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

$$\frac{f \vdash a_1/h_1 \Rightarrow v_1/h_2 \quad v_1 \neq \mathsf{ClosV}(\_, \_, \_)}{f \vdash a_1(a_2)/h_1 \Rightarrow \mathsf{Stuck}/h_2} \quad [\mathsf{EvAppB1}]$$

$$\frac{f \vdash a_1/h_1 \Rightarrow \mathsf{ClosV}(x_i^{\mathsf{D}}, a', f')/h_2 \quad f \vdash a_2/h_2 \Rightarrow \mathsf{Stuck}/h_3}{f \vdash a_1(a_2)/h_1 \Rightarrow \mathsf{Stuck}/h_3} \quad [\mathsf{EvAppB2}]$$

B Wrong rules and exception propagation in L2

We present rules for propagating exceptions, and for checking if a program gets stuck, and propagating stuckness if it arises. The rules include both additional rules for getting stuck, and rules meant to replace existing rules (Rules [EvAsgn,EvAppDef,EvApp]), since these require additional premises in order to ensure that exceptions are correctly propagated.

$Exc \ni X ::= \ldots \mid \mathsf{Stuck}$	$\mathcal{A}(\mathsf{ExcV}(X)) = \mathsf{ExcV}(X)$ $\mathcal{A}(v) = \mathsf{ExcV}(\mathsf{Stuck}) \quad \text{when } v \neq \mathsf{ExcV}(\_)$ $\mathcal{A}(u) = \mathsf{ExcV}(\mathsf{Stuck}) \quad \text{when } u \neq \mathsf{ExcV}(\_)$

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

$$\frac{f \overset{\text{LHS}}{\vdash} lhs/h_1 \Rightarrow u/h_2 \quad u \neq (\_,\_)}{f \vdash lhs/h_1 \Rightarrow \mathcal{A}(u)/h_2} \quad [\text{EvLhsB1}]$$

$$\frac{f \overset{\text{LHS}}{\vdash} lhs/h_1 \Rightarrow (f', x_i^{\text{D}})/h_2 \quad \forall v.\ \neg(f', h_2 \models \mathsf{get}(x_i^{\text{D}}) \Rightarrow v)}{f \vdash lhs/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h_2} \quad [\text{EvLhsB2}]$$

$$\frac{\mathcal{G} \overset{\text{G}}{\vdash} p : x_i^{\text{R}} \longmapsto x_j^{\text{D}} \quad \forall s.\ \neg \mathcal{G} \overset{\text{G}}{\vdash} x_j^{\text{D}} \longrightarrow\!\triangleright s}{f \vdash \textbf{new}\ x_i^{\text{R}}/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h_1} \quad [\text{EvNewB1}]$$

$$\frac{\forall p\ x_i^{\text{R}}.\ \neg(\mathcal{G} \overset{\text{G}}{\vdash} p : x_i^{\text{R}} \longmapsto x_j^{\text{D}})}{f \vdash \textbf{new}\ x_i^{\text{R}}/h \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h} \quad [\text{EvNewB2}]$$

$$\frac{f \overset{\text{LHS}}{\vdash} lhs/h_1 \Rightarrow u/h_2 \quad u \neq (\_,\_)}{f \vdash lhs\ := e/h_1 \Rightarrow \mathsf{ExcV}(X)/h_2} \quad [\text{EvAsgnB1}]$$

$$\frac{f \overset{\text{LHS}}{\vdash} lhs/h_1 \Rightarrow (f', x_i^{\text{D}})/h_2 \quad f \vdash e/h_2 \Rightarrow \mathsf{ExcV}(X)/h_3}{f \vdash lhs\ := e/h_1 \Rightarrow \mathsf{ExcV}(X)/h_3} \quad [\text{EvAsgnB2}]$$

$$\frac{\begin{array}{l} f \overset{\text{LHS}}{\vdash} lhs/h_1 \Rightarrow (f', x_i^{\text{D}})/h_2 \quad f \vdash e/h_2 \Rightarrow v/h_3 \\ \forall v\ h_4.\ \neg(f', h_3 \models \mathsf{set}(x_i^{\text{D}}, v) \Rightarrow h_4) \end{array}}{f \vdash lhs\ := e/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h_3} \quad [\text{EvAsgnB3}]$$

$$\frac{\begin{array}{l} f \overset{\text{LHS}}{\vdash} lhs/h_1 \Rightarrow (f', x_i^{\text{D}})/h_2 \quad f \vdash e/h_2 \Rightarrow v/h_3 \quad v \neq \mathsf{ExcV}(\_) \\ \forall v\ h_4.\ \neg(f', h_3 \models \mathsf{set}(x_i^{\text{D}}, v) \Rightarrow h_4) \end{array}}{f \vdash lhs\ := e/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h_3} \quad [\text{EvAsgnB3}]$$

$$\frac{\begin{array}{l} f \overset{\text{LHS}}{\vdash} lhs/h_1 \Rightarrow (f', x_i^{\text{D}})/h_2 \quad f \vdash e/h_2 \Rightarrow v/h_3 \quad v \neq \mathsf{ExcV}(\_) \\ f', h_3 \models \mathsf{set}(x_i^{\text{D}}, v) \Rightarrow h_4 \end{array}}{f \vdash lhs\ := e/h_1 \Rightarrow v/h_4} \quad [\text{EvAsgn}]$$

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

$$\frac{f \vdash a_1/h_1 \Rightarrow \mathsf{DFun}(v_1)/h_2 \quad f \vdash a_2/h_2 \Rightarrow \mathsf{ExcV}(X)/h_3}{f \vdash a_1(a_2)/h_1 \Rightarrow \mathsf{ExcV}(X)/h_3} \quad [\text{EvAppDefB}]$$

