Master of Science Thesis

Modular Analysis of Concurrent Pointer Programs

submitted by

Jens Katelaan

July 29, 2015

Chair for Software Modeling and Verification
RWTH Aachen University

Supervisors:
Prof. Dr. Thomas Noll
Prof. Dr. Ir. Joost-Pieter Katoen

Abstract

Programs with shared-memory concurrency are inherently difficult to get right: They are prone to all the memory-related errors that are familiar from the single-threaded setting, such as null pointer dereferences and unintended aliasing. In addition, the possible interference between parallel execution threads gives rise to new classes of errors, such as data races. As thread interleaving is nondeterministic in nature and heap-manipulating programs generally have an unbounded state space due to dynamic memory allocation, the application of formal methods is challenging in this setting.

In this thesis I develop a static analysis for proving shape invariants, absence of null pointer dereferences, as well as data-race freedom of programs with fork–join parallelism. To this end, I develop a formal semantics based on hypergraphs and access permissions for a model programming language with fork–join parallelism. I derive an abstract interpretation that uses hyperedge replacement grammars to safely approximate the program's semantics and discuss how to implement this interpretation in a framework for data-flow analysis to automatically generate procedure contracts. The result is a modular analysis for proving the above properties in the presence of recursive data structures and dynamic (possibly recursive) thread creation.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Jens Katelaan
Aachen, den 29.07.2015

Contents

CHAPTER 1

Introduction

1.1 Motivation and Related Work

Writing correct software is a difficult task. So difficult, in fact, that even delivered software products average 15 to 50 errors per 1000 lines of code [McC04]—despite prevalence of systematic testing and advanced software engineering methodologies. This is not surprising, as even the most careful system design leaves room for errors and even the most experienced tester may forget to cover a corner case. For this reason, formal methods are increasingly used to prove a program's properties with mathematical certainty, rather than relying on inherently incomplete mechanisms such as testing. In formal methods, we use techniques from mathematics and logic to specify and reason about systems—an approach which has long been advocated, especially for **critical systems** [Rus95]. Formal methods range from completely manual proofs over partly automated methods such as interactive theorem proving to fully automated methods such as model checking. In this thesis I shall focus on methods from the area of static program analysis. In static program analysis, we employ fully automatic, source–code-guided analysis techniques (hence **static**). More concretely, I develop an analysis for the **abstract interpretation** [CC77] of heap-manipulating programs in the presence of fork–join parallelism.

A **heap-manipulating program** is any program that allocates memory on the heap rather than exclusively operating on a stack. Every program that uses dynamic data structures or object orientation, or otherwise explicitly or implicitly deals with pointers, falls under this umbrella. In other words: Nearly every non-trivial program written in any modern programming language is a heap-manipulating program.

Pointers are difficult to reason about; as a result, the incorrect use of pointers is one of the most common sources of errors [FGL96]. **Null dereferences**, **memory leaks**, and **unintended aliasing** are all errors that most software developers are familiar with. In addition to such low-level mistakes, there are also more complex functional properties to consider. A programmer may, for example, develop a function under the assumption that it receives a noncyclic list as input or that it receives two input lists that are disjoint. If these implicit **shape invariants** are violated, the program may behave in unexpected ways [SRW02].

Recent years have brought further complications into the programming mainstream: With the advent of many-core systems as well as the cloud, **parallel, concurrent, and distributed programming** have become ubiquitous. Even though research in these areas has been carried out for fifty years—Dijkstra's seminal paper on cooperating sequential processes [Dij68] was written in 1965—we are arguably still not in a position to claim that we have definite answers to questions such as "How do I best structure a parallel program?" and "How can I ensure that it is safe to run my program in parallel?".

Many (imperative) programming languages implement a variant of the **fork–join model** of parallel computation. In this model, an execution thread, the parent thread **forks** off one or more children that may potentially run in parallel with the parent thread. When the parent thread needs to access the results of the child's computation, it performs a **join** operation, which causes the parent thread to block until the computation of the forked child has completed. In the classical model, the forked threads are executed completely independently from the parent thread and from each other; the only point of interaction is the join. While this model goes back as far as 1963 [Con63], it continues to be an important principle for structuring parallel programs: Both Java's threading model [Gos00] and C's POSIX threads [IEEE95] are (extended) implementations of the fork–join model, and fork–join parallelism continues to be one of the design patterns presented in text books on concurrent and parallel programming [DGT93; Lea00].

Concurrency gives rise to new classes of bugs, which are often extremely hard to catch, because they depend on the interleaving of the execution threads. The canonical example (formulated in a language with parallel composition) usually runs along the following lines:

$$(y := x + 1; x := y) \parallel (z := x + 1; x := z)$$

This program runs $(y := x + 1; x := y)$ and $(z := x + 1; x := z)$ in parallel, denoted by the parallel composition operator $\parallel$. Each sequential computational increments the value of a shared variable x. The result may, however, be either $x + 1$ or $x + 2$, depending on the interleaving of the threads' statements.

This kind of—usually unintended—ambiguity is called a **data race**. Unfortunately, most flavors of shared-memory concurrency, including fork–join parallelism, are prone to data races [DGT93; Pac11].

In this thesis I study an imperative programming language with explicit pointers, fork–join parallelism and shared memory concurrency. This small language, which I simply call PL, is simpler than Java or C, but is at the same time complex enough to be prone to all the memory errors I discussed so far. As such, it was specifically designed as a study object to test the feasibility of a novel static analysis for formally proving the absence of memory errors, from simple ones such as null dereferences to violations of shape invariants to data races in the concurrent setting.

My work follows in a long research tradition: In the verification of parallel programs, one early work which comes to mind is the Owicki-Gries method for axiomatically verifying parallel programs, which was published in 1976 [OG76]. For an overview of the field up until 2001, see de Roever's book [Roe01]; for a more recent perspective, the work of Jones et al. [Jon12; JHC13] is a good starting point.

In the area of heap analysis, **shape analysis** [SRW02] and **separation logic** [Rey02] are the most widespread approaches.[1] Heap-manipulating programs generally have infinite state space, as there is no upper bound for the size of the heaps they must be able to deal with (consider, for example, linked lists of arbitrary length). We therefore need an **abstraction**

[1] These are, however, not the only approaches by any means. Consider, for example, (implicit) dynamic frames [Kas11], region logic [BNR08], and forest automata [Hab+12].

mechanism to reason about such programs. In shape analysis, shape graphs serve this purpose. In separation logic, recursive predicates are used for abstracting recursive data structures such as lists. Separation logic is known for the separating conjunction, commonly denoted $*$, which expresses that a heap consists of two disjoint parts. This allows to reason locally about a statement's or procedure's effect without reasoning about the global heap, which is the key innovation of and reason for the popularity of separation logic [Rey02].

We use a different (but related [Dod08]) approach: We represent heaps as **hypergraphs**, i.e., as generalized graphs where edges may connect arbitrarily many rather than exactly two nodes. We use such generalized edges (hyperedges) to represent abstract data structures. The abstraction and concretization mechanisms are provided through **hyperedge replacement grammars**, a variant of context-free graph grammars. This approach shares with separation logic the possibility to reason locally about those parts of the heap that are relevant for the execution of a procedure; it has successfully been employed in the analysis of sequential heap-manipulating programs [Dod09; Rie09; HBJ12; Hei+15].

In this thesis I adapt the hypergraph model to parallel programs to be able to reason about noninterference of parallel processes in addition to sequential memory errors. To this end, I use permission accounting, based on the work by Boyland [Boy03] and Heule et al. [Heu+11]. This approach was inspired by the corresponding extension of separation logic [Bor+05; OHe07], and the application thereof to `Java` programs [HHH11].

The starting point for this thesis was the paper "Generating Abstract Graph-Based Procedure Summaries for Pointer Programs" by Jansen et al. [JN14]. This paper instantiates an interprocedural analysis framework by Knoop and Steffen [KS92] with a hypergraph-based domain to automatically prove the absence of certain types of memory errors.

I go beyond this work in two ways. First, I develop a formal operational semantics based on hypergraphs and an abstract interpretation of that semantics based on hyperedge replacement grammars. I use this abstract interpretation as a basis for the interprocedural analysis, thereby closing some gaps in the formalization, as the paper lacked a complete definition of the semantics. Second, I develop a semantics and an abstract interpretation for the fork–join scenario and extend the interprocedural analysis to this setting.

The resulting analysis has several desirable properties: It is sound, it is completely automatic[1], it enables local reasoning about procedures and threads, it is modular in the sense that procedures and threads can be analyzed independently, and it is decidable for large classes of programs.

Work on the verification of programs should, of course, not be a purely theoretical endeavor. Throughout the last decade, many mature tools for heap analysis have been developed. In 2005, Berdine et al. [BCO05b] presented a symbolic execution based on a decidable fragment of separation logic. This was the reasoning method based on separation logic that achieved a high degree of automation and resulted in the Smallfoot tool [BCO06]. Since then, many tools based on separation logic with various emphases have appeared. Examples include VeriFast [JP08], which supports concurrency and is integrated with an SMT solver, THOR [Mag+08], which can deal with arithmetic properties as well as shape properties, and SLAyer [BCI11], which achieves a very high degree of automation for low-level code. Embeddings of SL fragments into first-order theories to reduce heap analysis to SMT solving have also been studied [PWZ13]. The Chalice tool [LMS09] provides permission-based reasoning for a concurrent object-oriented programming language. The VerCors

[1]After development of hyperedge replacement grammars for the recursive data structures that occur in the program.

tool [BH14], builds upon Chalice to enable the verification of concurrent `Java` programs. This list is necessarily incomplete, but highlights the interest of the research community in providing tool support for heap verification and the verification of concurrent programs.

In this thesis I am mainly concerned with a thorough exposition of the underlying theory, but a tool implementing this theory is also currently under development.

1.2 Summary of Contributions

My contributions are as follows.

1. A complete formal semantics of heap-manipulating programs with (a restricted form of) fork–join parallelism based on hypergraphs
2. An abstract interpretation of that semantics
3. A reformulation of the interprocedural analysis from [JN14] in terms of the abstract interpretation that applies to sequential as well as concurrent programs

1.3 Outline

I begin by defining basic notation, the syntax of our programming language, and the hypergraph model of program states in Chapter 2. Chapter 3 develops the formal semantics and grammar-based abstract interpretation of sequential programs written in our programming language. Chapter 4 extends both the semantics and the abstract interpretation to concurrent programs. In Chapter 5, I develop an interprocedural analysis based on the abstract interpretations. This development extends upon work by Jansen and Noll [JN14]. I conclude and discuss possibilities for future work in Chapter 6.

Basic familiarity with order theory and data-flow analysis are assumed throughout this thesis; to make the thesis self-contained, I sum up the key concepts in Appendices A and B. In addition, I give a brief tour of my prototypical implementation in Appendix C.

1.4 How to Read this Thesis

The text is quite technical at times, as I made an effort to rigorously define all terms and give a complete, unambiguous formalization of the presented program analyses. I tried to convey the intuition in all cases, so that it will often be sufficient if you just skim through the formal details.

There also is a detailed index at the end of the thesis, where you can look up all terms and abbreviations that I introduce throughout the thesis. Symbols can be found at the beginning of that index.

Whenever I want to draw attention to clarifying remarks, I set them apart from the surrounding text using Ⓡ. Particularly important assumptions or observations are highlighted using ❗.

CHAPTER 2

Background

This chapter sets the stage for the rest of the thesis. I begin by introducing a few notational conventions in Section 2.1. I present PL, a toy programming language with pointers and fork–join shared memory concurrency in Section 2.2; PL shall be the object of study throughout the remainder of the thesis. Section 2.3 introduces hypergraphs and hyperedge replacement grammars and motivates their use as a model for the representation and abstraction of heaps. It contains both the theoretical background and an informal preparation for the hypergraph-based semantics and static analyses that we shall explore in later chapters.

2.1 Basic Notation

I sum up the notational conventions that I use in this thesis in the following list. Throughout the text, I shall write

- 2^A for the power set of A
- $f : A \dashrightarrow B$ for a partial function from A to B.
- $f(x) = \bot$ if f is a partial function and undefined at x
- $Dom(f)$ and $Cod(f)$ for the domain and codomain of f
- $f \restriction C$ to mean the restriction of $f : A \to B$ to the domain $C \cap A$
- $f \cup (k \mapsto v)$ for the function that is obtained by extending or updating f according to the given mapping, i.e., $(f \cup (k \mapsto v))(x) := \begin{cases} v & \text{if } x = k \\ f(x) & \text{otherwise} \end{cases}$
- $f \cup g$ for combining functions with disjoint domains, i.e., given $f : A \to B, g : C \to D$, $A \cap C = \emptyset$, we define $(f \cup g)(x) := \begin{cases} f(x), & \text{if } x \in A \\ g(x), & \text{otherwise} \end{cases}$
- $\langle x_1, \ldots, x_n \rangle$ for **ordered sequences** and ϵ for the empty sequence $\langle\rangle$
- $\langle x_1, \ldots, x_n \rangle \cdot \langle y_1, \ldots, y_m \rangle$ for the concatenation of sequences $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$
- $x_1 :: \langle x_2, \ldots, x_n \rangle$ for destructuring the sequence $\langle x_1, x_2, \ldots, x_n \rangle$ into its head and tail
- $|x|$ refers to the cardinality of a set x as well as the length of a sequence x
- $x \in \langle x_1, \ldots, x_n \rangle$ to denote that $x \in \{x_1, \ldots, x_n\}$
- $s(i)$ to refer to the i-th element of a sequence $s = \langle x_1, \ldots, x_n \rangle$, where $i \in \{1, \ldots, n\}$
- A^*, A^+ and A^ω for finite, non-empty finite and infinite sequences over A, respectively

It is, of course, not necessary to memorize this list; you should just remember that you can look up notation here.

2.2 A Concurrent Pointer Language

2.2.1 Syntax

In this thesis we shall study a pointer-based programming language with fork–join parallelism. In this section I define the syntax of the language and describe an informal semantics; the formal semantics shall be developed in Chapter 3. I shall refer to the language simply as PL and to its sequential fragment—that is, to PL programs without fork or join statements—as PL_{seq}. PL is defined by the EBNF-style grammar in Fig. 2.1 on the facing page.

Let us have a look at this definition in detail. PL is a parallel, imperative, and statically typed programming language. A PL program consists of a list of procedures, and a list of type declarations. Although this is not enforced by the grammar in Fig. 2.1, we shall assume that both procedures and types have unique identifiers and that there is a single procedure named `main` that serves as the unique program entry point. Each procedure expects a non-empty list of typed parameters of pairwise different names. Procedure calls may be recursive.

R Following the convention of, e.g., [Aho+06], I use **procedure** as a generic term in place of any of **function**, **subroutine**, **method**, or **message**.

In addition to the **procedure declarations**, each program contains a list of **type declarations**. PL types are similar to record types in C: Each type declaration defines a fixed list of typed **fields**. They can be accessed via a string identifier, that we will henceforth call a **selector**. Type definitions may (and very often will) contain self-references. This makes it possible to define recursive data types such as lists or binary trees.

Variables in a PL program are either **pointers**, declared via the **var** keyword, or **thread tokens**, declared via **thread**.

Pointers are typed according to the type declarations given in the program. The type's fields can be accessed via the selector syntax $x.s$, mirroring the familiar syntax from C, C++, or `Java` (where we would access attributes of objects rather than fields of records).

R Despite the restriction of PL to pointer variables, it is Turing-complete: We can encode the natural numbers by associating 0 with the null pointer and defining a successor type **type** S **is** $prev$. On the basis of this encoding, we can easily define a counter machine model.

There are two types of references in PL: References to variables that consist of single identifiers, such as x, and references to selectors, such as $x.s$. We do not allow nesting of selectors such as $x.s.r$, because a maximum dereferencing depth of 1 simplifies the semantics.

We shall assume that all variables are declared and assigned a static type before usage. This is done via statements such as **var** $x : t$. This way we can easily check for consistent typing of all variables throughout the program.[1] Given that x has been declared to be of type t, we can either allocate new memory of the given type via $x :=$ **new** t or make x an alias to another pointer via $x := p$, provided that p either refers to a location of type t or the **null** pointer.

[1]Without this restriction, we would need to run an extra static analysis to rule out programs of the kind **if** $x = y$ **then** $z :=$ **new** $list$ **else** $z :=$ **new** $tree$, which we are going to disallow to conform to other statically typed languages such as C or `Java`.

```
⟨prog⟩      ::= ⟨proc-list⟩ ⟨type-list⟩

⟨proc-list⟩ ::= ⟨proc⟩ | ⟨proc⟩ ⟨proc-list⟩

⟨proc⟩      ::= procedure ⟨id⟩ '(' ⟨typeddecl-list⟩ ')' is ⟨stmt⟩

⟨stmt⟩      ::= 'skip'                                          (* No operation *)
              | 'var' ⟨id⟩ : ⟨id⟩                    (* Pointer variable declaration *)
              | ⟨ref⟩ ':=' ⟨ptr⟩                                    (* Assignment *)
              | ⟨id⟩ ':=' 'new' ⟨id⟩                                (* Allocation *)
              | 'call' ⟨id⟩ '(' ⟨id-list⟩ ')'                    (* Procedure call *)
              | 'thread' ⟨id⟩                         (* Thread variable declaration *)
              | ⟨id⟩ ':=' 'fork' ⟨id⟩ '(' ⟨id-list⟩ ')'          (* Thread forking *)
              | 'join' ⟨id⟩                                     (* Thread joining *)
              | ⟨stmt⟩ ';' ⟨stmt⟩                                (* Concatenation *)
              | 'if' ⟨expr⟩ 'then' ⟨stmt⟩ 'else' ⟨stmt⟩            (* Conditional *)
              | 'while' ⟨expr⟩ 'do' ⟨stmt⟩                          (* While loop *)
              | '(' ⟨stmt⟩ ')'                    (* Grouping to resolve ambiguity *)

⟨expr⟩      ::= ⟨ptr⟩ '=' ⟨ptr⟩
              | ⟨ptr⟩ '≠' ⟨ptr⟩
              | ⟨expr⟩ '∧' ⟨expr⟩
              | ⟨expr⟩ '∨' ⟨expr⟩

⟨ref⟩       ::= ⟨id⟩ | ⟨id⟩ '.' ⟨id⟩

⟨ptr⟩       ::= ⟨ref⟩ | 'null'

⟨type-list⟩ ::= ⟨type⟩ | ⟨type⟩ ⟨type-list⟩

⟨type⟩      ::= 'type' ⟨id⟩ 'is' ⟨typeddecl-list⟩

⟨typeddecl-list⟩ ::= ⟨typeddecl⟩ | ⟨typeddecl⟩ ',' ⟨typeddecl-list⟩

⟨typeddecl⟩ ::= ⟨id⟩ ':' ⟨id⟩

⟨id-list⟩   ::= ⟨id⟩ | ⟨id⟩ ',' ⟨id-list⟩

⟨id⟩        ::= [a-zA-Z][a-zA-z0-9]*
```

Figure 2.1.: Definition of PL

We shall also assume that both variables and thread tokens can only be declared top-level, i.e., not within if or while blocks. If that were not the case, scoping for if and while would become more difficult: We would have to clean up upon leaving the block. I opted against this flexibility to simplify the semantics.

There are no return values, so the only way to realize return values in PL is to assign to a parameter's fields. It is also possible to override parameters directly, but we shall assume that parameters are passed **by value**, so overriding a parameter does not have any effect at call site. This mirrors the way that object references are passed to methods in `Java`.[1]

There is no explicit way to free memory. Instead we assume the presence of a garbage collector. I made this choice mainly because it fits the hypergraph-based memory model very nicely, as we shall see in the following section and chapter. The approach is not limited to garbage-collected languages, however—the semantics in Chapter 3 could be adapted to support explicit memory deallocation. The language constructs discussed so far constitute the **atomic statements** of PL.

Definition 2.1 — **Atomic statement.** We call $c \in \mathbf{Cmd}$ **atomic** if c is of the form **skip**, **var** x, $p_1 := p_2$, $p :=$ **new** t.

Atomic statements are atomic in the sense that they cannot be split. Most atomic PL statements will not actually be atomic in the sense that they can be compiled into a single machine instruction.

Apart from the restriction to pointers, PL is quite standard: Usual imperative control-flow is available via comma-separated command sequences, (stateful) **if** branching, **while** loops, and side-effecting procedure calls. The conditions for **if** and **while** are (possibly nested) disjunctions and conjunctions of pointer comparisons.[2]

Beside blocking procedure calls to a procedure p via **call** $p(x_1, \ldots, x_n)$, it is also possible to run p in a separate thread: $tid :=$ **fork** $p(x_1, \ldots, x_n)$ creates a thread, stores the thread's unique identifier in tid, and runs p on arguments $x_1, \ldots, x_n$ in the new thread. Using tid, we can later **join** tid, i.e., block execution until the thread's execution has finished. In this way, PL supports fork/join parallelism very much akin to POSIX threads in C [**pthread95**] and similar to `Java` [Gos00].[3] Note that, both for call and for fork statements, we only allow variables as parameters, not arbitrary pointer expressions. While this is inconvenient from the programmer's point of view, it simplifies the call and fork semantics.

PL's multi-threading is restricted in one important way, however: Thread variables cannot be passed around. Thus threads are either joined in the same scope where they are forked or are not joined at all. I shall discuss the motivation for and implications of this restriction when I develop the formal semantics for fork and join in Chapter 4.

! Allow me to call attention once again to the most important characteristics of PL, so as to avoid any confusion in the subsequent formalization of the semantics.

- We assume garbage collection of memory after the last reference to a memory location has run out of scope; there is no explicit deallocation mechanism.
- All data in PL program are heap-allocated records, each of which consists of a fixed number of pointers to other records, as defined in the corresponding type declarations. Each record has a fixed static type.

[1]Confusingly, pointers are called references in Java, even though they are always passed by value.

[2]There is no direct support for Boolean literals, but we can, of course, express them: $true \equiv null = null$, $false \equiv null \neq null$.

[3]In `Java`, we fork off `Thread` objects rather than procedures, and threads can be joined multiple times.

- Variable and thread token declarations are not allowed within if or while blocks.
- Parameters are passed call by value.
- Forked procedures must be joined in the same scope where they are forked or are never joined.

Example

We must postpone the discussion of the precise program semantics for the moment: We first need a memory model, which we shall develop in the next section. Let us instead look at a simple example program, depicted in Fig. 2.2. If you have some experience in a language like `C` or `Java`, this program will hardly be surprising to you. The program recursively copies a doubly-linked list. We make the assumption that pointers to the first and the last element of the list are passed as parameters to the main procedure, In each call to `copy`, a single list cell is copied. The recursion stops once we reach the last element of the original list.

A doubly-linked list element or `elem` contains pointers to the previous and next element, hence the definition of the `elem` type lists two fields, `prev` and `next`. Note that there is no `value` field, since we do not represent values in PL. As it stands, the program is in fact a PL_{seq} program, i.e., a program written in the sequential fragment of PL. If we replaced

```
copy(head, last, copiedhead)
```

with

```
thread tid;
tid := fork copy(tmp, last, tmpcopy);
join tid
```

we would obtain a PL program that starts a separate execution thread for copying the list and then waits for that thread to finish.

```
procedure main(head : elem, last : elem) is
  var copiedhead : elem;
  copiedhead := new elem;
  if (head != last) then
    copy(head, last, copiedhead)
  else
    skip
procedure copy(cur : elem, last : elem, curcopy : elem) is
  var tmp : elem;
  var tmpcopy : elem;
  tmp := cur.next;
  tmpcopy := new elem;
  curcopy.next := tmpcopy;
  tmpcopy.prev := curcopy;
  if (tmp != last) then
    copy(tmp, last, tmpcopy)
  else
    skip
type elem is
  prev : elem;
  next : elem
```

Figure 2.2.: A PL program for copying doubly-linked lists

2.2.2 Mathematical representation

As syntax trees are quite unwieldy, we shall define an intermediate representation of PL programs.

R The definitions in the remainder of this section may look quite pointless at the moment, or at least very technical (which is exactly what they are). The reason to define a mathematical representation is that it is easier to define the formal semantics in terms of such a representation than in terms of a syntax tree. It is not necessary to understand the definitions in detail right now, let alone memorize them, to be able to follow the rest of the thesis. In later parts of the thesis, I shall explicitly refer you back to this section when we need the definitions.

First of all, let $\mathbf{Cmd}$ be the set of all statements or commands that can be generated from PL's grammar, i.e. those syntactical entities generated started from $\langle stmt \rangle$. Let $\mathbf{Cmd_{seq}}$ specifically refer to the commands of PL_{seq}. Further, let $\mathbf{Id}$ be the set of all valid identifiers.

We define a mathematical representation for the sets of all programs, procedures, and type declarations, $\mathbf{Progs}$, $\mathbf{Procs}$, and $\mathbf{Types}$.

Definition 2.2 — **Mathematical representation of programs.**

$$\begin{aligned} \mathbf{Progs} &:= \mathbf{Procs} \times \mathbf{Types} \\ \mathbf{Procs} &:= 2^{\mathbf{Id} \dashrightarrow ((\mathbf{Id},\mathbf{Id})^{+},\mathbf{Cmd})} \\ \mathbf{Types} &:= 2^{\mathbf{Id} \dashrightarrow (\mathbf{Id},\mathbf{Id})^{*}} \end{aligned}$$

The idea behind this mathematical representation is as follows.

- Each program can be regarded as a product of procedure declarations and type declarations
- The set of procedure declarations of a program can be viewed as a partial function defined on the identifiers of the declared procedures. Each valid identifier is mapped to a sequence of identifier pairs representing the parameters' names and types, as well as the procedure body.
- Likewise, the set of type declarations of a program can be viewed as partial function from identifiers—the declared types—to sequences of identifier pairs—the types' named fields.

R PL programs can be transformed into the above representation easily by standard parsing techniques. I shall therefore skip this transformation.

It is useful to have auxiliary notation to simplify accessing the relevant parts of the program representation.

- Let $P = (\Pi, \mathtt{T}) \in \mathbf{Progs}$. We define accessor functions $procs(P) := \Pi$ and $types(P) := \mathtt{T}$.
- Let $\pi := (\langle (p_1,t_1), \ldots, (p_k,t_k) \rangle, c) \in ((\mathbf{Id},\mathbf{Id})^{+}, \mathbf{Cmd})$ be a representation of a procedure's parameters and body, i.e., in the image of a function $\Pi \in \mathbf{Procs}$. $params(\pi) := \langle (p_1,t_1), \ldots, (p_k,t_k) \rangle$, $names(\pi) := \langle p_1, \ldots, p_k \rangle$, $typesig(\pi) := \langle t_1, \ldots, t_k \rangle$, and $body(p) := c$.
- Let $\mathtt{T} \in \mathbf{Types}$, t an identifier and $\mathtt{T}(t) = \langle (s_1,t_1), \ldots, (s_k,t_k) \rangle$. We write $sels(\mathtt{T}(t))$ to extract the sequence of selectors from the type definition, i.e., $sels(\mathtt{T}(t)) = \langle s_1, \ldots, s_k \rangle$.
- Finally, we sometimes need to refer to the sets of all variables and selectors that occur in a program P. We write $\mathbf{Var}_P$ and $\mathbf{Sel}_P$ to refer to these sets.

R The motivation for defining sets of procedures and type declarations as partial functions is as follows. Let P be a program in the sense of Def. 2.2. We can now simply access a procedure named `main` by the function application $procs(P)(main)$, its body by $body(procs(P)(main))$, and access the fields of a type `typ` via $types(P)(typ)$. This allows for a very concise and precise formulation of interprocedural and type-safe program semantics, as we shall see in Chapter 3.

■ Example 2.3 — **Mathematical representation.** Recall the example program in Fig. 2.2 on page 9. The mathematical representation of that program is $P := (\Pi,\mathtt{T})$, where

- $\Pi(main) = (\langle(head,elem),(last,elem)\rangle, \textbf{var}\ copiedhead : elem; \ldots \textbf{skip})$
- $\Pi(copy) = (\langle(cut,elem),(last,elem),(curcopy,elem)\rangle, \textbf{var}\ tmp : elem; \ldots \textbf{skip})$
- $\Pi(id) = \bot$ for all $id \notin \{main,copy\}$
- $\mathtt{T}(elem) = \langle(prev, elem),(next, elem)\rangle$
- $\mathtt{T}(id) = \bot$ for all $id \neq elem$
- $\mathbf{Var}_P = \{head, last, copiedhead, cur, curcopy, tmp, tmpcopy\}$
- $\mathbf{Sel}_P = \{prev, next\}$

■

The definitions of **Progs**, **Procs**, and **Types**, are purely syntactical in nature. For example, programs that call procedures that have not been declared are also contained in **Progs**. This observation motivates the final two definitions of this section.

Definition 2.4 — **Type system.** Let $\mathtt{T} : \mathbf{Id} \dashrightarrow \mathbf{Id}^* \in \mathbf{Types}$. $\mathtt{T}$ is a **type system** if

- $\mathtt{T}$ is non-empty: $Dom(\mathtt{T}) \neq \emptyset$
- $\mathtt{T}$ is self-contained: For all $t \in Dom(\mathtt{T})$ and for all $i \in \{1,\ldots,|\mathtt{T}(t)|\}$, $\mathtt{T}(t)(i) \in Dom(\mathtt{T})$

Definition 2.5 — **Well-formed program.** A program $P \in \mathbf{Progs}$ is **well-formed** if

- It has a `main` procedure, i.e., $procs(P)(main)$ is defined
- All call and fork statements refer exclusively to procedures that have been declared, i.e., $procs(P)(id)$ is defined for all $\textbf{call}\ id(\ldots)$ and $t := \textbf{fork}\ id(\ldots)$ statements in P. Furthermore, the correct number and types of arguments are passed to all calls and forks.
- All types used in procedure and variable declarations and memory allocation are declared, i.e., $types(P)(id)$ is defined for all
  - $typesig(\pi)$, where $\pi \in Dom(\Pi(P))$
  - $\textbf{var}\ x : id$, $x := \textbf{new}\ id$ in P
- The set of type declarations is a type system according to Def. 2.4

The set of well-formed programs is exactly the set of programs that we would expect to compile without error.

! We are only interested in the semantics and analysis of well-formed programs. Henceforth, we shall therefore assume that all programs be well-formed in the sense of Def. 2.5; a condition which would, of course, be enforced by any compiler for PL.

This completes the formalization of the syntax of PL. In the next section, I motivate the use of hypergraphs as a model for representing the state of PL programs and develop the necessary theoretical foundations. The current section together with the upcoming section come together in Chapter 3, where we shall explore formal hypergraph-based semantics for (the mathematical representation of) PL programs.

2.3 Hypergraphs as Model of Program States

To reason formally about programs with recursive procedures and local variables, we need a mathematical model of the stack. Likewise, to reason formally about programs with pointers, we need a mathematical model of the heap. Consequently, the model of the stack and heap that we develop in this section is going to be central to the remainder of this thesis.

Let us briefly pause to consider one key requirement: Our model must support a means of **abstraction**.

> Definition 2.6 — **Abstraction [Rus95].** Abstraction is the process of simplifying certain details of a system description or model so that the main issues are exposed. Abstraction is the key to gaining intellectual mastery of any complex system, and a prerequisite to effective use of formal methods. In formal methods, abstraction is part of the process of developing a mathematical model that is a **simplification** or approximation of reality but that **retains the properties of interest**.

We cannot reason individually about all—infinitely many—different heaps that a given program might conceivably operate on. In terms of the example program from the previous section, we would like to be able to reason abstractly about copying arbitrary doubly-linked lists, not specifically about lists of, say, length 23. To this end, we have to identify some common structural properties of the heaps that may occur in the execution of the program and then reason about the entire class of all heaps that exhibit these structural properties. In the example, an appropriate class of heaps may be that of all non-circular doubly-linked lists connected via `next` and `prev` pointers.

This pattern—abstraction of heaps followed by abstract reasoning to derive general results—is common to many verification formalisms for pointer programs; consider, for example, the (recursive) predicates of separation logic [Rey02], or the shape graphs of shape analysis [SRW02].

In this thesis, **hypergraphs** serve as an integrated representation of the heap as well as the local stack variables. To abstract structural properties, we use **hyperedge replacement grammars** (HRGs). The idea to use hypergraphs and HRGs to model (abstract) heaps is not new: In this section I present and expand upon ideas published between 2008 and 2015 [Dod09; Rie09; HNR10; Jan+11; JN14; Hei+15].

This section is structured as follows. Section 2.3.1 is an informal introduction to the hypergraph-based model of heaps and stacks. The subsequent formalization is broken into two parts, the introduction of hypergraphs in Section 2.3.2 and of hyperedge replacement grammars in Section 2.3.3. The latter section in particular is quite lengthy and may be skimmed on first reading, to be consulted as reference when the material is used in later chapters. (All terms and definitions are indexed at the end of the thesis.)

2.3.1 Motivation

When verifying interprocedural pointer programs, i.e., heap-manipulating programs that involve calls to procedures, one needs to model both **local variables** and the **runtime heap**. Such a model, together with a representation of the call stack(s)[1], then constitutes the **program state**.