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

$$\frac{f \vdash a_1/h_1 \Rightarrow v_1/h_2 \quad v_1 \neq \mathsf{NumV}(\_)}{f \vdash a_1 \oplus a_2/h_1 \Rightarrow \mathcal{A}(v_1)/h_2} \quad [\text{EvArithOpB1}]$$

$$\frac{f \vdash a_1/h_1 \Rightarrow \mathsf{NumV}(z_1)/h_2 \quad f \vdash a_2/h_2 \Rightarrow v_2/h_3 \quad v_2 \neq \mathsf{NumV}(\_)}{f \vdash a_1 \oplus a_2/h_1 \Rightarrow \mathcal{A}(v_2)/h_3} \quad [\text{EvArithOpB2}]$$

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

$$\frac{f \vdash a_1/h_1 \Rightarrow v_1/h_2 \quad v_1 \neq \mathsf{ClosV}(\_,\_,\_)}{f \vdash a_1(a_2)/h_1 \Rightarrow \mathcal{A}(v_1)/h_2} \quad [\text{EvAppB1}]$$

$$\frac{f \vdash a_1/h_1 \Rightarrow \mathsf{ClosV}(x_i^{\mathsf{D}}, a', f')/h_2 \quad f \vdash a_2/h_2 \Rightarrow \mathsf{ExcV}(X)/h_3}{f \vdash a_1(a_2)/h_1 \Rightarrow \mathsf{ExcV}(X)/h_3} \quad [\text{EvAppB2}]$$

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

$$\frac{\forall x_j^{\mathsf{D}}.\ \neg(\mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}})}{f \vdash^{\mathrm{LHS}} x_i^{\mathsf{R}}/h \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h} \quad [\text{LhsVarB1}]$$

$$\frac{\mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}} \quad \forall f'\ x_j^{\mathsf{D}}.\ \neg(f, h \models p \Rightarrow (f', x_j^{\mathsf{D}}))}{f \vdash^{\mathrm{LHS}} x_i^{\mathsf{R}}/h \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h} \quad [\text{LhsVarB2}]$$

$$\frac{f \vdash e/h_1 \Rightarrow v/h_2 \quad v \neq \mathsf{RecordV}(\_)}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathsf{R}}/h_1 \Rightarrow \mathsf{ExcV}(v)/h_2} \quad [\text{LhsFAccB1}]$$

$$\frac{f \vdash e/h_1 \Rightarrow \mathsf{RecordV}(f_{rec})/h_2 \quad \forall s.\ \neg(\mathcal{G} \vdash^{\mathrm{G}} x_i^{\mathsf{R}} \longrightarrow s)}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathsf{R}}/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h_2} \quad [\text{LhsFAccB2}]$$

$$\frac{\begin{array}{l} f \vdash e/h_1 \Rightarrow \mathsf{RecordV}(f_{rec})/h_2 \quad \mathcal{G} \vdash^{\mathrm{G}} x_i^{\mathsf{R}} \longrightarrow s \\ \vdash^{\mathrm{F}} \mathsf{initDefault}(s, \{\mathbf{I} \mapsto \{\mathcal{S}_{h_2}(f_{rec}) \mapsto f_{rec}\}\})/h_2 \Rightarrow f'/h_3 \\ \forall x_j^{\mathsf{D}}.\ \neg(\mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}) \end{array}}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathsf{R}}/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h_3} \quad [\text{LhsFAccB3}]$$

$$\frac{\begin{array}{l} f \vdash e/h_1 \Rightarrow \mathsf{RecordV}(f_{rec})/h_2 \quad \mathcal{G} \vdash^{\mathrm{G}} x_i^{\mathsf{R}} \longrightarrow s \\ \vdash^{\mathrm{F}} \mathsf{initDefault}(s, \{\mathbf{I} \mapsto \{\mathcal{S}_{h_2 f_{rec}}(|) \to f_{rec}\}\})/h_2 \Rightarrow f'/h_3 \\ \mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}} \quad \forall f''\ x_j^{\mathsf{D}}.\ \neg(f', h_3 \models p \Rightarrow (f'', x_j^{\mathsf{D}})) \end{array}}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathsf{R}}/h_1 \Rightarrow \mathsf{ExcV}(\mathsf{Stuck})/h_3} \quad [\text{LhsFAccB4}]$$

C From Records to Classes

Section 4 showed that using scopes to describe frames also scales as a model to describe and prove the type soundness of records. In this section we consider the extension of records to *classes*.

C.1 Well-formed Terms

Figure 23 defines the extension of L1 with classes instead of records. A program in L3 consists of a set of declarations and a program expression (enclosed in a **`begin ... end`** block) that may utilize these declarations. In L3, the only kind of declaration is class definitions. A class definition consists of a declaration x_i^{D}, an optional reference to a superclass, x_i^{R}, and a sequence of fields.

A key feature of class-based inheritance is *overriding*, i.e., when an inheriting class replaces a field of a superclass by another field. L3 supports overriding by syntactically distinguishing between: (i) actual fields in classes, consisting of a declaration x_i^{D}, a type annotation and an initialization expression a; and (ii) an override, consisting of a reference to the field or method which is being overridden in a superclass, and an initialization expression a which overrides it.

Instead of record types, L3 has class types $\mathtt{Class}(x_i^{\mathsf{D}})$ that classify expressions producing objects that are instances of the class identified by x_i^{D}. Additionally, the type $\mathtt{ClassDef}(x_i^{\mathsf{D}})$ classifies values recording the AST of class definitions. These values are used to instantiate new objects of a class by iterating through the field initializers stored in the class definition and initializing the corresponding fields of the newly created object. This is reminiscent of treating classes as prototypes, and using cloning to instantiate new objects.

The role of this type is to classify the representation of the classes with initial values of their fields, such that it can be dereferenced and used to initialize new instances of the class. Source programs should not be able to dereference class definitions directly.