[1]The plural applies in the multi-threaded setting

A heap can naturally be viewed as a graph, where each vertex corresponds to a memory location or a block of locations—for example, representing a whole record structure or object—while the edges of the graph represent pointers between locations.

There are multiple ways of defining such a graph encoding of heaps, influenced both by the memory model (of the programming language and underlying machine) and the desired level of abstraction. In our particular model, we shall always assume that each node represents a whole PL record. Each PL record has a fixed data type, which is why we shall need typed nodes. Records have fixed numbers of fields according to their data type. Each field contains a pointer to another record, i.e., to another vertex in the graph representation of the heap. These pointers are modeled as labeled edges, where each edge is labeled with the name of the field that contains the pointer. This labeling is necessary to distinguish between fields, since the whole record structure is represented as a single node of the graph, so we usually need to associate more than one pointer of any given type with any given node.

Optionally, a heap may also contain a **null node** for each data type. These null nodes represent the (unique) **null pointer**. Analogously to, for example, `Java` object references, uninitialized pointers of any type will be represented by pointing the corresponding edge to the type's null node. We introduce one null node per type rather than one global null node to simplify the semantics of static typing (see Chapter 3). Null nodes are added and removed on demand; if no pointers of type t have been assigned null, and there are no uninitialized pointers of type t in the heap or uninitialized variables of type t on the stack, the corresponding heap graph should not contain the null node of type t.[1]

In accordance with the definition of PL, we do not model any data apart from pointers to other locations. Thus a heap graph only models the structure of a heap, not the concrete values that a memory location contains.

I will henceforth call graphs that follow the above encoding **heap configurations** or **heap graphs**, but defer the formal definition until Section 2.3.2.

■ Example 2.7 — **Heap graph for a doubly-linked list.** Consider the case of doubly-linked lists. Each element of a doubly-linked list usually consists of a value, which we do not model in our approach, and pointers to the previous and next element of the list. A doubly-linked list with four elements would thus be represented by a graph with four nodes and eight edges as depicted in Fig. 2.3. As before, we use `next` and `prev` as names for the pointers. We (arbitrarily) assign names $v_1 \dots v_4$ to the list elements; these names are not necessary, but useful to refer to nodes in the text. The first node v_1 does not have a predecessor, the last node v_4 does not have a successor. In our graph model, we depict this by adding a special null node, which we label $\perp$ (for undefined). To improve readability of the graphs, we sometimes depict several or zero such null nodes. This is why there are two null nodes in the figure. It is, however, sufficient (and desirable for the implementation) to only add one null node per type, which is in fact enforced by the formal definition of heap graphs in Section 2.3.2. ■

[1]This model does not account for the possibility of C-like uninitialized fields: Rather than pointing to `null`, an uninitialized pointer in C may point to any arbitrary memory location, often leading to non-deterministic results upon dereferencing. If we wanted a more C-like memory model, we could adapt our model to incorporate a special **uninitialized node** in addition to and separate from the **null node**. This would allow us to differentiate between null pointer dereferences and other segmentation faults, thereby yielding a closer but still deterministic approximation of the C semantics. A true C semantics is not possible with our model, however, since it is inherently **storeless** [BIL03]

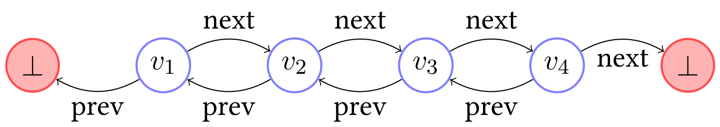


Figure 2.3.: Heap graph of a doubly-linked list with four elements $v_1, \ldots, v_4$. The pointers between the nodes are represented by the labeled edges, uninitialized pointers are represented by pointers to special null nodes that are labeled with $\perp$.

As motivated at the beginning of this chapter, we usually do not want to reason specifically about lists of length four. The programs that are the objects of our analyses are generally written to process arbitrary input data. For example, a procedure for list traversal should be able to traverse lists of arbitrary length rather than only lists of length four. Since lists of different lengths are represented by different heap graphs, the state space of any program that deals with lists will generally be infinite. The same observation applies to other recursive data structures such as trees. Since we need to have a finite state space if we want to apply data-flow analyses and verification techniques such as model checking, we employ abstraction techniques to obtain a finite representation of the state space. This abstraction necessarily comes at the cost of over-approximating the state space, so it is essential that it preserves enough of the original structure of the input data to be able to draw meaningful conclusions [Rus95; Jon12].

One such way to abstract heap graphs is via **graph grammars**. Akin to string grammars, graph grammars consist of a set of **production rules** for rewriting graphs. In this thesis, we employ hyperedge replacement grammars (HRGs), a type of context-free (hyper-)graph grammars. Several other types of graph grammars have been studied; for an introduction see [DKH97]. HRGs operate on hypergraphs, which generalize graphs to allow edges that connect arbitrarily many nodes (rather than just two). The generalized edges are called **hyperedges**, hence the name hyperedge replacement grammar. (A formal definition of HRGs is given later in this chapter.)

Each HRG is defined in terms of **nonterminal** hyperedges, the analog of nonterminal symbols in string grammars. Each production rule of the grammar provides a way to replace a single nonterminal hyperedge with a hypergraph. The replacing hypergraph may again contain nonterminals. Consequently, arbitrarily long sequences of hyperedge replacements—called **derivations** or derivation sequences—may be possible.

■ Example 2.8 Continuing our running example, Fig. 2.4 shows a simple hyperedge replacement grammar for doubly-linked lists. The figure introduces the rectangle notation for representing hyperedges: Throughout the remainder of the thesis, I shall always draw ordinary edges—i.e., pointers between locations—as arrows. I shall draw all other hyperedges as rectangles that are connected to all the attached nodes of the hyperedge. The digits above the connecting edges reflect the order in the attachment.[1] In this example, this applies to the L edges.

Now let us examine the grammar. A list is either empty or consists of a head element and the remaining list, commonly called the tail of the list: L $::= \epsilon \mid$ a.L. This algebraic structure is directly captured in the two rules of the grammar: The first rule says that a list nonterminal, i.e., a hyperedge labeled L, that is connected to two nodes can be replaced by just a `prev` and a `next` edge. In this case, we interpreted the L hyperedge as representing an empty list. The second rule says that we can also replace a nonterminal by adding a

[1] I adopted this notational convention from my sources [Jan+11; JGN14; JN14].

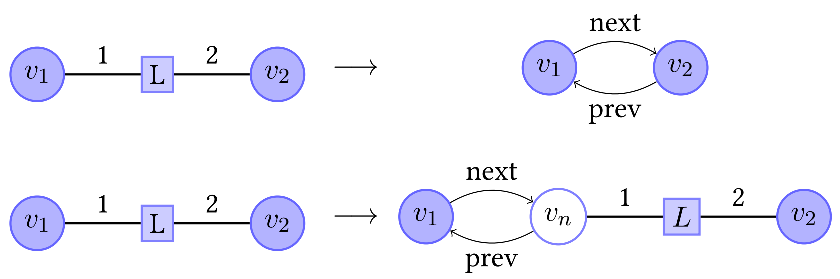


Figure 2.4.: A hyperedge replacement grammar for doubly-linked lists. The first rule replaces an abstract L edge with an empty list, whereas the second rule produces a concrete head element v_n followed by another abstract L edge.

new node (here named v_n) and a new L nonterminal, again connected to two nodes, and abstracting the remainder of the list.[1] In other words, applying the second rule to the original L nonterminal yields the head and a (possibly empty) tail list. The blue background of v_1 and v_2 signals that these nodes remain unchanged by the rule application. ■

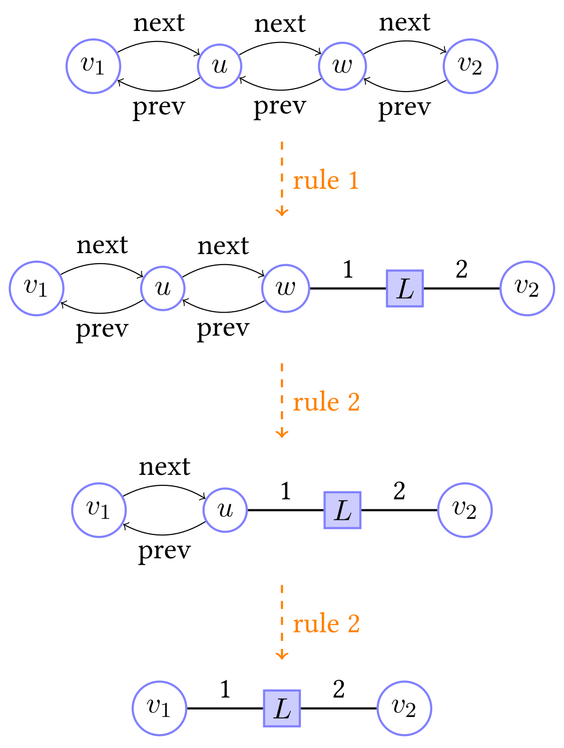
A hypergraph that contains a nonterminal is thus abstract in the sense that it represents multiple concrete heaps that can be derived via rule application. Given a hypergraph that contains nonterminals, we can apply rules forward to derive a graph that is more concrete in the sense that the set of heaps it represents is a subset of the set represented by the hypergraph before rule application. We therefore commonly refer to forward application of HRG rules as **concretization**. Conversely, we can also apply rules backward, to obtain more concise and less precise representations, a process called **abstraction**.

■ Example 2.9 Fig. 2.5 on the following page illustrates rule applications in both directions. In Fig. 2.5a, we show the abstraction process. Starting from the graph in Fig. 2.3—modulo renaming of nodes and omitting the null nodes for clarity—we apply the first rule from the grammar defined in Fig. 2.4 backward to replace the connection between the last two nodes by an abstract L hyperedge. We then apply the second rule twice to obtain a graph with just two nodes and a single L nonterminal.

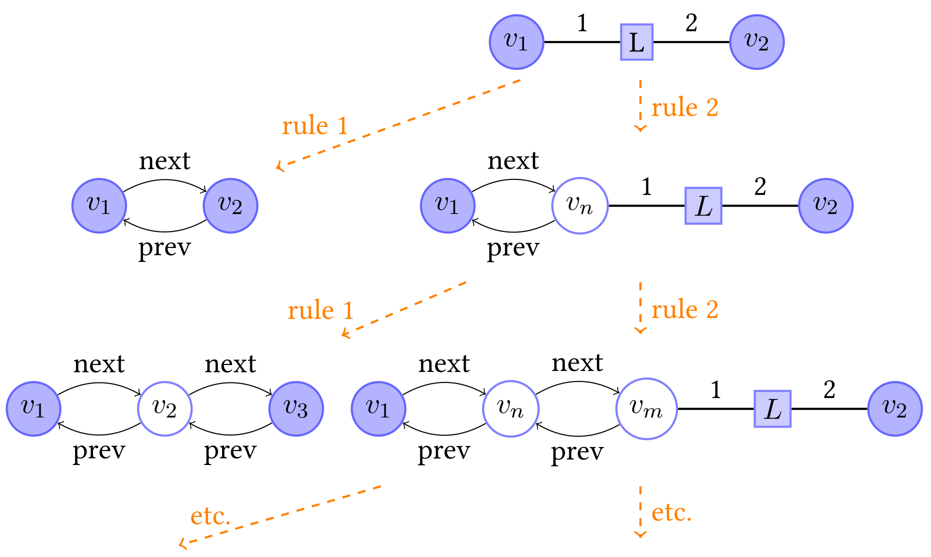
Fig. 2.5b shows the forward application starting from this abstract hypergraph. This exemplifies how the abstraction process over-approximates the real state space: We started with a concrete four-element list, but by performing full abstraction using the HRG, we lost this information and are thus able to derive arbitrary lists in the subsequent concretization. If this is undesirable in our given setting, we could either use a different grammar for performing the abstraction or only perform partial abstraction. ■

The purpose of abstraction is clear in our setting: It enables finite abstractions of the state space. The merit of concretization might be less obvious. It becomes clear when thinking about the semantics of programs. To assign a meaning to the statement $x.next := y$, for example, we must resolve a list element's `next` pointer. But what happens if x points to a node in the heap that is only connected to a nonterminal L edge, because the list has been fully abstracted? We need to apply concretization **on demand** to replace the node's adjacent nonterminal by concrete locations and pointers; we will see the details in Section 3.2, where we develop an abstract interpretation based on abstraction and on-demand concretization.

[1]Note that in this simple case, since the nonterminal is attached to only two nodes, we would not need to generalize our model to hypergraphs. For an example with hyperedges attached to three nodes, see [Jan+11]



(a) HRG-based abstraction by right-to-left application of production rules



(b) HRG-based derivation of (more) concrete graphs by left-to-right application of production rules

Figure 2.5.: HRG-based abstraction and concretization of doubly-linked lists

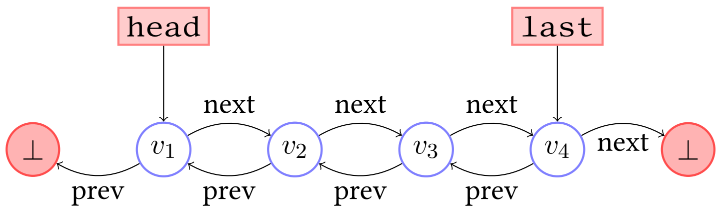


Figure 2.6.: Integrated representation of heap and stack. Hyperedges of degree 1 have been added for the variables that are currently in scope, `head` and `tail`

R You may have noticed that, in the previous example, the abstraction of the four-element list to a single nonterminal only succeeded because we chose to begin the abstraction by applying the first rule to the last two nodes of the list. Had we chosen any other way to apply the first rule, the abstraction process would have gotten stuck with at least two L hyperedges remaining. If such divergence in the abstraction process can never occur, the HRG is called **backward confluent**. By adding a new rule to the grammar in Fig. 2.4 that merges two adjacent nonterminals, we can easily establish this property. We shall study backward confluence as well as other desirable properties of HRGs in Section 2.3.3.

Besides the heap, we also need to incorporate the stack into our model of the program states. One way to do this, taken from [JN14], is to introduce additional hyperedges for keeping track of those variables that are currently in scope: For each variable, we add a hyperedge of degree 1, label it with the variable identifier and attach it to the memory location that the variable currently points to.

■ Example 2.10 — **Integrating the stack.** Let us assume that we run a list-processing program that has a variable `head` that points to the first element of the list and a variable `last` that points to the last element.[1] Fig. 2.6 shows how this information is added to the heap graph of the four-element list from Fig. 2.3. Note that we reuse the rectangle notation. This does not introduce ambiguity, since I exclusively use uppercase letters for nonterminals and exclusively use identifiers that start with a lowercase letter for variables. ■

The informal examples given above illustrate the approach to modeling the heap and stack that I take in this thesis (based on [Rie09; Dod09] as well as numerous subsequent publications including [Hei+15; JN14]). Before turning to the formalization, let us try to distill the essence of the approach[2] from the examples I presented.

- We model the heap as (hyper)graph, where nodes represent memory locations and edges represent pointers between locations.
- We add the local variables to the heap graph as hyperedges of degree 1.
- We capture the general structure of the heaps via hyperedge replacement grammars.
- We abstract concrete heaps via backward application of HRG-rules to obtain a finite over-approximation of the state space.
- For the abstract interpretation of programs, we concretize parts of the heap on demand via (forward) HRG derivations.

The next step shall be the formalization of these steps.

2.3.2 Heap Graphs

In this section I will make precise the notions of hypergraphs and heap graphs. We saw in the examples above that we need to label hyperedges, both for defining grammars and for

[1] Such as the program in Fig. 2.2 on page 9, for example.
[2] Measured in `\items` per `itemize` rather than alcohol by volume

modeling pointers between locations and local variables. Formally, these labels come from a fixed alphabet, each element of which is associated with a fixed **rank**[1], a natural number that determines the number of nodes that are attached to a hyperedge of that label. It makes sense to fix this rank: An abstract list hyperedge L always connects two nodes, never one or three. This idea is usually captured in the definition of **ranked alphabets** [DKH97].

Here we define a more specific variant of ranked alphabets following the observation that we always have exactly three distinct types of symbols in the hypergraphs that we consider:

- Variable identifiers for the hyperedges of degree 1 that we use to model the stack
- Selector identifiers labeling the concrete pointer hyperedges of degree 2
- Nonterminal symbols for use in the graph grammars

This observation motivates the following definition.

Definition 2.11 — **Heap Alphabet.** A **heap alphabet** Σ is a tuple $(\Sigma_V, \Sigma_S, \Sigma_{NT}, \mathrm{rk})$, where

- Σ_V is a finite set of **variable identifiers**
- Σ_S is a finite set of **selector identifiers**
- Σ_{NT} is a finite set of **nonterminal symbols**
- $\mathrm{rk} : (\Sigma_V \dot\cup \Sigma_S \dot\cup \Sigma_{NT}) \to \mathbb{N}_{>0}$ is the **ranking function**
- $\mathrm{rk}(\sigma) = 1$ for all $\sigma \in \Sigma_V$ and $\mathrm{rk}(\sigma) = 2$ for all $\sigma \in \Sigma_S$

By slight abuse of notation, I shall also refer to $\Sigma_V \dot\cup \Sigma_S \dot\cup \Sigma_{NT}$ by Σ.

To simplify notation, we assume that variables, selectors and nonterminals are always disjoint for any given program. This restriction can, of course, be lifted easily, e.g. by using typed labels.[2]

■ Example 2.12 In our running doubly-linked list example, $\Sigma = (\Sigma_V, \Sigma_S, \Sigma_{NT}, \mathrm{rk})$, $\Sigma_V = \{head, last\}$, $\Sigma_S = \{next, prev\}$, $\Sigma_{NT} = \{L\}$, and $\mathrm{rk}(L) = 2$ ■

We would like to use a heap alphabet together with a type system (cf. Def. 2.4 on page 11), because our heaps contain typed nodes. To this end, we must ensure that the heap alphabet contains all selectors that occur in the type system. We call this property **compatibility**.

Definition 2.13 — **Type–system-compatible heap alphabet.** A heap alphabet Σ is called **compatible** with a type system $\mathtt{T}$ if $\Sigma_S \supseteq Dom(\mathtt{T})$.

Our variant of hypergraphs is defined in terms of type systems and heap alphabets compatible with said type systems.

Definition 2.14 — **Hypergraph.** Let $\mathtt{T}$ be a type system and $(\Sigma_V, \Sigma_S, \Sigma_{NT}, \mathrm{rk})$ be a heap alphabet compatible with $\mathtt{T}$. A hypergraph $\mathcal{H}$ over $(\Sigma_V, \Sigma_S, \Sigma_{NT}, \mathrm{rk})$ and $\mathtt{T}$ is a tuple $\mathcal{H} = (V, E, att, lab, ext, typ, isnull)$, where

- V is the set of **nodes** or vertices
- E is the set of **hyperedges**
- $att : E \to V^*$ is the **attachment** function
- $lab : E \to \Sigma$ is the **labeling** function
- $ext : V^*$ is the sequence of **external nodes**
- $typ : V \to \mathtt{T}$ is a **type assignment**

[1]The term **type** is also commonly used, e.g. in [DKH97]. I shall instead only use the term rank to avoid confusion with the notion of (data) types in programming languages such as PL.

[2]Which is exactly what is done in the implementation.

- $isnull : V \to \{true, false\}$ indicates which nodes are null nodes
- $|att(e)| = \text{rk}(lab(e))$ for all $e \in E$

To refer to the i-th attached node of an edge e and the i-th external node, respectively, I write $att(e)(i)$ and $ext(i)$.

When I need to make explicit the hypergraph $\mathcal{H}$ to which a component belongs, I write $V_{\mathcal{H}}$,$E_{\mathcal{H}}$, etc. I use the notation $\mathbf{HG}_{\mathrm{T},\Sigma}$ or simply $\mathbf{HG}$ for the set of all hypergraphs over Σ and T.

Apart from typ and $isnull$, this definition is standard [DKH97]. Since each node in our setting represents a data structure of a fixed type, we need the additional typ component. The $isnull$ predicate lets us distinguish between initialized data structures and null nodes.

By modeling the hyperedges via a separate set E and an attachment function rather than as subsets of V, we obtain ordered hyperedges, i.e, we generalize the notion of ordered graphs. This is necessary in our setting, because the direction of hyperedges is essential for the semantics—it makes a big difference whether an `L` nonterminal that is attached to nodes n and m represents a list with head n or head m. Likewise, the external nodes—whose purpose will become clear once we formally define hyperedge replacement—are ordered.

All edges are labeled with symbols from a heap alphabet, as was the case in the examples we saw in Section 2.3.1; otherwise it wouldn't make sense to speak of `L` edges, for example. Note that I demand in the definition that hypergraphs be well-formed in the sense that each hyperedge is attached to exactly as many nodes as prescribed by the rank of its assigned label.

■ Example 2.15 Consider again the doubly-linked with four elements referenced by `head` and `last` variables (cf. Fig. 2.6). This corresponds to the hypergraph

$$\mathcal{H} = (V, E, att, lab, ext, typ, isnull) \in \mathbf{HG}_{\mathrm{T},\Sigma}$$

where

- Σ was defined in Ex. 2.12
- $\mathrm{T} = \{elem\}$
- $V = \{v_1,v_2,v_3,v_4,v_\perp\}$
- $E = \{n_1,n_2,n_3,n_4,p_1,p_2,p_3,p_4,h,l\}$
- $lab(h) := head, lab(l) := last, lab(n_i) := next, lab(p_i) := prev$
- $att(h) := \langle v_1 \rangle$, $att(n_1) := \langle v_1,v_2 \rangle$, $att(p_1) := \langle v_1,v_\perp \rangle$, $att(n_2) := \langle v_2,v_3 \rangle$, etc.
- $ext := \epsilon$
- $typ(v) = elem \; \forall v \in V$
- $isnull(v) = true$ iff $v = v_\perp$

Note that the naming of the nodes and edges—v_1,v_2,n_1,etc.—is completely arbitrary: The functions lab, att, typ, and $isnull$, as well as the word ext, assign the "meaning" to the members of V and E. ■

R While we technically have to include the ranking function and the type system whenever we formally define a hypergraph, both are almost always clear from the context. Thus I will only explicitly mention Σ and T when this improves clarity.

Not all hypergraphs represent valid heaps. A node with two outgoing `next` selector edges would, for example, be completely fine by Def. 2.14, but does not make sense in our model of the heap. This warrants another definition, namely that of the set of all valid **heap graphs**, henceforth referred to as **heap configurations** in accordance with the literature [JN14].

Definition 2.16 — **Heap configuration.** A hypergraph $\mathcal{H} \in \mathbf{HG}_{\mathtt{T},\Sigma}$ is called a (well-typed) **heap configuration** or **heap graph** if

- $\mathcal{H}$ is finite
- For all $v \in \Sigma_V$, $\mathcal{H}$ contains at most one hyperedge e with $lab(e) = v$
- For all nodes n and all $s \in \Sigma_S$, $\mathcal{H}$ contains at most one hyperedge e with $att(e)(1) = n$ and $lab(e) = s$
- For all $t \in \mathtt{T}$, $|\{v \in V \mid typ(v) = t \wedge isnull(v) = true\}| \leq 1$
- For all $v \in V$ with $isnull(v) = true$, $\{e | lab(e) \in \Sigma_S \wedge att(e)(1) = v\} = \emptyset$
- $\mathcal{H}$ is **well-typed** w.r.t. $\mathtt{T}$, i.e., for all types $t \in \mathtt{T}$, all $v \in V_{\mathcal{H}}$ with $typ_{\mathcal{H}}(v) = t$, and all $(s,t') \in \mathtt{T}(t)$,
  - There exists $e \in E_{\mathcal{H}}$ such that $lab_{\mathcal{H}}(e) = s$, $att_{\mathcal{H}}(e)(1) = v$, and $typ_{\mathcal{H}}(att_{\mathcal{H}}(e)(2)) = t'$.
  - There is no edge $e \in E_{\mathcal{H}}$ such that $lab_{\mathcal{H}}(e) \notin sels(\mathtt{T}(t))$

We write $\mathbf{HC}_{\mathtt{T},\Sigma} \subset \mathbf{HG}_{\mathtt{T},\Sigma}$ for the set of all well-typed heap configurations or simply $\mathbf{HC}$ if Σ and $\mathtt{T}$ are clear from the context.

Let us dissect this definition point by point.

- While there is no upper bound on the size of a heap, every heap is finite.
- Each pointer variable can only point to at most one heap location.
- At most one heap location can be assigned to each field of a record structure
- There is at most one null node per type.
- Null nodes do not have any outgoing selector edges. In this way we model that null nodes do not represent data structures (which would usually contain fields of pointers to other locations), but instead serve as a sink for null pointers.
- All our heap configurations are defined in terms of a type system, and in fact one invariant we want to preserve is **well-typedness** with respect to this type system: When we have a node of type t in a heap configuration, that heap configuration should also contain a selector edge for each pointer field that type t defines, and this edge should point to a node of the type of the field. It should also contain no other selector edges for the node.

R In the literature—for example in [Jan+11]—no type system is assumed. The conditions regarding null nodes and well-typedness are therefore not present in the standard definition of heap configurations.

As we shall soon see, it makes sense to distinguish between heap configurations that contain only terminal edges—concrete pointers and variables—and those that contain at least one nonterminal edge, henceforth called concrete and abstract heap configurations.

Definition 2.17 — **Concrete and abstract heap configurations.** Given a heap alphabet $\Sigma = (\Sigma_V, \Sigma_S, \Sigma_{NT}, \mathrm{rk})$ and a types system $\mathtt{T}$, we define

- **concrete heap configurations** $\mathbf{HC}^0_{\mathtt{T},\Sigma} := \{\mathcal{H} \in \mathbf{HC}_{\mathtt{T},\Sigma} \mid lab(\mathcal{H}) \cap \Sigma_{NT} = \emptyset\}$
- **(partially) abstract heap configurations** $\mathbf{HC}^{\geq 1}_{\mathtt{T},\Sigma} := \{\mathcal{H} \in \mathbf{HC}_{\mathtt{T},\Sigma} \mid lab(\mathcal{H}) \cap \Sigma_{NT} \neq \emptyset\}$

I shall omit the Σ and $\mathtt{T}$ indices and write $\mathbf{HC}^0 \subset \mathbf{HC}$ and $\mathbf{HC}^{\geq 1} \subset \mathbf{HC}$ when the indices are clear from the context.

In the previous examples, all hypergraphs that contained at least one L edge were (partially) abstract, while the others were concrete.

We conclude this section with some additional terminology, so that we can talk precisely about hypergraphs in the upcoming section. First, the direction of selector edges and thus of pointers is important.

Definition 2.18 — **Outgoing and incoming edges.** Let $\mathcal{H} \in \mathbf{HC}_{\mathrm{T},\Sigma}$ and $v \in V_H$. The set of **outgoing edges** of v, $out(v)$, is defined as $\{e \in E_{\mathcal{H}} \mid lab(e) \in \Sigma_S \wedge att(e)(1) = v\}$. Conversely, the $\{e \in E_{\mathcal{H}} \mid lab(e) \in \Sigma_S \wedge att(e)(2) = v\}$ is the set of **incoming** edges

Second, we need to be able to talk about subgraphs. More precisely, we need **section hypergraphs**, which generalize induced subgraphs.

Definition 2.19 — **Section Hypergraph.** Let H be a hypergraph and $W \subseteq V_{\mathcal{H}}$. Let $F := \{e \mid e \in E_H, att_H(e) \subseteq W\}$ and ext_W be the largest subsequence of $ext_{\mathcal{H}}$ that contains only nodes from W. The section hypergraph of H and W, $H \times W$, is the hypergraph

$$(W, F, att \restriction F, lab \restriction F, ext_W, typ \restriction W, isnull \restriction W)$$

Section hypergraphs are important for grammar-based abstraction, but also in the definition of the local procedure semantics in Chapter 3. In both cases, the nodes on the boundary between the section and the rest of the hypergraph play a special role, which warrants a special notation.

Definition 2.20 — **Section Boundary.** Let $\mathcal{H} \in \mathbf{HG}_{\mathrm{T},\Sigma}$ and $\mathcal{G} = \mathcal{H} \times W$ for some $W \subseteq V_{\mathcal{H}}$. The **section boundary** between $\mathcal{H}$ and $\mathcal{G}$, bound($\mathcal{H}$,$\mathcal{G}$) is defined as follows.

$$\mathrm{bound}(\mathcal{H},\mathcal{G}) := \{v \in V_{\mathcal{G}} \mid \exists e \in E_{\mathcal{H}} \setminus E_{\mathcal{G}}. v \in att(e)\}$$

In addition, we also define difference and union of hypergraphs in the obvious way.

Definition 2.21 — **Hypergraph difference.** Let $\mathcal{H}$,$\mathcal{G} \in \mathbf{HC}_{\mathrm{T},\Sigma}$. We define

$$\mathcal{H} - \mathcal{G} := (V, E, att, lab, ext, typ, isnull)$$

where

- $V := V_{\mathcal{H}} \setminus V_{\mathcal{G}}$
- $E := E_{\mathcal{H}} \setminus \{e \in E_{\mathcal{H}} \mid att(e) \cap V_{\mathcal{G}} \neq \emptyset\}$
- $att := att_{\mathcal{H}} \restriction E, lab := lab_{\mathcal{H}} \restriction E$
- ext is the longest subsequence of $ext_{\mathcal{H}}$ that contains only nodes from V
- $typ := typ_{\mathcal{H}} \restriction V, isnull := isnull_{\mathcal{H}} \restriction V$

Definition 2.22 — **Hypergraph union.** Let $\mathcal{H}$,$\mathcal{G} \in \mathbf{HC}_{\mathrm{T},\Sigma}$. We define

$$\mathcal{H} \cup \mathcal{G} := (V, E, att, lab, ext, typ, isnull)$$

where for all $a \in \{V,E,att,lab,typ,isnull\}$, $a := a_{\mathcal{H}} \cup a_{\mathcal{G}}$ and $ext := ext_{\mathcal{H}} \cdot ext_G$. Note that, because of the order on external nodes, hypergraph union is not commutative.

Finally, we sometimes need to determine whether two hypergraphs are **isomorphic**. We only call a mapping between edges an isomorphism if it preserves all structure apart from external nodes, i.e., all of lab,typ,$isnull$. External nodes play a special role and are therefore excluded from the definition. (Otherwise, we would, for example, not be able the right-hand side of a rule against part of a hypergraph.)

Definition 2.23 — **Hypergraph isomorphism.** Let $\mathcal{H}$,$\mathcal{G} \in \mathbf{HG}_{\mathrm{T},\Sigma}$ be hypergraphs with $|V_{\mathcal{H}}| = |V_{\mathcal{G}}|$, $|E_{\mathcal{H}}| = |E_{\mathcal{G}}|$, and without isolated nodes (i.e., for all nodes there is an edge that is attached to the node). Let $\varphi : E_{\mathcal{H}} \to E_{\mathcal{G}}$. φ is an isomorphism between $\mathcal{H}$ and $\mathcal{G}$ if

- φ is a bijection
- For all e, if $lab_{\mathcal{H}}(e) = \sigma$, then $lab_{\mathcal{G}}(\varphi(e)) = \sigma$
- For all e_1,e_2,i,j, if $att_{\mathcal{H}}(e_1)(i) = att_{\mathcal{H}}(e_2)(j)$, then $att_{\mathcal{G}}(\varphi(e_1))(i)$ $= att_{\mathcal{G}}(\varphi(e_2))(j)$
- For all e and i, $typ(att_{\mathcal{H}}(e)(i)) = typ(att_{\mathcal{G}}(\varphi(e))(i))$ and $isnull(att_{\mathcal{H}}(e)(i)) = isnull(att_{\mathcal{G}}(\varphi(e))(i))$

We write $\mathcal{G} \cong \mathcal{H}$ for isomorphic hypergraphs $\mathcal{G}$,$\mathcal{H}$.

R You might object to the definition of isomorphisms on edges rather than nodes. Note, however, that in our setting, the edges are not a relation but instead also objects in the structure: The universe of a hypergraph structure is the disjoint union of nodes and edges. An isomorphism in this setting has to preserve all functions on both nodes and edges. Conveniently, since we assume that there are no isolated nodes, it is, sufficient to define the isomorphism on edges, as this induces an isomorphism on the nodes in this restricted setting.

Our main motivation for looking at hypergraphs rather than graphs is the suitability of hyperedge replacement grammars for heap abstraction, which we examine next.

2.3.3 Hyperedge Replacement Grammars

Graph grammars have been used in various application areas from compiler construction to database design, from biology to software modeling and validation [Roz99]. In this thesis I use hyperedge replacement grammars (HRGs), a formalism for context-free hypergraph transformations [DKH97]. Just like string grammars are commonly defined via production rules that operate on nonterminal and terminal symbols, HRGs operate on hyperedges with nonterminal or terminal labels.

R **Partition of heap alphabets into terminals and nonterminals.** In this thesis, we deal with heap alphabets $(\Sigma_V, \Sigma_S, \Sigma_{NT}, \mathrm{rk})$ as opposed to general ranked alphabets in the sense of [DKH97]. In our special case, $\Sigma_V \cup \Sigma_S$ constitute the terminal labels, and Σ_{NT} the nonterminal labels.