Expressions are the same as in L2, except for the introduction of a **`null`** expression which represents the NullV value. Since classes have obligatory initialization expressions, it is useful to have explicit **`null`** values in order to construct cyclic objects.

C.2 Well-bound Terms

The well-boundness rules for L3 first check that all declarations are well-bound, specifically that class definitions and their initializer expressions are. Subsequently, program expressions are checked. The well-boundness rules for expressions are identical to those for records (except that record types become class types).

Figure 24 defines the well-boundness of programs for L3, where the well-boundness. The figure omits well-boundness of expressions and lhs expressions, since these are identical to the rules for records (Figure 17), except for an additional trivial well-boundness rule for **`null`** (also omitted). There are two rules for class definitions: Rule [WbCD] and Rule [WbCDE]. The first of these is practically identical to the corresponding rule for records. The latter applies to classes with links to superclasses, and checks that the superclass reference actually resolves to a declaration with associated scope.

Extending a class is handled by an import edge in Rule [WbRDE]. This import edge makes fields declared in superclass(es) reachable from the inheriting classes.

Annotated syntax

$$
\begin{array}{rcl}
t & ::= & \texttt{Int} \mid t \to t \mid \texttt{Class}\,(x_i^{\mathsf{D}}) \\
& \mid & \texttt{ClassDef}\,(x_i^{\mathsf{D}}) \\
p & ::= & d^*\ \texttt{begin}\ e\ \texttt{end} \\
d & ::= & \texttt{class}\ x_i^{\mathsf{D}}\ (\texttt{extends}\ x_i^{\mathsf{R}})?\ \{\ \mathit{fld}^*\ \} \\
\mathit{fld} & ::= & x_i^{\mathsf{D}}\ \texttt{:}\ t\ \texttt{:=}\ a \mid \texttt{override}\ x_i^{\mathsf{R}}\ \texttt{:=}\ a \\
e & ::= & z \mid \mathit{lhs} \mid a \oplus a \mid \texttt{fun}\,(x_i^{\mathsf{D}}\texttt{:}\ t)\{\ a\ \} \\
& \mid & a(a) \mid \texttt{new}\ x_i^{\mathsf{R}} \mid \mathit{lhs}\ \texttt{:=}\ a \mid a\texttt{;}\ a \\
& \mid & \texttt{null} \\
\mathit{lhs} & ::= & x_i^{\mathsf{R}} \mid a\texttt{.}x_i^{\mathsf{R}} \\
a & ::= & e^{s,t}
\end{array}
$$

Subtyping

$$\frac{\mathcal{G} \vdash^{\mathsf{G}} x^{\mathsf{D}}_{sub} \longrightarrow\!\triangleright s_{sub} \quad \mathcal{G} \vdash^{\mathsf{G}} x^{\mathsf{D}}_{sup} \longrightarrow\!\triangleright s_{sup} \quad \mathcal{G} \vdash^{\mathsf{G}} s_{sub} \xrightarrow{\mathbf{I}} s_{sup}}{\texttt{Class}\,(x^{\mathsf{D}}_{sub}) < \texttt{Class}\,(x^{\mathsf{D}}_{sup})}\ [\text{SubClass}]$$

$$\frac{t_1' <: t_1 \quad t_2 <: t_2'}{t_1 \to t_2 < t_1' \to t_2'}\ [\text{SubFun}]$$

$$t <: t\ [\text{SubRefl}]$$

$$\frac{t_1 < t_2 \quad t_2 <: t_3}{t_1 <: t_3}\ [\text{SubTrans}]$$

Figure 23 Annotated syntax of L3 with distinguished references and declarations

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

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

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

$$\frac{\begin{array}{l}\mathcal{G} \vdash^{\mathsf{G}} s \longrightarrow x_i^{\mathsf{D}} \quad \mathcal{G} \vdash^{\mathsf{G}} x_i^{\mathsf{D}} \longrightarrow\!\triangleright s' \\ \mathcal{G} \vdash^{\mathsf{G}} s' \xrightarrow{\mathbf{P}} s \quad (\forall \mathit{fld} \in \mathit{flds}.\ \vdash^{\mathsf{B}}_{\mathsf{F}} \mathit{fld}^{s'}) \\ \mathcal{K}(s') = \emptyset\end{array}}{\vdash^{\mathsf{B}}_{\mathsf{D}} (\texttt{class}\ x_i^{\mathsf{D}}\ \{\ \mathit{flds}\ \})^s}\ [\text{WbCD}]$$

$$\frac{\begin{array}{l}\mathcal{G} \vdash^{\mathsf{G}} s \longrightarrow x_i^{\mathsf{D}} \quad \mathcal{G} \vdash^{\mathsf{G}} x_i^{\mathsf{D}} \longrightarrow\!\triangleright s' \\ \mathcal{G} \vdash^{\mathsf{G}} s' \xrightarrow{\mathbf{P}} s \quad (\forall \mathit{fld} \in \mathit{flds}.\ \vdash^{\mathsf{B}}_{\mathsf{F}} \mathit{fld}^{s'}) \\ \mathcal{G} \vdash^{\mathsf{G}} p : x_j^{\mathsf{R}} \longmapsto x_k^{\mathsf{D}} \quad \mathcal{G} \vdash^{\mathsf{G}} x_k^{\mathsf{D}} \longrightarrow\!\triangleright s'' \\ \mathcal{G} \vdash^{\mathsf{G}} x_j^{\mathsf{R}} \longrightarrow s \quad \mathcal{G} \vdash^{\mathsf{G}} s' \xrightarrow{\mathbf{I}} s''\end{array}}{\vdash^{\mathsf{B}}_{\mathsf{D}} (\texttt{class}\ x_i^{\mathsf{D}}\ \texttt{extends}\ x_j^{\mathsf{R}}\ \{\ \mathit{flds}\ \})^s}\ [\text{WbCDE}]$$

Well-bound fields $\boxed{\vdash^{\mathsf{B}}_{\mathsf{F}} \mathit{fld}^s}$

$$\frac{\vdash^{\mathsf{B}} e^s \quad \mathcal{G} \vdash^{\mathsf{G}} s \longrightarrow x_i^{\mathsf{D}}}{\vdash^{\mathsf{B}}_{\mathsf{F}} (x_i^{\mathsf{D}}\ \texttt{:}\ t\ \texttt{:=}\ e^s)^s}\ [\text{WbFD}]$$

$$\frac{\begin{array}{l}\mathcal{G} \vdash^{\mathsf{G}} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}} \\ \mathcal{G} \vdash^{\mathsf{G}} x_i^{\mathsf{R}} \longrightarrow s \quad \vdash^{\mathsf{B}} e^s\end{array}}{\vdash^{\mathsf{B}}_{\mathsf{F}} (\texttt{override}\ x_i^{\mathsf{R}}\ \texttt{:=}\ e^s)^s}\ [\text{WbFO}]$$

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

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

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

$$\frac{\begin{array}{l}(\forall \mathit{fld} \in \mathit{flds}.\ \vdash^{\mathsf{T}}_{\mathsf{F}} \mathit{fld}) \\ \mathcal{G} \vdash^{\mathsf{G}} x_i^{\mathsf{D}} : \texttt{ClassDef}\,(x_i^{\mathsf{D}})\end{array}}{\vdash^{\mathsf{T}}_{\mathsf{D}} \texttt{class}\ x_i^{\mathsf{D}}\ \{\ \mathit{flds}\ \}}\ [\text{WtCD}]$$

$$\frac{\begin{array}{l}(\forall \mathit{fld} \in \mathit{flds}.\ \vdash^{\mathsf{T}}_{\mathsf{F}} \mathit{fld}) \\ \mathcal{G} \vdash^{\mathsf{G}} x_i^{\mathsf{D}} : \texttt{ClassDef}\,(x_i^{\mathsf{D}}) \\ \mathcal{G} \vdash^{\mathsf{G}} p : x_j^{\mathsf{R}} \longmapsto x_k^{\mathsf{D}} \\ \mathcal{G} \vdash^{\mathsf{G}} x_k^{\mathsf{D}} : \texttt{ClassDef}\,(x_k^{\mathsf{D}})\end{array}}{\vdash^{\mathsf{T}}_{\mathsf{D}} \texttt{class}\ x_i^{\mathsf{D}}\ \texttt{extends}\ x_j^{\mathsf{R}}\ \{\ \mathit{flds}\ \}}\ [\text{WtCDE}]$$

Well-typed fields $\boxed{\vdash^{\mathsf{T}}_{\mathsf{F}} \mathit{fld}}$

$$\frac{\mathcal{G} \vdash^{\mathsf{G}} x_i^{\mathsf{D}} : t \quad \vdash^{\mathsf{T}} e^{t'} \quad t' <: t}{\vdash^{\mathsf{T}}_{\mathsf{F}} x_i^{\mathsf{D}}\ \texttt{:}\ t\ \texttt{:=}\ e^{t'}}\ [\text{WtFD}]$$

$$\frac{\begin{array}{l}\mathcal{G} \vdash^{\mathsf{G}} p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}} \quad \mathcal{G} \vdash^{\mathsf{G}} x_j^{\mathsf{D}} : t \\ \vdash^{\mathsf{T}} e^{t'} \quad t' <: t\end{array}}{\vdash^{\mathsf{T}}_{\mathsf{F}} \texttt{override}\ x_i^{\mathsf{R}}\ \texttt{:=}\ e^{t'}}\ [\text{WtFO}]$$

Figure 24 Well-boundness and well-typedness in L3

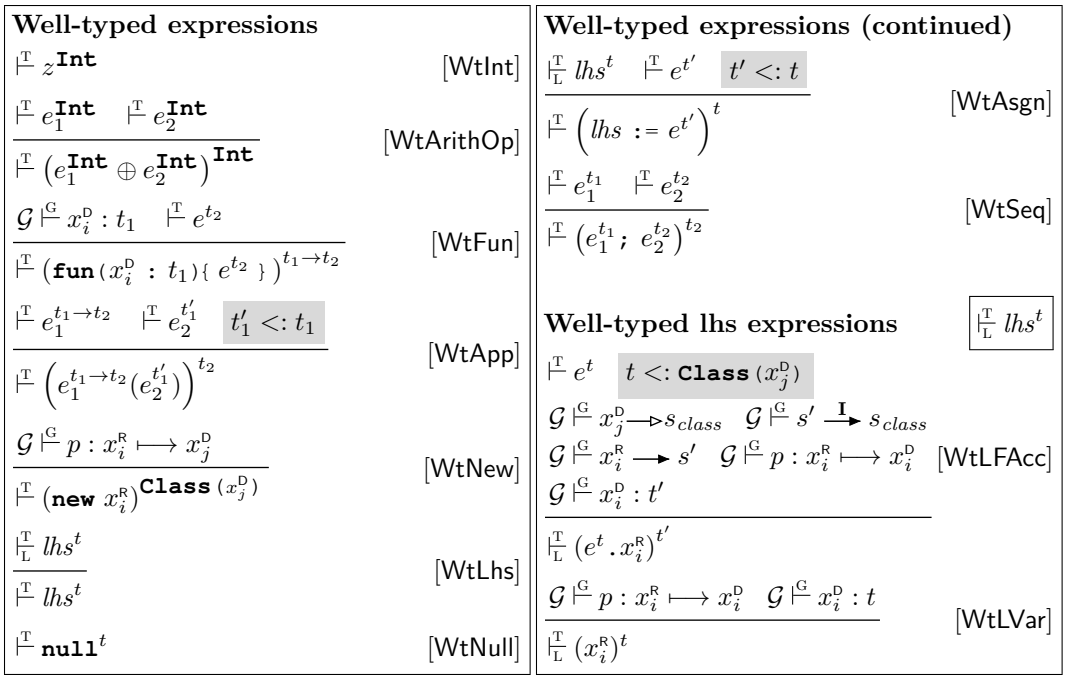

Figure 25 Well-typed expressions in L3

C.3 Well-typed Terms

Another key feature of L3 is that it supports subtyping. Figure 23 defines the notion of subtyping for the language: Rule [SubClass] says that **Class**(x_i^{D}) is a subtype of **Class**(x_j^{D}) if there is an import link from the associated scope of x_i^{D} to the associated scope of x_j^{D}. Because of the way we represent scopes of classes, this notion of subtyping corresponds exactly to the notion that one would expect. We use $<$ for direct subtyping, and $<:$ for the reflexive-transitive closure of $<$. Rule [SubFun] is the usual [15] subtyping rule for function types.