Definition 2.24 — **Hyperedge replacement grammar.** Let T be a type system and $(\Sigma_V, \Sigma_S, \Sigma_{NT}, \mathrm{rk})$ be a heap alphabet compatible with T. A hyperedge replacement grammar $\mathfrak{G}$ over Σ and T is a finite set of **production rules** or **productions** $\mathcal{R}_1, \ldots \mathcal{R}_n$, where each rule $\mathcal{R}_i$

- is a tuple $(\sigma_i, \mathcal{G}_i)$, for some $\sigma_i \in \Sigma_{NT}$, $\mathcal{G}_i \in \mathbf{HG}_{\mathrm{T},\Sigma}$
- must satisfy $\mathrm{rk}(\sigma_i) = |ext_{G_i}|$

I sometimes write $\sigma_i \to \mathcal{G}_i$ instead of $(\sigma_i, \mathcal{G}_i)$, following the conventions for writing down (string) grammars.

The similarity with context-free string grammars is apparent in the syntactical structure of the rules: Each rule's left-hand side must consist of a single nonterminal. We thus say that HRGs are context free [DKH97].

■ Example 2.25 Recall Ex. 2.9 from page 15. The (partially) abstract heaps in the language derivation contain L nonterminal edges attached to two nodes. When these nonterminals are replaced by hypergraphs, the replacing hypergraphs are attached to the same nodes as the removed L edge. This is done by matching the external nodes of the replacing graph with the attached nodes of the nonterminal edge. (In the example, the external nodes are indicated by a blue background.)

To enable matching, we need an order on the attached nodes as well as the external nodes, which shows up as the numbers 1 and 2 in the figure and is reflected in the definition of hypergraphs (Def. 2.14 on page 18): Both the image of the attachment function att, and the external nodes ext, are defined to be ordered sequences rather than sets. ■

The forward and backward application of production rules $\sigma \rightarrow \mathcal{G}$ is formalized using hyperedge replacement and subgraph abstraction, respectively. In this thesis, I usually refer to forward application as **concretization** and backward application as **abstraction**, since in our application domain, nonterminal edges represent abstracted data structures, whereas terminal edges represent concrete pointers in the heap.

Forward application

Let $\mathcal{H}$ be a hypergraph and $\sigma \rightarrow \mathcal{G}$ be a production rule. To apply the rule forward, we need a hyperedge $e \in E_H$ with $lab(e) = \sigma$, i.e., labeled with the left-hand side of the rule. Forward application of the rule then is the **hyperedge replacement** of e with the right-hand side of the rule $\mathcal{G}$, i.e., $\mathcal{H}[e/\mathcal{G}]$.

Definition 2.26 — **Hyperedge replacement.** Let $\mathcal{H}$,$\mathcal{G}$ be hypergraphs and $e \in E_{\mathcal{H}}$ be a hyperedge with $|ext_{\mathcal{G}}| = \text{rk}(lab(e))$. Let (w.l.o.g.) $V_{\mathcal{H}} \cap V_{\mathcal{G}} = \emptyset$. The replacement of e by $\mathcal{G}$ in $\mathcal{H}$, $\mathcal{H}[e/\mathcal{G}] =: \mathcal{J}$, is the hypergraph

$$(V_{\mathcal{J}}, E_{\mathcal{J}}, att_{\mathcal{J}}, lab_{\mathcal{J}}, ext_{\mathcal{J}}, typ_{\mathcal{J}}, isnull_{\mathcal{J}})$$

where

- $V_{\mathcal{J}} = V_{\mathcal{H}} \cup (V_{\mathcal{G}} \setminus ext_{\mathcal{G}})$
- $E_{\mathcal{J}} = (E_{\mathcal{H}} \setminus \{e\}) \cup E_{\mathcal{G}}$
- $lab_{\mathcal{J}} = (lab_{\mathcal{H}} \cup lab_{\mathcal{G}}) \upharpoonright E_{\mathcal{J}}$
- $att_{\mathcal{J}} = (att_{\mathcal{H}} \upharpoonright (E_{\mathcal{H}} \setminus \{e\})) \cup rename \circ att_{\mathcal{G}}$
- $ext_{\mathcal{J}} = ext_{\mathcal{H}}$
- $typ_{\mathcal{J}} = (typ_{\mathcal{H}} \cup typ_{\mathcal{G}}) \upharpoonright V_{\mathcal{J}}$
- $isnull_{\mathcal{J}} = (isnull_{\mathcal{H}} \cup isnull_{\mathcal{G}}) \upharpoonright V_{\mathcal{J}}$

with $rename := id_{V_{\mathcal{J}}} \cup \{ext_{\mathcal{G}}(1) \mapsto att_{\mathcal{H}}(e)(1), \ldots, ext_{\mathcal{G}}(rk(e)) \mapsto att_{\mathcal{H}}(e)(rk(e))\}$, $f \cup g$ is the point-wise union of the functions, and $\upharpoonright$ denotes restriction of the domain.

The edge e is removed from $\mathcal{H}$, and $\mathcal{G}$ is inserted in its place by matching the nodes that are attached to e with the external nodes of $\mathcal{G}$. This is commonly called a gluing approach (as opposed to a connecting approach) to graph grammars [Roz99]. Note that, as discussed before, it only makes sense to replace a hyperedge e with a hypergraph $\mathcal{G}$ if e's rank is equal to the number of external nodes in $\mathcal{G}$, hence the additional requirement.

We say that $\mathcal{H}[e/\mathcal{G}]$ is derived from $\mathcal{H}$ and write $\mathcal{H} \Longrightarrow \mathcal{H}[e/\mathcal{G}]$ for such a **direct derivation** and $\overset{*}{\Longrightarrow}$ for the reflexive-transitive closure of $\Longrightarrow$.

Backward application

Intuitively, to apply a rule to $\mathcal{H}$ backward, we need to find a subgraph of $\mathcal{H}$ that is isomorphic to the right-hand side of the rule. This subgraph must be a section hypergraph, i.e., an induced subgraph in the sense that if it contains nodes $n_1, \ldots, n_i$, it must also contain all hyperedges in $\mathcal{H}$ that are attached exclusively to nodes from $n_1, \ldots, n_i$. Otherwise the result of replacing the subgraph with a single nonterminal edge would not be well-defined, since we would remove said nodes from the graph while retaining an edge that is attached to some of the removed nodes. We must also not lose external nodes through abstraction. These requirements are formalized in the definition of **subgraph abstraction**.

Definition 2.27 — **Subgraph abstraction.** Let $\mathcal{H} \in \mathbf{HG}_{\mathrm{T},\Sigma}$, $W \subseteq V_{\mathcal{H}}$ and $\mathcal{G} = \mathcal{H} \times W$ be a section hypergraph. Let $\sigma \in \Sigma_{NT}$, and $ext_{match} \in (V_{\mathcal{G}})^{\mathrm{rk}(\sigma)}$ be a sequence of nodes. We require further $ext_{\mathcal{G}} \setminus \mathrm{bound}(\mathcal{G},\mathcal{H}) = \emptyset$, i.e., the only external nodes in $\mathcal{G}$ are on the boundary between $\mathcal{H}$ and $\mathcal{G}$ (cf. Def. 2.20). We assume $e_{new} \notin E_{\mathcal{H}}$. The abstraction of $\mathcal{G}$ by σ in $\mathcal{H}$, written $\mathcal{H}[\mathcal{G}\&ext_{match}/\sigma] =: \mathcal{J}$, is the hypergraph

$$(V_{\mathcal{J}}, E_{\mathcal{J}}, att_{\mathcal{J}}, lab_{\mathcal{J}}, ext_{\mathcal{J}}, typ_{\mathcal{J}}, isnull_{\mathcal{J}})$$

where

- $V_{\mathcal{J}} = (V_{\mathcal{H}} \setminus V_{\mathcal{G}}) \cup ext_{match}$
- $E_{\mathcal{J}} = (E_{\mathcal{H}} \setminus E_{\mathcal{G}}) \cup \{e_{new}\}$
- $lab_{\mathcal{J}} = (lab_{\mathcal{H}} \restriction E_{\mathcal{J}}) \cup (e_{new} \mapsto \sigma)$
- $att_{\mathcal{J}} = (att_{\mathcal{H}} \restriction E_{\mathcal{J}}) \cup (e_{new} \mapsto ext_{match})$
- $ext_{\mathcal{J}} = ext_{\mathcal{H}}$
- $typ_{\mathcal{J}} = typ_{\mathcal{H}} \restriction V_{\mathcal{J}}$
- $isnull_{\mathcal{J}} = isnull_{\mathcal{H}} \restriction V_{\mathcal{J}}$

Note that the condition $ext_{\mathcal{G}} \setminus \mathrm{bound}(\mathcal{G},\mathcal{H}) = \emptyset$ ensures that external nodes cannot be lost through abstraction.

To apply a rule $\sigma \to \mathcal{G}$ backward, we find a subset $W \subseteq V_{\mathcal{H}}$ such that the section hypergraph $\mathcal{H} \times W \cong \mathcal{G}$ and does not contain external nodes that are not on the boundary. Let φ be an isomorphism between $\mathcal{H} \times W$ and $\mathcal{G}$ and $\varphi_V : V_{\mathcal{G}} \to W$ be the bijection on nodes induced by φ. Let $\langle x_1, \ldots, x_{\mathrm{rk}(\sigma)} \rangle := ext_{\mathcal{G}}$.

Backward rule application then consists in the subgraph abstraction $\mathcal{H}[\mathcal{H} \times W\&ext_{match}/e]$, where $ext_{match} := \langle \varphi_V(x_1), \ldots, \varphi_V(x_{\mathrm{rk}(\sigma)}) \rangle$.

Given a hypergraph $\mathcal{H}$ and a HRG, we can iterate this abstraction process to obtain a **full abstraction** of $\mathcal{H}$.

Definition 2.28 — **Fully abstract hypergraph.** Let $\mathfrak{G}$ be an HRG and $\mathcal{H}, \mathcal{G}$ be hypergraphs. $\mathcal{G}$ is **fully abstract** w.r.t. $\mathfrak{G}$ if no rule of $\mathfrak{G}$ can be applied backward to $\mathcal{G}$. $\mathcal{G}$ is a **full abstraction** of $\mathcal{H}$ if it is fully abstract and can be derived from $\mathcal{H}$ via a sequence of backward rule applications.

Languages of HRGs

Let us examine the semantics of HRGs in more detail. Recall that string grammars define **languages** of finite words: Those words that only contain terminal symbols and can be derived via finitely many rule applications from a dedicated start nonterminal. Analogously, HRGs define languages of finite hypergraphs.

In the following, let $\mathfrak{G}$ be an HRG and σ be a nonterminal symbol occurring in $\mathcal{G}$. Starting from a hypergraph that consists solely of a σ hyperedge and external nodes, we can try to derive concrete hypergraphs by iterated forward rule application. The set of graphs derivable in this manner constitutes the language of $\mathfrak{G}$ and σ. To make this precise, we first need to define the "start hypergraph", called the **handle** of σ [Jan+11].

Definition 2.29 — **Handle.** Let $\mathtt{T}$ be a type system and Σ be a heap alphabet consistent with $\mathtt{T}$. Let $\sigma \in \Sigma_{NT}$ be a nonterminal and $\tau := \langle t_1, \ldots, t_{\mathrm{rk}(\sigma)} \rangle \in (Dom(\mathtt{T}))^{\mathrm{rk}(\sigma)}$ be a sequence of types. The (typed) **handle** of σ and τ, $hnd(\sigma,\tau)$, is the hypergraph

$$(V_\sigma, E_\sigma, att_\sigma, lab_\sigma, ext_\sigma, typ_\sigma, isnull_\sigma)$$

where

- $V_\sigma := \{1, \ldots, rk(\sigma)\}$
- $E_\sigma := \{e_\sigma\}$
- $att_\sigma(e_\sigma) := \langle 1, \ldots, rk(\sigma) \rangle$
- $lab(e_\sigma) := \sigma$
- $ext_\sigma := \{1, \ldots, \mathrm{rk}(\sigma)\}$
- $typ(i) := t_i$ for all $i \in \{1, \ldots, \mathrm{rk}(\sigma)\}$
- $isnull_\sigma(i) := false$ for all $i \in \{1, \ldots, \mathrm{rk}(\sigma)\}$

Intuitively, the derivation of the language of $\mathfrak{G}$ and σ starts at the handle and proceeds as I described right before the previous definition. In contrast to [Jan+11], we have to include type information, because according to our definition of hypergraphs, all nodes are typed. Formally, the language is defined as follows.

Definition 2.30 — **Language of a nonterminal.** Given an HRG $\mathfrak{G}$ over $(\Sigma_V, \Sigma_S, \Sigma_{NT}, \mathrm{rk})$ and $\mathtt{T}$, a start nonterminal $\sigma \in \Sigma_{NT}$, and $\tau := \langle t_1, \ldots, t_{\mathrm{rk}(\sigma)} \rangle \in (Dom(\mathtt{T}))^{\mathrm{rk}(\sigma)}$, we define the **language** of $\mathfrak{G}$, σ, and τ written $\mathcal{L}(\mathfrak{G},\sigma,\tau)$, as

$$\mathcal{L}(\mathfrak{G},\sigma,\tau) := \{\mathcal{H} \in \mathbf{HG}^0_{\mathtt{T},\Sigma} \mid hnd(\sigma,\tau) \overset{*}{\Longrightarrow} \mathcal{H}\}$$

■ Example 2.31 — **The language of all doubly-linked lists.** Recall the list grammar from Ex. 2.8 on page 14. The handle $hnd(\mathtt{L},\langle elem,elem \rangle)$ and (part of) the language generated from the handle are shown in Fig. 2.5 on page 16. More generally, all (finite) doubly-linked lists can be generated from $hnd(\mathtt{L},\langle elem,elem \rangle)$ using that grammar. ■

We can generalize this from handles to all hypergraphs: The language of an (abstract) hypergraph is the set of all concrete hypergraphs derivable from that graph.

Definition 2.32 — **Language of an abstract hypergraph.** The **language of an abstract hypergraph** w.r.t. a grammar $\mathfrak{G}$, $\mathcal{L}(\mathfrak{G},\mathcal{H})$, is defined by

$$\mathcal{L}(\mathfrak{G},\mathcal{H}) := \{\mathcal{G} \in \mathbf{HG}^0_{\mathtt{T},\Sigma} \mid \mathcal{H} \overset{*}{\Longrightarrow} \mathcal{G}\}$$

2.3.4 HRGs for Data Structures

Not all HRGs are suitable for representing (abstract) data structures. In addition, not all HRGs that represent data structures are well-behaved from an algorithmic point of view. In this section, I explain these two statements and explore which additional properties a HRG must possess to be used in the automated analysis of PL programs. The main source of this section is [Jan+11], and modulo some changes—mostly to accommodate type systems—most definitions I introduce here can be found there as well.

Throughout this section, we assume a type system $\mathtt{T}$ and a heap alphabet $\Sigma = (\Sigma_V,\Sigma_S,\Sigma_{NT}, rk)$ compatible with $\mathtt{T}$.

First note that, in general, $\mathcal{L}(\mathfrak{G},\sigma)$ is a subset of $\mathbf{HG}^0_{\mathtt{T},\Sigma}$rather than of $\mathbf{HC}^0_{\mathtt{T},\Sigma}$, i.e., the language may contain hypergraphs that are not valid heap configurations. We are not interested in such HRGs. Instead, we restrict our attention to HRGs that capture valid data structures on the heap, **data structure grammars**.

Definition 2.33 — **Data Structure Grammar.** An HRG $\mathfrak{G}$ over Σ and T is a **data structure grammar** (DSG) if for all $\sigma \in \Sigma_{NT}$, there exists a unique τ such that $\mathcal{L}(\mathfrak{G},\sigma,\tau) \subseteq \mathbf{HC}^0_{\mathrm{T},\Sigma}$.

If $\mathfrak{G}$ is a DSG, we write $nttype(\sigma) := \tau$ and $\mathcal{L}(\mathfrak{G},\sigma) := \mathcal{L}(\mathfrak{G},\sigma,\tau)$

The purpose of the uniqueness constraint is to justify speaking of the (one, unique) language of a nonterminal rather than one language out of the set of languages of a nonterminal, which will turn out to be a convenient simplification. We can always ensure that such a unique typing exists by renaming the fields of a type in case of ambiguity.

A restriction to DSGs is possible in light of the following result.

Theorem 2.34 It is decidable whether a given HRG is a DSG.