The rules in Figure 24 use so-called *algorithmic* subtyping; e.g., Figure 24 uses subsumption premises in typing rules instead of a general subsumption rule like:

$$\frac{\vdash^{\mathrm{T}} e^{t'} \quad t <: t'}{\vdash^{\mathrm{T}} e^{t}} \qquad \text{[Sub]}$$

Rule [WtCD] and Rule [WtCDE] mark the only uses of the **ClassDef** type: the rules check that declarations x_i^{D} that identify class types are associated with class definition values, i.e., values of type **ClassDef**(x_i^{D}), at runtime. Corresponding frame slots at runtime will thus contain the class definition to be used for initializing new instances of the class.

C.4 Dynamic Semantics

Figure 27 defines values and exceptions for L3. A class definition value $\mathsf{ClassDefV}(x_i^{\mathrm{D}}, \mathit{optR}, \mathit{flds}, f)$ records: the declaration of the class definition, x_i^{D}; an optional reference optR to a superclass definition; a list of fields flds; and the lexical scope frame f in which the class is defined, and in which class field initializers should be evaluated. Class definitions are stored in slots and are used for initializing new objects.

There are several ways to implement and model objects with inheritance at runtime. Following the frames-as-scopes methodology, the dynamic semantics in Figure 28 sticks closely to how the scopes of classes are modelled in the well-boundness and well-typedness rules from Figures wbwt-classfunplus and 25. There, the scopes of classes are organized in a nested, hierarchical fashion, where subclass inheritance is modelled by import links between scopes. We thus let an object be given by a hierarchy of frames: a frame instance with a link to its superclass frame instance (if any); and similarly for the superclass frame. Crucial is that frames in the hierarchy of an object are not shared; each object instance has distinct superclass frames.

Consider, for example, the program in Figure 26 and its scope graph. Here, the scope of class `B` has an import edge to the scope of `A`. An object frame instantiating `B` thus has a similarly hierarchical structure.

Rule [EvNew] relies on an auxiliary dedicated relation for object initialization, using the judgment $f \vdash^{\mathrm{O}} x_i^{\mathrm{R}}/h \Rightarrow f'/h$, which takes a reference to a class definition. If a class definition inherits from a superclass (Rule [OIP]), the $\vdash^{\mathrm{O}}$ judgment recursively instantiates and initializes the superclass frame. Once the superclass frame has been created, Rule [OIP] creates the class frame, and initializes its fields by means of the class definition.

The judgment $f', f' \vdash^{\mathrm{IR}} \mathit{flds}/h_4 \Rightarrow h_5$ is used in Rule [OIP] and Rule [OI] to iterate through the fields of a class definition, and executing the initialization expressions, using Rules [IRCD, IRCR, IRNil]. Following the well-boundness rules for field declarations (Rules [WbFD, WbFO] in Figure 24), initialization expressions for fields are scoped by the class scope. Consequently, initialization expressions may refer to other fields in the class

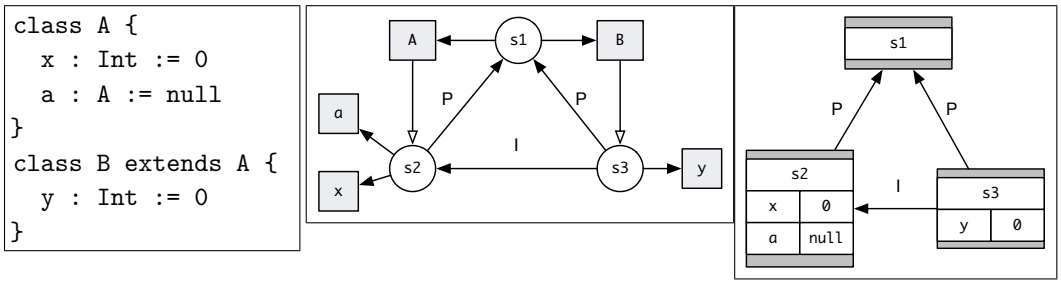

Figure 26 An example program, its scope graph, and its object frame structure

Values and exceptions

$$
\begin{aligned}
\mathit{Val} \ni v ::= {} & \mathsf{NumV}(z) \mid \mathsf{ClosV}(x_i^{\mathrm{D}}, a, f) \\
\mid {} & \mathsf{ObjectV}(f) \mid \mathsf{NullV} \\
\mid {} & \mathsf{ExcV}(X) \mid \mathsf{DFun}(v) \\
\mid {} & \mathsf{ClassDefV}(x_i^{\mathrm{D}}, x_i^{\mathrm{R}}?, \mathit{fld}^*, f) \\
\mathit{Exc} \ni X ::= {} & \mathsf{NullPointer}
\end{aligned}
$$

Runtime casts $h \vdash^{\mathrm{C}} \mathsf{upcast}(f, s) \Rightarrow f'$

$$\frac{\mathcal{S}_h(f) = s}{h \vdash^{\mathrm{C}} \mathsf{upcast}(f, s) \Rightarrow f} \ [\mathsf{CRefl}]$$

$$\frac{s' \in \mathit{Dom}(\mathcal{K}_h(f)(\mathbf{I})) \quad h \vdash^{\mathrm{C}} \mathsf{upcast}(\mathcal{K}_h(f)(\mathbf{I})(s'), s) \Rightarrow f''}{h \vdash^{\mathrm{C}} \mathsf{upcast}(f, s) \Rightarrow f''} \ [\mathsf{CUp}]$$

Figure 27 Values and exceptions for L3 (left), and runtime cast semantics by traversing frame links (right)

which may not have been initialized at the point where they are accessed. The semantics of field initialization in 29 does not specify the order in which fields should be initialized: a field initializer is non-deterministically selected by Rules [IRCD,IRCR]. If a field is accessed which is not initialized, the value for that field is the default value, since Rules [OIP,OI] use initDefault to initialize object frames.

Since objects are represented as hierarchies of frames, lhs expressions for field access now rely on paths that traverse the structure of these frames. For example, consider the following program expression, where `A` and `B` refer to the class definitions of `A` and `B` in Figure 26:

```
begin
  (fun (a : A) { a.x })(new B)
end
```

The path of the field access expression $a.x_i^{\mathrm{R}}$ is calculated based on `a` being an object of type `A`; i.e., $\mathbf{E}(\mathbf{I}, s_{imp}) \cdot \mathbf{D}(\mathtt{x})$, where s_{imp} is the import scope for the field access expression. 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 a `x` slot via its import link. The actual path to the expression is thus: $\mathbf{E}(\mathbf{I}, s_{imp}) \cdot \mathbf{E}(\mathbf{I}, s_{\mathtt{A}}) \cdot \mathbf{D}(\mathtt{x})$.

In order to deal with this mismatch between the statically computed path and the actual path, the semantics of field access in Figure 28 upcasts an object frame before constructing the import frame used for accessing the field in Rule [LhsFAcc]. Upcasting is defined in Figure 27, where the judgment $h \vdash^{\mathrm{C}} \mathsf{upcast}(f, s) \Rightarrow f'$ asserts that a frame f can be cast to a frame that instantiates s, by following the import links of f to arrive at a frame f' in heap h such that $\mathcal{S}_h(f') = s$.

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

$$\frac{\vdash^{\mathrm{F}} \mathsf{initDefault}(s, \emptyset)/\emptyset \Rightarrow f_{root}/h_1 \quad f_{root} \vdash^{\mathrm{DS}} ds/h_1 \Rightarrow h_2 \quad f_{root} \vdash e/h_2 \Rightarrow v/h_3}{\vdash^{\mathrm{P}} ds\ \texttt{begin}\ e^s\ \texttt{end} \Rightarrow v/h'} \quad [\mathsf{EvProg}]$$

Class definition initialization $\boxed{f \vdash^{\mathrm{DS}} ds/h \Rightarrow h'}$

$$\frac{d \in ds \quad d = (\texttt{class}\ x_i^{\mathrm{D}}\ \{\ \mathit{flds}\ \}) \quad f, h_1 \models \mathsf{set}(x_i^{\mathrm{D}}, \mathsf{ClassDefV}(x_i^{\mathrm{D}}, \mathsf{None}, \mathit{flds}, f)) \Rightarrow h_2 \quad f \vdash^{\mathrm{DS}} (ds - \{d\})/h_2 \Rightarrow h_3}{f \vdash^{\mathrm{DS}} ds/h_1 \Rightarrow h_3} \quad [\mathsf{EvCD}]$$

$$\frac{d \in ds \quad d = (\texttt{class}\ x_i^{\mathrm{D}}\ \texttt{extends}\ x_j^{\mathrm{R}}\ \{\ \mathit{flds}\ \}) \quad f, h_1 \models \mathsf{set}(x_i^{\mathrm{D}}, \mathsf{ClassDefV}(x_i^{\mathrm{D}}, \mathsf{Some}(x_j^{\mathrm{R}}), \mathit{flds}, f)) \Rightarrow h_2 \quad f \vdash^{\mathrm{DS}} (ds - \{d\})/h_2 \Rightarrow h_3}{f \vdash^{\mathrm{DS}} ds/h_1 \Rightarrow h_4} \quad [\mathsf{EvCDE}]$$

$$\frac{}{f \vdash^{\mathrm{DS}} \emptyset/h \Rightarrow h} \quad [\mathsf{EvDNil}]$$

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

$$\frac{f \vdash^{\mathrm{O}} x_i^{\mathrm{R}}/h_1 \Rightarrow f'/h_2}{f \vdash \texttt{new}\ x_i^{\mathrm{R}}/h_1 \Rightarrow \mathsf{ObjectV}(f')/h_4} \quad [\mathsf{EvNew}]$$

Lhs expression evaluation $\boxed{f \vdash^{\mathrm{LHS}} lhs/h \Rightarrow (f', x_i^{\mathrm{D}})/h'}$

$$\frac{f \vdash e/h_1 \Rightarrow \mathsf{ObjectV}(f_{obj})/h_2 \quad \mathcal{G} \vdash^{\mathrm{G}} s \longrightarrow x_i^{\mathrm{R}} \quad \mathcal{G} \vdash^{\mathrm{G}} s \xrightarrow{\mathbf{I}} s_{obj} \quad h_2 \vdash^{\mathrm{C}} \mathsf{upcast}(f_{obj}, s_{obj}) \Rightarrow f'_{obj} \quad \vdash^{\mathrm{F}} \mathsf{initDefault}(s, \{\mathbf{I} \mapsto \{\mathcal{S}_{h_2}(f'_{obj}) \mapsto f'_{obj}\}\})/h_2 \Rightarrow f'/h_3 \quad \mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_j^{\mathrm{D}} \quad f', h \models p \Rightarrow (f'', x_j^{\mathrm{D}})}{f \vdash^{\mathrm{LHS}} e.x_i^{\mathrm{R}}/h_1 \Rightarrow (f'', x_j^{\mathrm{D}})/h_3} \quad [\mathsf{LhsFAcc}]$$