Since the focus of this thesis is not on the theoretical foundations of graph grammars, I omit the proof; the untyped version is proven in [Jan+11, Appendix A] and the adaptation to nodes with data types is straightforward. DSGs are, however, still not strong enough for our purposes, as we shall see in the remainder of this section.

Productivity

The first problem is that DSGs may contain nonterminals from which no concrete heap configuration can be derived. We would like to exclude this and hence demand that the DSGs be **productive**.

Definition 2.35 — **Productivity.** $\mathfrak{G}$ is **productive** if for all nonterminals $\sigma \in \Sigma_{NT}$ that occur in $\mathfrak{G}$, $\mathcal{L}(\mathfrak{G},\sigma) \neq \emptyset$. In other words, $\mathfrak{G}$ is productive if any abstract hypergraph over Σ represents at least one concrete hypergraph.

R We can efficiently determine the productive nonterminals of $\mathfrak{G}$: All nonterminals that have at least one rule whose right-hand side is concrete are obviously productive. Additionally, if a rule's right-hand side is already known to contain only productive nonterminals, the nonterminal on its left-hand side is productive, too. Hence we can iteratively compute the set of productive nonterminals.

Increasing Grammars

$\mathfrak{G}$ is **increasing** if for all rules $\sigma \to \mathcal{G} \in \mathfrak{G}$, either $\mathcal{G} \in \mathbf{HG}^0_{\mathrm{T},\Sigma}$ or $|E_{\mathcal{G}}| > 1$. In an increasing DSG, the right-hand side of each rule is either concrete or "bigger" than its left-hand side. This is a desirable property since it guarantees that the full abstraction of any graph is reached after a finite number of abstraction steps. As our abstract interpretation of PL programs (cf. Sections 3.2 and 4.2) is based on the full abstraction of heap configurations, we demand that our DSGs are increasing to guarantee the termination of the abstraction algorithm.

Increasingness is obviously decidable and can be easily established if we have already established productivity: Each nonincreasing chain of nonterminals $\sigma_1,\sigma_2,\ldots$ must eventually lead to a (partially) concrete heap graph in a productive DSG. We can collect these graphs $\mathcal{G},\mathcal{H},\ldots$ and add new rules $\sigma_1 \to \mathcal{G}, \sigma_1 \to \mathcal{H},\ldots$. Afterwards the non-increasing rule $\sigma_1 \to \sigma_2$ may be removed.

Local concretizability

Recall that each hyperedge e is attached to $\mathrm{rk}(lab(e))$ nodes. We shall have to talk about specific attachment points of nonterminal edges, called **tentacles**.

Definition 2.36 — **Tentacle.** Let $\sigma \in \Sigma_{NT}$, and $i \in \{1, \ldots, \mathrm{rk}(\sigma)\}$. The pair (σ, i) is called a **tentacle**.

Definition 2.37 — **Reduction and non-reduction tentacle.** Let $\mathfrak{G}$ be an HRG over T and Σ. Let (σ, i) be a tentacle. (σ, i) is a **reduction tentacle** if for all $\mathcal{H} \in L(\mathfrak{G},\sigma), \forall v \in ext_{\mathcal{H}}.out(v) = \emptyset$. Otherwise (σ, i) is a **non-reduction tentacle**.

The significance of reduction tentacles is as follows. Let $\mathcal{H}$ be an abstract heap graph that contains a hyperedge e_σ with $lab(e_\sigma) = \sigma$ and $att(e_\sigma)(i) = v_i$ for $1 \leq i \leq \mathrm{rk}(\sigma)$. Say that $\mathcal{H}$ also contains a variable edge e_x with $lab(e_x) = x$. Intuitively, it is safe to attach e_x to v_i if and only if (σ, i) is a reduction tentacle: In that case we know that whatever the concrete hypergraph represented by the σ-labeled nonterminal edge, there is no outgoing edge from v_i into this hypergraph. Thus dereferencing pointers $x.s$ will never cause us to "run into" the abstracted part of the graph. For non-reduction tentacles, on the other hand, it might be necessary to partially concretize $\mathcal{H}$ prior to following pointers $x.s$. This is captured in the following definition.

Definition 2.38 — **Violation points and admissibility.** Let $e \in E_{\mathcal{H}}$ with $lab(e) = \sigma \in \Sigma_{NT}$. The pair (e,i) is a **violation point** if (σ, i) is a non-reduction tentacle and there exists an e' with $lab(e') \in \Sigma_V$ and $att(e')(1) = att(e)(i)$. We write $\mathrm{vp}_{\mathfrak{G}}(\mathcal{H})$ for the set of all violation points of $\mathcal{H}$ w.r.t. a grammar $\mathfrak{G}$.

$\mathcal{H}$ is called **admissible** if it does not contain any violation point.

Intuitively, it is admissible to dereference a pointer such as $x.s$ even in abstract heap configurations, as long as it does not contain any violation points. This is what we will exploit in the abstract interpretation of PL. To do so we, of course, need to be able to turn inadmissible heap configurations into admissible ones. To this end, we use $\mathfrak{G}$ to concretize at the violation points. In general, there is, however, no limit to the number of concretization steps that we may need to execute to remove an violation point. This is obviously problematic from an algorithmic point of view.

Ideally, we would like to work with grammars that ensure that each violation point can be removed through a single rule application. This means that there must be a set of rules for each non-reduction tentacle that insert a concrete node between the (former) violation point and any other nonterminals that they may generate. This set of rules must be **language-preserving** as well: We must not lose any information about the abstracted data structures by removing violation points. If a DSG has such a set of language-preserving rules for each non-reduction tentacle—i.e., each possible violation point—it is called **locally concretizable**. Let us formalize this property.

Definition 2.39 — **Grammar restriction.** The σ-restriction of $\mathfrak{G}$, $\mathfrak{G}^\sigma$, is the set $\{\sigma \to \mathcal{G}\} \cap \mathfrak{G}$. We write $\overline{\mathfrak{G}^\sigma}$ for $\mathfrak{G} \setminus \mathfrak{G}^\sigma$.

Definition 2.40 — **Local conretizability.** Let $\mathfrak{G}$ be a DSG and $\langle t_1, \ldots, t_{\mathrm{rk}(\sigma)} \rangle := nttype(\sigma)$ (cf. Def. 2.33). $\mathfrak{G}$ is **locally concretizable** if for all nonterminals σ in $\mathfrak{G}$, there exist grammars $\mathfrak{G}^\sigma_1, \ldots, \mathfrak{G}^\sigma_{\mathrm{rk}(\sigma)} \subseteq \mathfrak{G}^\sigma$ such that

1. $\forall i \in \{1, \ldots, \mathrm{rk}(\sigma)\} : \mathcal{L}(\mathfrak{G},\sigma) = \mathcal{L}(\mathfrak{G}^\sigma_i \cup \overline{\mathfrak{G}^\sigma}, \sigma)$

2. $\forall i \in \{1,\dots,\mathrm{rk}(\sigma)\}, \forall(\sigma \to \mathcal{G}) \in \mathfrak{G}_i^\sigma, \forall s \in sels(t_i) \exists e \in E_\mathcal{G}.att(e)(1) = ext_\mathcal{G}(i) \wedge lab(e) = s$

The first part of the definition guarantees that the subset of rules is indeed language-preserving; the second part guarantees that we can use $\mathfrak{G}_i$ to derive all outgoing edges at (N,i) in one step.

If a DSG $\mathfrak{G}$ is locally concretizable, one concretization step per violation point is always sufficient to obtain an admissible heap configuration, because we can always restrict the concretization to the language-preserving fragment of $\mathfrak{G}$ that removes the violation point.

There are sensible DSGs that do not have this property, as will become apparent in the following example.

■ Example 2.41 The doubly-linked list HRG as presented in Fig. 2.4 is **not** locally concretizable: It only allows concretization at the head of the list. Since the last element of a list has an outgoing edge as well—its type is $\{prev\}$, as you may recall—it is a potential violation point. It is impossible to remove this violation point without turning the abstract graph at hand into (one of infinitely many) concrete graphs, thus defeating the purpose of abstraction. It is easy to solve this problem by adding a rule for concretizing at the end of a list, yielding the DSG in Fig. 2.7. ■

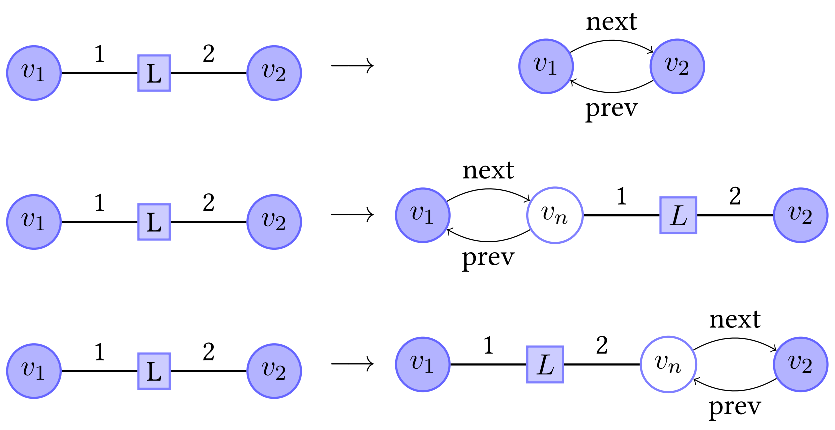


Figure 2.7.: Locally concretizable DSG for doubly-linked lists

Proposition 2.42 Every DSG can be transformed into an equivalent locally concretizable DSG.

If you are interested in the proof, I would once again like to refer you to [Jan+11].

Backward confluence

In Section 2.3.1 on page 17, we observed that the full abstraction of a hypergraph may not be unique: In general, changing the order of rule applications or selecting different subgraphs for the abstraction may lead to different full abstractions. An HRG that yields a unique full abstractions for each hypergraphs is called **backward confluent**.

■ Example 2.43 The grammar in Fig. 2.7 is not backward confluent. That can easily be fixed by adding a rule for merging adjacent nonterminals, see Fig. 2.8. ■

Proposition 2.44 — **Decidability of backward confluence.** It is decidable whether a given DSG is backward confluent.[Plu10]

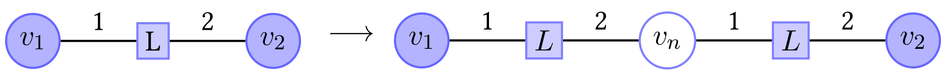


Figure 2.8.: Additional rule for backward confluent list abstraction

Unlike all the other properties we examined in this section, it is unknown whether we can transform all DSGs into equivalent backward confluent DSGs [JN14]. I shall nevertheless assume backward confluence in much of the remainder of this thesis; the reason for this is explained in context later in the text.

Heap abstraction grammars

We now have a formal understanding of all properties that we need for effective grammar-based abstraction and concretization, and hence finally a sufficient basis for defining the semantics. Before turning to the semantics, we bundle our requirements for HRGs into the final definition of this chapter.

Definition 2.45 — **(Backward-confluent) heap abstraction grammar.** An HRG $\mathfrak{G}$ is called a **heap abstraction grammar** (HAG) if it satisfies the following four additional properties.

1. $\mathfrak{G}$ is a DSG
2. $\mathfrak{G}$ is productive
3. $\mathfrak{G}$ is increasing
4. $\mathfrak{G}$ is locally concretizable

If $\mathfrak{G}$ is backward-confluent as well, we use the acronym BCHAG.

Unless stated otherwise, we always assume that grammars are HAGs throughout the remainder of the thesis.

R Jansen et al. [Jan+11] also introduce the notion of **typedness**. A typed DSG is one where for all concrete heap configuration derived from a nonterminal, the nodes on the boundary between the concrete heap configuration and the rest of the graph have the same incoming edges. In our setting with strongly typed nodes, this property is automatically satisfied by all DSGs.

CHAPTER 3

Semantics and Abstract Interpretation of Sequential Programs

In this chapter I explore programming language semantics for PL_{seq}, the sequential fragment of the programming language introduced in Section 2.2. The semantics will employ heap configurations and heap abstraction grammars, as presented in Section 2.3.

In Section 3.1, I develop a **concrete semantics** for PL_{seq}. In the concrete semantics, program states consist of a model of the call stack and concrete heap configurations that represent the heap and the local variables. The semantics is defined via hypergraph transformations. In Section 3.2, I present an **abstract interpretation** that safely approximates the concrete semantics. This abstract interpretation uses heap abstraction grammars to abstract from the shape of the heap. Having thus defined concrete and abstract semantics for PL_{seq}, we shall turn to the semantics of concurrent PL programs in Chapter 4.

3.1 Concrete Semantics of Sequential Programs

In this section I develop a structural operational semantics [Plo04] for PL_{seq}, the sequential fragment of PL, based on the hypergraph model of the previous chapter. The semantics is concrete in the sense that it is defined in terms of concrete heap configurations that do not contain nonterminals. Nevertheless, the semantics is already abstract due to the abstraction inherent in the heap configuration model: We abstract from concrete memory locations. This is in accordance with our goal of analyzing the shape of the heap, whereas the physical memory layout is not of interest for our analyses. Our hypergraph-based semantics can thus be characterized as a **storeless semantics** [Jon81; BIL03]. A key advantage of abstracting from the store already in the concrete semantics is that this step can be left out in the abstraction process. Additionally, storeless semantics are well-suited for garbage-collected languages such as PL, because they do not distinguish between heaps with isomorphic reachable parts [Rin+05].

For clarity, we split the treatment of intraprocedural semantics and interprocedural semantics into two parts, Sections 3.1.1 and 3.1.2.

3.1.1 Intraprocedural Semantics

We begin by formalizing the execution of **intraprocedural** PL_{seq} programs, i.e., programs without procedure calls. We use concrete (well-typed) heap configurations as per Def. 2.17 as the memory model for representing the execution state of a PL_{seq} program.

Notation	Meaning
$p \xmapsto{\mathcal{H}} v$	In $\mathcal{H}$, resolving pointer p yields node v
$p \xmapsto{\mathcal{H}} \bot$	In $\mathcal{H}$, pointer p cannot be resolved
$(\mathcal{H}',v : t) \leftarrow new_{\mathcal{H}}$	v is a new node (i.e., $v \notin V_{\mathcal{H}}$), $typ(v) = t$, and $\mathcal{H}'$ is obtained by adding v to $\mathcal{H}$
$(\mathcal{H}',v : t) \leftarrow null_{\mathcal{H}}$	v is the (possibly new) null node of type t if $\mathcal{H}$ already contains such a node, then $\mathcal{H}' = \mathcal{H}$
$(\mathcal{H}',e) \leftarrow new_{\mathcal{H}}(l,\langle v_1,\ldots,v_k\rangle)$	e is a new edge (i.e., $e \notin E_{\mathcal{H}}$) with $lab(e) = l$ and $att(e) = \langle v_1,\ldots,v_k\rangle$, and $\mathcal{H}'$ is obtained by adding e to $\mathcal{H}$
$\mathcal{H}[a/a']$	Replace $a \in \{V,E,lab,att,ext,typ,isnull\}$ with a'

Table 3.1.: Auxiliary notation (Informal)

Throughout this section, we shall assume that our input is a (concrete) well-typed heap configuration $\mathcal{H}$ over a set of types $\mathtt{T}$ and a heap alphabet Σ compatible with $\mathtt{T}$.

Additional notation

We start off by defining some auxiliary notation to allow more concise definitions of the commands' semantics. Recall that we use $\langle v_1,\ldots,v_k\rangle$ for denoting ordered sequences, i.e., $\langle v_1,\ldots,v_k\rangle \in V^k \subset V^*$. In addition, we shall use the notation introduced in Table 3.1.

The notation is defined formally in Fig. 3.1. Recall that we write $f \cup (k \mapsto v)$ as abbreviation for adding k to the domain of the function f and mapping it to v. Note that the definitions are all well-defined, since heap configurations may only have at most one edge per variable identifier and at most one edge per variable-selector pair.

With these additional notational conveniences available, we now turn to the definition of the semantics. We begin by defining transformation functions for each atomic statement (cf. Def. 2.1 on page 8) in turn. More concretely, we define a function $t_c : \mathbf{HC}_{\mathtt{T},\Sigma} \to (\mathbf{HC}_{\mathtt{T},\Sigma} \cup \mathbf{err})$ for each atomic statement $c \in \mathbf{Cmd}$, where $\mathbf{err}$ signals an error.