Object initialization $\boxed{f \vdash^{\mathrm{O}} x_i^{\mathrm{R}}/h \Rightarrow f'/h'}$

$$\frac{f \vdash^{\mathrm{LHS}} x_i^{\mathrm{R}}/h_1 \Rightarrow (f', x_j^{\mathrm{D}})/h_2 \quad (\mathcal{D}_{h_2}(f'))(x_j^{\mathrm{D}}) = \mathsf{ClassDefV}(x_j^{\mathrm{D}}, \mathsf{Some}(x_k^{\mathrm{R}}), \mathit{flds}, f_{par}) \quad f \vdash^{\mathrm{O}} x_k^{\mathrm{R}}/h_2 \Rightarrow f_{sup}/h_3 \quad \mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_l^{\mathrm{D}} \quad \mathcal{G} \vdash^{\mathrm{G}} x_l^{\mathrm{D}} \longrightarrow\!\triangleright s' \quad \vdash^{\mathrm{F}} \mathsf{initDefault}(s', \{\mathbf{I} \mapsto \{\mathcal{S}_{h_3}(f_{sup}) \mapsto f_{sup}\}, \mathbf{P} \mapsto \{\mathcal{S}_{h_3}(f_{par}) \mapsto f_{par}\}\})/h_3 \Rightarrow f'/h_4 \quad f', f' \vdash^{\mathrm{IR}} \mathit{flds}/h_4 \Rightarrow h_5}{f \vdash^{\mathrm{O}} x_i^{\mathrm{R}}/h_1 \Rightarrow f'/h_5} \quad [\mathsf{OIP}]$$

$$\frac{f \vdash^{\mathrm{LHS}} x_i^{\mathrm{R}}/h_1 \Rightarrow (f', x_j^{\mathrm{D}})/h_2 \quad (\mathcal{D}_{h_2}(f'))(x_j^{\mathrm{D}}) = \mathsf{ClassDefV}(x_j^{\mathrm{D}}, \mathsf{None}, \mathit{flds}, f_{par}) \quad \mathcal{G} \vdash^{\mathrm{G}} p : x_i^{\mathrm{R}} \longmapsto x_l^{\mathrm{D}} \quad \mathcal{G} \vdash^{\mathrm{G}} x_l^{\mathrm{D}} \longrightarrow\!\triangleright s' \quad \vdash^{\mathrm{F}} \mathsf{initDefault}(s', \{\mathbf{P} \mapsto \{\mathcal{S}_{h_2}(f_{par}) \mapsto f_{par}\}\})/h_2 \Rightarrow f'/h_3 \quad f', f' \vdash^{\mathrm{IR}} \mathit{flds}/h_3 \Rightarrow h_4}{f \vdash^{\mathrm{O}} x_i^{\mathrm{R}}/h_1 \Rightarrow f'/h_4} \quad [\mathsf{OI}]$$

Figure 28 Dynamic semantics for L3(continued in Figure 29)

Field initialization $f, f' \vdash^{\text{IR}} \mathit{flds}/h \Rightarrow h'$

$$\frac{\mathit{fld} \in \mathit{flds} \quad \mathit{fld} = (x_i^{\mathsf{D}} \ \texttt{:}\ t \ \texttt{:=}\ a) \quad f_{ev} \vdash a/h_1 \Rightarrow v/h_2 \quad f_{tgt}, h_2 \models \mathsf{set}(x_j^{\mathsf{D}}, v) \Rightarrow h_3 \quad f_{tgt}, f_{ev} \vdash^{\text{IR}} (\mathit{flds} - \{\mathit{fld}\})/h_3 \Rightarrow h_4}{f_{tgt}, f_{ev} \vdash^{\text{IR}} \mathit{flds}/h_1 \Rightarrow h_4} \ [\mathsf{IRCD}]$$

$$\frac{\mathit{fld} \in \mathit{flds} \quad \mathit{fld} = (\texttt{override}\ x_i^{\mathsf{R}} \ \texttt{:=}\ a) \quad f_{tgt} \vdash^{\text{LHS}} x_i^{\mathsf{R}}/h_1 \Rightarrow (f', x_j^{\mathsf{D}})/h_2 \quad f_{ev} \vdash a/h_2 \Rightarrow v/h_3 \quad f', h_3 \models \mathsf{set}(x_j^{\mathsf{D}}, v) \Rightarrow h_4 \quad f_{tgt}, f_{ev} \vdash^{\text{IR}} (\mathit{flds} - \{\mathit{fld}\})/h_4 \Rightarrow h_5}{f_{tgt}, f_{ev} \vdash^{\text{IR}} \mathit{flds}/h_1 \Rightarrow h_5} \ [\mathsf{IRCR}]$$

$$f_{tgt}, f_{ev} \vdash^{\text{IR}} \emptyset/h \Rightarrow h \quad [\mathsf{IRNil}]$$

Figure 29 Field initialization semantics for L3

Good frame

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

$$\mathsf{wellTyped}(h, f) \triangleq$$
$$\forall t\ x_i^{\mathrm{D}} \in \mathcal{D}(\mathcal{S}_h(f)).\ \mathcal{G} \overset{\mathrm{G}}{\vdash} x_i^{\mathrm{D}} : t \implies \forall v.\ \mathcal{D}_h(f)(x_i^{\mathrm{D}}) = v \implies$$
$$\exists t'.\ h \overset{\mathrm{V}}{\vdash} v : t' \ \wedge\ t' <: t$$