Skip statements

Let us begin with the most trivial case: The **skip** statement, which is equivalent to a noop and is mainly included in PL to enable empty **else** branches. Unsurprisingly, **skip** statements do not affect the heap configuration in our semantics:

$$t_{\mathbf{skip}}(\mathcal{H}) := \mathcal{H}$$

Variable declarations

In PL, variables need to be declared before they are used. We are going to disallow the redeclaration or shadowing of local variables. In other words, if a variable identifier x is already in scope, be it as variable or procedure parameter, the statement **var** $x : t$ is illegal. This has the advantage that each variable identifier has a fixed and easily discernible type throughout its entire scope. The incurred price is a loss of flexibility, as we neither have nested scopes nor the ability to assign different types depending on the control-flow.

$$
\begin{array}{ll}
x \overset{\mathcal{H}}{\longmapsto} v, & \exists e \in E.lab(e) = x \wedge att(e) = \langle v \rangle \\
x.s \overset{\mathcal{H}}{\longmapsto} v, & \exists e_x,e_s \in E \exists w \in V.(lab(e_x) = x \wedge lab(e_s) = s \\
 & \quad \wedge att(e_x) = \langle w \rangle \wedge att(e_s) = \langle wv \rangle) \\
p \overset{\mathcal{H}}{\longmapsto} \bot, & \nexists v \in V.p \overset{\mathcal{H}}{\longmapsto} v \\
(\mathcal{H}',v:t) \leftarrow new_{\mathcal{H}} & v \notin V \\
 & \wedge \mathcal{H}' = \mathcal{H}[V/V \cup \{v\}, typ/typ', isnull/isnull'] \\
 & \text{where } typ' = typ \cup (v \mapsto t) \\
 & \text{and} \quad isnull' = isnull \cup (v \mapsto false) \\
(\mathcal{H}',v:t) \leftarrow null_{\mathcal{H}} & (v \in V \wedge typ(v) = t \wedge isnull(v) = true \wedge \mathcal{H}' = \mathcal{H}) \\
 & \vee (v \notin V \wedge \mathcal{H}' = \mathcal{H}[V/(V \cup \{v\}), typ', isnull']) \\
 & \text{where } typ' = typ \cup (v \mapsto t) \\
 & \text{and} \quad isnull' = isnull \cup (v \mapsto true) \\
(\mathcal{H}',e) \leftarrow new_{\mathcal{H}}(l,\langle v_1,\ldots,v_k \rangle) & e \notin E \wedge \mathcal{H}' = \mathcal{H}[E/E \cup \{e\}, att/att', lab/lab'] \\
 & \text{where } att' = (att \cup (e \mapsto \langle v_1,\ldots,v_k \rangle) \\
 & \text{and} \quad lab' = (lab \cup (e \mapsto l))
\end{array}
$$

Figure 3.1.: Auxiliary notation (Formal definitions)

$$
t_{\mathbf{var}\ x:t}(\mathcal{H}) := \begin{cases} \mathcal{H}'', & \text{if } x \overset{\mathcal{H}}{\longmapsto} \bot \\ \mathbf{err} & \text{otherwise} \end{cases}
$$
$$
\mathbf{where}\ (\mathcal{H}',v:t) \leftarrow null_{\mathcal{H}} \wedge (\mathcal{H}'',e_x) \leftarrow new_{\mathcal{H}'}(x,\langle v \rangle)
$$

Figure 3.2.: Semantics of variable declarations

The restriction manifests itself in the semantics of the **var** statement, as defined in Fig. 3.2: We only add an edge if $x \overset{\mathcal{H}}{\longmapsto} \bot$. Otherwise we throw an error, denoted **err**.[1]

The new edge e_x is labeled with the variable identifier x and initially points to the null node of type t, representing a null pointer. This reflects the separation of declaration and initialization of variables in PL.[2]

R We can now see the justification for using typed null nodes in our heap model: If we did not assign x to a typed null node but rather to a generic untyped null node, we would lose the information that x is declared to be of type t. Consequently, we would lose type safety, because we would not be able to detect erroneous assignments such as $x := y$ for $y : t'$, where $t' \neq t$.

[1]This kind of error could, of course, be caught before execution through an additional static analysis.

[2]Assigning null pointers to uninitialized variables corresponds to the Java semantics, where all uninitialized references are null pointers. To come closer to C semantics, where uninitialized pointers may hold any value, we would need to distinguish between uninitialized pointers and null pointers. Real C semantics, i.e., the assignment of truly arbitrary pointer values, cannot be modeled in a sensible fashion in storeless semantics.

$$t_{x:=p}(\mathcal{H}) = \begin{cases} \mathcal{H}[att/att[e_x \mapsto \langle v\rangle]], & \text{if } \exists e_x \in E \exists v,w \in V \exists t \in \mathtt{T}. \\ & \quad p \xmapsto{\mathcal{H}} v \wedge x \xmapsto{\mathcal{H}} w \\ & \quad \wedge lab(e_x) = x \\ & \quad \wedge typ(v) = typ(w) \\ \mathbf{err}, & \text{otherwise} \end{cases}$$

Figure 3.3.: Semantics of variable assignments

$$t_{x.s:=p}(\mathcal{H}) = \begin{cases} \mathcal{H}[att/att[e_s \mapsto \langle wv\rangle]], & \text{if } \exists e_x,e_s \in E \exists v,w \in V \exists t \in \mathtt{T}. \\ & \quad p \xmapsto{\mathcal{H}} v \wedge x \xmapsto{\mathcal{H}} w \\ & \quad \wedge lab(e_x) = x \wedge lab(e_s) = s \\ & \quad \wedge att(e_s)(1) = w \\ & \quad \wedge typ(v) = typ(att(e_s)(2)) \\ \mathbf{err}, & \text{otherwise} \end{cases}$$

Figure 3.4.: Semantics of selector assignments

Assignments

Now let us consider assignments $x := p$, where x is a variable identifier and p an arbitrary pointer. The corresponding graph transformer is defined in Fig. 3.3.

We look up the node v that p points to and the edge e_x that is labeled with x. We compute the result by updating the attachment for e_x to v. If either lookup is impossible, or if the types are inconsistent, we return an error. Note that this is well-defined for all heap configurations, as they may contain at most one edge with $lab(e) = x$ for each variable x.

Similarly, we define a transformer for assignments $x.s := p$. Here we have to make sure that

- $\mathcal{H}$ contains a (variable) edge e_x for x and a (selector) edge e_s for s
- These edges are attached to the same node w. This way, we make sure that we have the right selector edge, i.e., e_s points from w to some other node
- p can be resolved, yielding node v
- The types of the node that e_s currently points to, $att(e_s)(2)$, and the new node v coincide

If all these requirements are satisfied, we update the attachment of e_s. Otherwise, we report an error; see Fig. 3.4.

R There is one special case we ignored in both assignment transformers: If $p = null$, the assignment is always allowed, even if the corresponding null node is not yet in $\mathcal{H}$. In that case, the null node would be added before executing the assignment. We need another pair of rules to handle this case, but do not give them here, because they are straightforward modification of the rules in Figs. 3.3 and 3.4.[1]

[1] If this were a text book, I would write "The definition is left as an exercise for the reader."

$$t_{x:=\mathbf{new}\ t}(\mathcal{H}) = \begin{cases} withSels(\mathcal{H}'', v_{new}, \mathtt{T}(t)) & \text{if } \exists e_x \in E.lab(e) = x \\ & \quad \wedge(\mathcal{H}', v_{new} : t) \leftarrow new_{\mathcal{H}} \\ & \quad \wedge \mathcal{H}'' = \mathcal{H}'[att/att[e_x \mapsto v_{new}]] \\ \mathbf{err}, & \text{otherwise} \end{cases}$$

$$\begin{array}{l} \mathbf{where} \\ withSels(\mathcal{H}, v_{new}, \epsilon) = \mathcal{H} \\ withSels(\mathcal{H}, v_{new}, \langle s_1 : t_1, \ldots, s_n : t_n\rangle) = \\ \quad \mathbf{let} \quad \mathcal{G} = withSels(\mathcal{H}, v_{new}, \langle s_2 : t_2, \ldots, s_n : t_n\rangle) \\ \quad\quad\quad (\mathcal{G}', v_{null} : t_1) \leftarrow null_{\mathcal{G}} \\ \quad\quad\quad (\mathcal{G}'', e_{s_1}) \leftarrow new_{\mathcal{G}'}(s_1, \langle v_{new}, v_{null}\rangle) \\ \quad \mathbf{in} \quad \mathcal{G}'' \end{array}$$

Figure 3.5.: Semantics of memory allocation

Memory allocation

Next we turn to the semantics of memory allocation statements, $x := \mathbf{new}\ t$. Allocating new memory of type t consists in

- Creating a new node v_{new} of type t
- Adding a sel-labeled edge from v_{new} to $null_{t'}$ for each selector $sel : t'$ in the type signature of type t
- Attaching the variable edge that is labeled with x to the new node

We formalize this in Fig. 3.5. Recall from Section 2.2.2 on page 10 that type systems are modeled as functions and $\mathtt{T}(t) = \langle(s_1,t_1), \ldots, (s_n,t_n)\rangle$ is the type signature of t. As we demand that the variable x be declared before it is used, we know that an x-labeled edge e_x must exist prior to the execution of the **new** statement.

$withSels$ is an auxiliary function that creates selector-to-null edges for node v and typed selectors $\langle s_1 : t_1, \ldots, s_n : t_n\rangle$.

Transition relation

Having defined the semantics of all the atomic statements, the next step is the definition of the semantics for non-atomic intraprocedural statements, i.e., sequential composition, conditionals, and **while** loops. I shall do so in the form of structural operational semantics [Plo04].

R **Denotational semantics.** Given that we have already defined the semantics of atomic statements as functions on heap configurations, it might seem more natural to give a denotational semantics for PL_{seq} than to give an operational semantics. I decided against this, because the semantics of parallel PL programs cannot be defined in terms of traditional denotational semantics. Since the parallel semantics in Chapter 4 are operational, I also chose operational semantics for the sequential setting. It is, however, easily possible to define a denotational semantics for the sequential fragment PL_{seq} using the usual domain-theoretic approach.

To define the semantics of conditionals and loops, we first need to define conditional evaluation. To this end, we define a function $condEval : \mathbf{HC}_{\mathtt{T},\Sigma} \times \mathbf{BExp} \to \{true, false\}$, where **BExp** is the set of all Boolean expressions.

$$\begin{array}{lll}
condEval(\mathcal{H},p_1 = p_2) = true & \text{iff} & (\exists v \in V.p_1 \xmapsto{\mathcal{H}} v \wedge p_2 \xmapsto{\mathcal{H}} v) \\
 & & \vee(\exists v,w \in V.p_1 \xmapsto{\mathcal{H}} v \wedge p_2 \xmapsto{\mathcal{H}} w \\
 & & \wedge isnull(v) = true \wedge isnull(w) = true) \\
condEval(\mathcal{H},p_1 \neq p_2) = true & \text{iff} & condEval(\mathcal{H},p_1 = p_2) = false \\
condEval(\mathcal{H},b_1 \wedge b_2) = true & \text{iff} & \{true\} = \{condEval(\mathcal{H},b_1), condEval(\mathcal{H},b_2)\} \\
condEval(\mathcal{H},b_1 \vee b_2) = true & \text{iff} & true \in \{condEval(\mathcal{H},b_1), condEval(\mathcal{H},b_2)\}
\end{array}$$

R The second part of the definition of $condEval(\mathcal{H},p_1 = p_2)$ ensures that comparing two null pointers always returns true, even if they are of different type. This corresponds to the semantics of `Java`. An alternative solution would be to disallow comparing pointers of different types altogether.

Now we are ready to define the transition relation for intraprocedural statements, $\rightarrow$. Each rule relates two program states, where the first component of each state is either the remaining program to be executed or $\downarrow$ to signify termination, and the second component is the current heap graph or **err** in case of failure. Formally,

$$\rightarrow \subseteq (\mathbf{Cmd} \times (\mathbf{HC}_{\mathrm{T},\Sigma} \cup \{\mathbf{err}\})) \times (\mathbf{Cmd} \cup \{\downarrow\} \times (\mathbf{HC}_{\mathrm{T},\Sigma} \cup \{\mathbf{err}\}))$$

The inference rules are listed in Fig. 3.6 on the next page. The first four rules—the inference rules for skip statements, variable declarations, assignments, and memory allocation—are derived from the transformation functions for atomic statements in the obvious way. The next six rules define the semantics of composite programs. The final two rules formalize short-circuited error handling. We opt for a small-step instead of a big-step semantics to lay the foundations for the data-flow analysis in Chapter 5.

R This is admittedly not a true small-step semantics, because we short-circuit in the case of errors and evaluate even compound conditionals in single steps. In addition, our "atomic" commands are already not atomic on any realistic machine, as we have included the allocation of memory for whole records in the set of atomic commands. This choice of granularity will, however, turn out to be convenient when we turn to data-flow analysis in Chapter 5.

Note that the execution relation is deterministic. The proof of this fact is straight-forward and omitted. We write $\xrightarrow{*}$ for the reflexive-transitive closure of $\rightarrow$. We say that a program c run on initial heap $\mathcal{H}$ terminates with heap $\mathcal{H}'$ if $(c, \mathcal{H}) \xrightarrow{*} (\downarrow, \mathcal{H}')$.

■ Example 3.1 Fig. 3.7 on the facing page illustrates the intraprocedural semantics for a simple example program that prepends an item to a list. We assume that we start execution with a four-element list, whose elements are accessible via `head` and `last` pointers. This list is depicted at the top of Fig. 3.7. We then execute four statements, displayed between the graphs, which gradually transform the graph according to the intraprocedural semantics as defined up to Fig. 3.6, yielding a list with an additional element at the front. ■

3.1.2 Interprocedural Semantics

In this section we shall develop the semantics of call statements, thus arriving at a complete operational semantics for PL_{seq}.

To this end, we need to model the **call stack**. The details of the call stack implementation differ between instruction sets, operating systems, programming languages, and compilers,

$$\frac{}{(\mathbf{skip}, \mathcal{H}) \rightarrow (\downarrow, \mathcal{H})} \qquad \frac{t_{\mathbf{var}\ x:t}(\mathcal{H}) = \mathcal{H}'}{(\mathbf{var}\ x : t, \mathcal{H}) \rightarrow (\downarrow, \mathcal{H}')}$$

$$\frac{p_1 \neq null \qquad t_{p_1 := p_2}(\mathcal{H}) = \mathcal{H}'}{(p_1 := p_2, \mathcal{H}) \rightarrow (\downarrow, \mathcal{H}')} \qquad \frac{t_{x := \mathbf{new}\ t}(\mathcal{H}) = \mathcal{H}'}{(x := \mathbf{new}\ t, \mathcal{H}) \rightarrow (\downarrow, \mathcal{H}')}$$

$$\frac{c_1 = \downarrow}{(c_1; c_2, \mathcal{H}) \rightarrow (c_2, \mathcal{H})} \qquad \frac{(c_1, \mathcal{H}) \rightarrow (c_1', \mathcal{H}')}{(c_1; c_2, \mathcal{H}) \rightarrow (c_1'; c_2, \mathcal{H}')}$$

$$\frac{condEval(\mathcal{H}, b) = true}{(\mathbf{if}\ b\ \{c_1\}\ \mathbf{else}\ \{c_2\}, \mathcal{H}) \rightarrow (c_1, \mathcal{H})} \qquad \frac{condEval(\mathcal{H}, b) = false}{(\mathbf{if}\ b\ \{c_1\}\ \mathbf{else}\ \{c_2\}, \mathcal{H}) \rightarrow (c_2, \mathcal{H})}$$

$$\frac{condEval(\mathcal{H}, b) = true}{(\mathbf{while}\ b\ \{c\}, \mathcal{H}) \rightarrow (c; \mathbf{while}\ b\ \{c_1\}, \mathcal{H})} \qquad \frac{condEval(\mathcal{H}, b) = false}{(\mathbf{while}\ b\ \{c\}, \mathcal{H}) \rightarrow (\downarrow, \mathcal{H})}$$

$$\frac{t_c(\mathcal{H}) = \mathbf{err}}{(c, \mathcal{H}) \rightarrow (\downarrow, \mathbf{err})} \qquad \frac{}{(c, \mathbf{err}) \rightarrow (\downarrow, \mathbf{err})}$$

Figure 3.6.: Small-step semantics for intraprocedural programs.