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

$$\frac{\mathcal{G} \overset{\mathrm{G}}{\vdash} x_i^{\mathrm{D}} \longrightarrow\!\triangleright \mathcal{S}_h(f)}{h \overset{\mathrm{V}}{\vdash} \mathsf{ObjectV}(f) : \texttt{Class}\,(x_i^{\mathrm{D}})}\ [\mathsf{VtObj}]$$

$$\frac{\overset{\mathrm{B}}{\underset{\mathrm{D}}{\vdash}}\ (\texttt{class}\ x_i^{\mathrm{D}}\ \texttt{extends}\ x_i^{\mathrm{R}}\ \{\ \mathit{flds}\ \})^{\mathcal{S}_h(f)} \quad \overset{\mathrm{T}}{\underset{\mathrm{D}}{\vdash}}\ \texttt{class}\ x_i^{\mathrm{D}}\ \texttt{extends}\ x_i^{\mathrm{R}}\ \{\ \mathit{flds}\ \}}{h \overset{\mathrm{V}}{\vdash} \mathsf{ClassDefV}(x_i^{\mathrm{D}}, \mathsf{Some}(x_j^{\mathrm{R}}), \mathit{flds}, f) : \texttt{ClassDef}\,(x_i^{\mathrm{D}})}\ [\mathsf{VtCDE}]$$

$$\frac{\overset{\mathrm{B}}{\underset{\mathrm{D}}{\vdash}}\ (\texttt{class}\ x_i^{\mathrm{D}}\ \{\ \mathit{flds}\ \})^{\mathcal{S}_h(f)} \quad \overset{\mathrm{T}}{\underset{\mathrm{D}}{\vdash}}\ \texttt{class}\ x_i^{\mathrm{D}}\ \{\ \mathit{flds}\ \}}{h \overset{\mathrm{V}}{\vdash} \mathsf{ClassDefV}(x_i^{\mathrm{D}}, \mathsf{None}, \mathit{flds}, f) : \texttt{ClassDef}\,(x_i^{\mathrm{D}})}\ [\mathsf{VtCD}]$$

Figure 30 Well-typed frames with subtyping, and value typing for class definitions

C.5 Type Soundness

Modelling objects and classes as frames scales to specify dynamic semantics for classes such that there is a close relationship between the static scopes and dynamic frames. Similarly to previous sections, this allows us to reuse the binding and typing invariants for heaps and frames to prove type soundness, with only minor extensions to support subtyping.

Figure 30 summarizes the only change required for the frame invariant: namely that slots may contain values of a subtype of the declaration type. The figure also recalls value typing rules for objects and class definitions. The value typing of objects (Rule [VtObj]) is essentially the same as the value typing of records from previous section. The value typing rules (Rules [VtCDE,VtCD]) for class definitions use the binding and typing rules (Rules [WbCDE,WbCD,WtCDE,WtCD]) for class definitions from Figure 24.

Theorem 3 proves subtype preservation for L3.

▶ **Theorem 3** (Subtype preservation for L3)**.**

$$\forall f\ s\ t\ e\ h_1\ h_2\ v.\ f \vdash e^{s,t}/h_1 \Rightarrow v/h_2 \implies \mathsf{goodHeap}(h_1) \implies$$
$$f \in \mathit{Dom}(h_1) \implies \mathcal{S}_{h_1}(f) = s \implies \overset{\mathrm{B}}{\vdash} e^s \implies \overset{\mathrm{T}}{\vdash} e^t \implies$$
$$\mathsf{goodHeap}(h_2) \ \wedge\ \exists t'.\ h_2 \overset{\mathrm{V}}{\vdash} v : t' \ \wedge\ t' <: t$$

In order to extend the preservation proof in Theorem 3 to type soundness, we need the following upcast lemma (Lemma 4) proving that upcasting an object frame from a sub-class to a super-class is guaranteed to succeed.

▶ Lemma 4 (Upcast lemma).

$$\forall x_i^{\mathrm{D}}\ x_j^{\mathrm{D}}\ h\ s_{obj}\ f.\ \texttt{Class}\,(x_i^{\mathrm{D}}) <: \texttt{Class}\,(x_j^{\mathrm{D}}) \implies \mathsf{goodHeap}(h) \implies$$
$$\mathcal{G} \overset{\mathrm{G}}{\vdash} x_j^{\mathrm{D}} \longrightarrow\!\triangleright s_{obj} \implies h \overset{\mathrm{V}}{\vdash} \mathsf{ObjectV}(f) : \texttt{Class}\,(x_i^{\mathrm{D}}) \implies$$
$$\exists f'.\ h \overset{\mathrm{C}}{\vdash} \mathsf{upcast}(s_{obj}, f) \Rightarrow f'$$

C.6 More Class Features and Design Alternatives

In this section we have shown that using scopes to describe frames scales to describe a class-based language with inheritance and upcasting, and proving its type soundness. There are several other features that we might have added to the language, and alternative designs that one might explore in future work.

Methods and late binding of `this` or `self`. Late binding of `this` or `self` is straightforward to add to L3 by representing methods as special fields of function type which expect their first argument to be an instance of the class being defined. For example, the following class definition:

```
class A {
  method m(x : Int) { this.f(0) }
  method f(y : Int) { 42 }
}
```

could desugar into the following L3 class definition:

```
class A {
  m : A -> Int -> Int :=
    fun (this : A) { fun (x : Int) { this.f(this)(0) } }
  f : A -> Int -> Int :=
    fun (this : A) { fun (x : Int) { 42 } }
}
```

Unfolding the inheritance hierarchy of frames. The memory layout for frames could also be represented more efficiently, by representing objects as a *single* frame, for example by unfolding or merging scopes and frames. This would shorten paths and save some of what is likely to be inefficient traversal of frames in the heap when accessing fields of objects. The focus of this paper is on showing how scopes describing frames scales as a language independent principle and basis for type soundness, not on efficiency (even though we expect the semantics to be implementable in a reasonably efficient way). We leave a further investigation of such optimizations to future work.