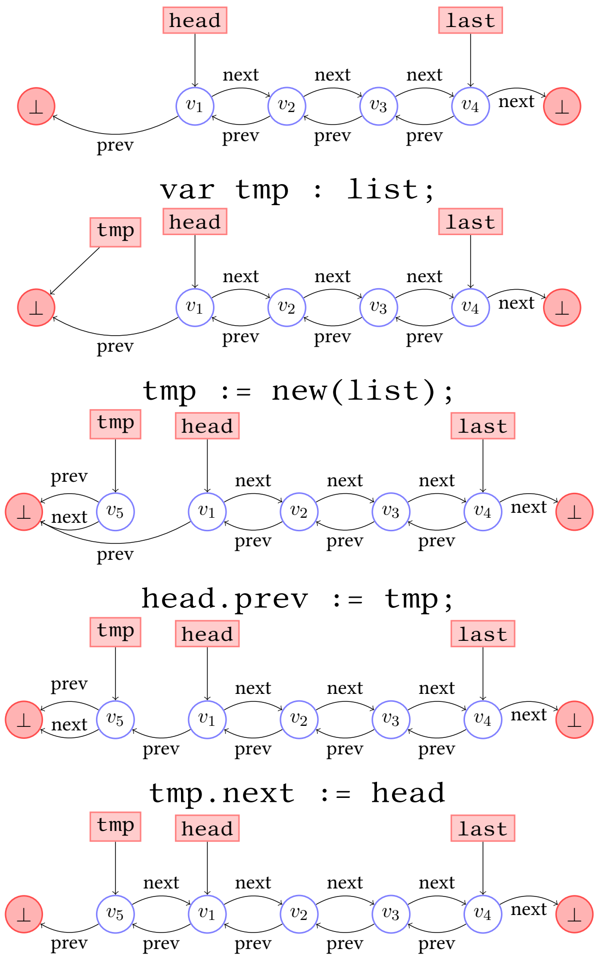

```
var tmp : list;
tmp := new(list);
head.prev := tmp;
tmp.next := head
```

Figure 3.7.: Application of the small-step semantics: Prepending an item to a list.

but for our purposes it suffices to have an abstract model of the call stack. A call stack consists of one **stack frame** or **activation record** per active procedure call. Let us briefly think about the information that must be stored in each frame [Aho+06, Section 7.2]:

- The call parameters (if any)
- The return address
- The local variables of the procedure (if any)

Instead of modeling the return address, we shall keep track of the remaining program statements of the procedure, if any, or $\downarrow$. This will make it trivial to reuse the intraprocedural semantics, where the program states were also defined in terms of remaining statements. As before, we use an integrated model of the heap and local variables: For each variable x in scope, we add an edge of degree 1 to the heap that is labeled with x and attached to the corresponding node. Consequently, each stack frame in our model consists of

- The command (sequence) c that is still to be executed, or $\downarrow$ to indicate termination
- A heap configuration that models the parameters, local variables and the part of the heap that is reachable from the parameters, or **err** in case of an error

Formally, we obtain the following stack representation.

$$\mathbf{Stk} := ((\mathbf{Cmd} \cup \{\downarrow\}) \times (\mathbf{HC}_{\mathrm{T},\Sigma} \cup \mathbf{err}))^{+}$$

Recall from Section 2.1 that we write $x_1 :: \langle x_2, \ldots, x_n \rangle$ for destructuring the sequence $\langle x_1, x_2, \ldots, x_n \rangle$ into its head and tail. This becomes useful in the formalization of the interprocedural semantics, which operates on the topmost entries of the stack.

Intuitively, each procedure call enlarges the stack by one entry. This entry is initialized with the body of the called procedure and the procedure's local view of the heap, i.e., the **reachable fragment** of the heap w.r.t. the arguments of the called procedure (formally defined in Def. 3.3 later in this section). This reachable fragment is the only part of the heap that the called procedure can possibly access or modify, so it is sufficient to keep this cut-down heap to define the semantics.

You might be wondering whether it is a good idea to operate on a stack of local heaps rather than on one global heap. After all, the call stack is an entirely separate issue from the heap. Having just one global heap seems appealing especially with the implementation in mind: Given a heap configuration $\mathcal{H}$ of size $\mathcal{O}(n)$ and a call stack of size $\mathcal{O}(m)$, the stack-of-local-heaps representation will need space $\mathcal{O}(n \cdot m)$ in the worst case, while keeping just one global heap will bring this down to $\mathcal{O}(n + m)$, as each of the $\mathcal{O}(m)$ stack frames only needs $\mathcal{O}(1)$ additional space for each of the $\mathcal{O}(1)$ parameters and variables used in the corresponding procedure.

There are, however, multiple reasons to opt for a procedure-local heap representation. First, this simplifies handling of the stack variables: Each heap graph only needs to contain the variable hyperedges for the currently active procedure, which is especially helpful because we allow recursion: We get correct static scoping almost for free.

Second, and more importantly, this choice improves modularity, by allowing local reasoning much like the separating conjunction of separation logic. By only keeping the local heap fragment in the program state, we can effectively reason about the local effects of the procedure in isolation. This does not impair correctness, since it is impossible for the procedure to infer with a fragment of the heap that it cannot reach from its parameters. This observation—that the analysis of procedure-local heap fragment suffices—is at the heart of our analysis method. It also turns out that the fear of increased memory complexity is largely unwarranted, since in our context, we actually only need to consider stacks of size at most two, as we shall see in Chapter 5.

Setting the stage for the interprocedural semantics

To define an interprocedural semantics, we need a context that provides access to all procedures. Thus all the following rules shall be parameterized by a program representation $P \in \mathbf{Progs}$. The judgments in this section are thus of the form $P \vdash stk_1 \Rightarrow stk_2$, to be read as "given the program P, the stack of commands and heaps stk_1 may be transformed into the stack stk_2 in a single step". Consequently,

$$\Rightarrow \subseteq \mathbf{Progs} \times \mathbf{Stk} \times \mathbf{Stk}$$

We use the double arrow to distinguish the interprocedural semantics from the intraprocedural semantics from the previous section. The reflexive-transitive closure is now denoted by $P \vdash stk_1 \overset{*}{\Rightarrow} stk_2$.

Before going on, you should thus briefly (re-)familiarize yourself with the mathematical program representation as developed in Section 2.2.2 on page 10, because we shall make use of the functions $procs$, $types$, $params$, $body$, $names$, and $typesig$ defined there to select the various relevant parts of $P \in \mathbf{Progs}$.

Lifting the intraprocedural semantics

The first thing to note is that the intraprocedural semantics can easily be lifted to the interprocedural, stack-based semantics, simply by applying its rules to the top element of the stack.

$$\frac{(c, \mathcal{H}) \to (c', \mathcal{H}')}{P \vdash (c, \mathcal{H}) :: stk \Rightarrow (c', \mathcal{H}') :: stk} \qquad \frac{(c, \mathcal{H}) \to (c', \mathbf{err})}{P \vdash (c, \mathcal{H}) :: stk \Rightarrow (c', \mathbf{err}) :: stk}$$

We would also like to short-circuit the interprocedural semantics in case of an error, so we add a rule

$$\frac{}{P \vdash (c, \mathbf{err}) :: stk \Rightarrow \langle(\downarrow, \mathbf{err})\rangle}$$

Towards call and return semantics: An example

The treatment of call and return statements is more interesting. Before attempting a formalization, let us have look at a simple example. Fig. 3.8 on the following page contains a brief example program that receives pointers to the head of two lists as well as one additional pointer into the first list, `node1`, as inputs. It passes the `node1` pointer to the `prepend` procedure, which creates a new list element and adds it in front of the node it receives as argument, `node`. Fig. 3.9 on page 41 illustrates the interprocedural semantics of this program, assuming (arbitrarily) that it is called on lists of length 3 and 2, respectively, and that the `node1` pointer points to the second element.

1. At call site—that is, in the `main` procedure—the heap consists of two lists, accessible via the head and node pointers. (top left)
2. Upon the call, `node1` is renamed to `node` (parameter passing) and the heap is truncated to the reachable fragment w.r.t. `node`. This is the local view of the heap argued for above. In addition, u_1 and u_2 are marked as external nodes (indicated by

```
procedure main(head1 : elem, node1: elem, head2 : elem) is
  prepend(node1)
procedure prepend(node : elem) is
  var tmp : node;
  tmp := new(node);
  node.prev := tmp;
  tmp.next := node
type elem is
  prev : elem;
  next : elem
```

Figure 3.8.: Prepending an element to a list

the blue background) to remember that they form the border between the local view of the heap and the rest of the heap. (This information is not necessary in the concrete semantics, but will be in the abstract semantics: Nodes on the boundary must not be lost through abstraction, or we cannot embed the local heap into the global heap upon procedure return. Abstraction was defined such that it cannot be applied to external nodes, so marking the boundary as external ensures that it is indeed not abstracted. As the abstract semantics are defined in terms of the concrete semantics, we already mark the boundary nodes as external in the concrete semantics.) The thus annotated heap fragment is placed on top of the call stack (not displayed).

3. Now we apply the intraprocedural semantics to the local view of the heap. This results in a list with an additional node at the front of the list, pointed to by `tmp`.
4. Before procedure return, all local variables are discarded, since they go out of scope at this point.
5. At return to `main`, we merge the two top-most stack entries, i.e., the heaps for `main` and `prepend`. To this end, we recompute the fragment of the heap that was passed to `prepend`—i.e., the first list—and the border between the fragment and the rest of the graph.[1] Apart from the border, the fragment is removed from `main`'s heap and replaced with the heap that resulted from executing `prepend`, hence introducing the new list element into the heap at call site. The same gluing approach that is used in hyperedge replacement (cf. Def. 2.26 on page 23) is used: The caller and the callee heap are glued together at the boundary.
   It is not possible to simply throw out the entire fragment at call site and take the union with the callee graph, because we would then lose the `head1` and `node1` variable edges.

Reachable fragments

We now want to formalize this process. We start with the formalization of reachable fragments.

Throughout the remainder of this section, let $P \in \mathbf{Progs}$ and $\mathbf{Var}_P$ denote the variables occurring in that program. We further assume that $\mathbf{Var}_P \subseteq \Sigma_V$.

Definition 3.2 — **Access path and reachability.** Let $\mathcal{H} \in \mathbf{HC}^0_{\mathrm{T},\Sigma}$, $x \in \mathbf{Var}_P$, and $v \in V_{\mathcal{H}}$. The sequence $\langle v_1, \dots, v_n \rangle \in (V_{\mathcal{H}})^n$ is called an **access path** from x to v if all of the following conditions hold

[1]In an implementation, we would, of course, handle this more cleverly

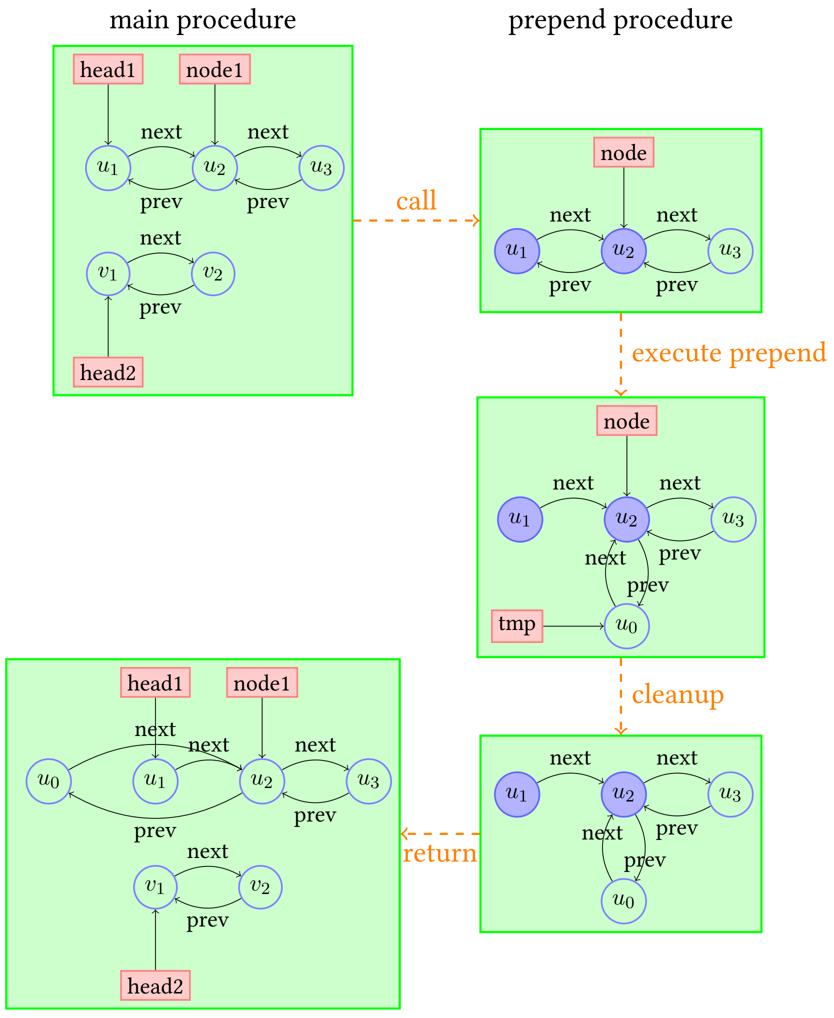


Figure 3.9.: Example: Semantics of call and return

- $v_1 = att_{\mathcal{H}}(e_x)(1)$, where e_x is the unique edge in $E_{\mathcal{H}}$ with $lab(e_x) = x$
- $v_n = v$
- For all $1 \leq i < n$, there exists en $e \in E_{\mathcal{H}}$ with $att_{\mathcal{H}}(e) = \langle v_i, v_{i+1} \rangle$

If there is an access path from x to v, we say v is **reachable** from x and write $reachable(x,v)$.

Definition 3.3 — **Reachable fragment.** Let $\mathcal{H} \in \mathbf{HC}_{\mathrm{T},\Sigma}$ and $X \in 2^{\mathbf{Var}_P}$. The **reachable fragment** of $\mathcal{H}$ w.r.t. X is defined as the section hypergraph $\mathcal{H} \times W$, where

$$W := \{v \in V_{\mathcal{H}} \mid \exists x \in X . reachable(x,v)\}$$

We write $reachable(\mathcal{H},X)$ for the reachable fragment of $\mathcal{H}$ w.r.t. X.

Less formally, the reachable fragment w.r.t. a set of variables is precisely the part of the heap that we can access via arbitrarily long chains of pointer dereferences, starting from any of the variables in the set. In particular, the reachable fragment w.r.t. a procedure's parameters is precisely the part of the heap that the procedure can access and thus potentially modify. Our procedure-local semantics shall thus be defined in terms of reachable fragments.

Cutpoints and the boundary of reachable fragments

One obstacle in the definition of procedure-local semantics are **cutpoints**. Cutpoints are those nodes on the boundary between caller and local callee heap that are not directly pointed to by a procedure parameter. As such, they are those nodes that have to be properly glued to the callee heap upon procedure return [Rin+05; Kre+13].

Definition 3.4 — **Cutpoint.** Let $\mathcal{H} \in \mathbf{HC}^0_{\mathrm{T},\Sigma}$, $X \in 2^{\mathbf{Var}_P}$, and $\mathcal{G}$ be the reachable fragment of $\mathcal{H}$ w.r.t. X. A node $v \in V_\mathcal{H}$ is a **cutpoint**, if all of the following conditions hold

- $v \in \mathrm{bound}(\mathcal{H},\mathcal{G})$, i.e., v is on the section boundary between $\mathcal{H}$ and $\mathcal{G}$
- v is not directly pointed to by a variable in X, i.e., for all $e \in E_\mathcal{G}$ with $lab(e) \in X$, $att_\mathcal{G}(e) \neq v$

■ Example 3.5 — **Cutpoints and boundaries.** In Fig. 3.9 on page 41, the boundary consisted of both u_1 and u_2, whereas u_1—not being an argument to `prepend`—was the sole cutpoint. ■

From the semantics point of view, cutpoints are easy to deal with. In fact, we just handle all nodes on the section boundary—i.e., both cutpoints and call arguments—uniformly: We mark them as external nodes to keep track of them throughout the execution of the procedure.[1]

We have to take care to do this in a deterministic fashion, though, to know which node in the call heap to match against which node of the callee. This is, however, not a problem. In the following, let the **distance** from a variable x to a node v be the minimal number of edges that we have to follow from the node pointed to by x to node v. (Hence the distance is 0 for the node pointed to by x itself, 1 for the nodes directly reachable from that node via selector edges, etc.)

Definition 3.6 — **Node order.** Let $\mathcal{H} \in \mathbf{HC}_{\mathrm{T},\Sigma}$. We define a total order $<_V$ on the nodes in $V_\mathcal{H}$ as follows. Let $v,w \in V_\mathcal{H}$.

- $v <_V w$ if the shortest path from a variable to v is shorter than the shortest path from a variable to w
- $v <_V w$ if the distance d from the nearest variables to v and w is the same, but the minimal variable at distance d to v is smaller (w.r.t. alphanumeric ordering on identifiers) than all variables at distance d to w
- $w <_V v$ otherwise

Since we operate in a garbage-collected model—that is, all memory that is not reachable from a variable is discarded—this is well-defined.

Definition 3.7 — **Ordered boundary.** Let $\mathcal{H} \in \mathbf{HC}_{\mathrm{T},\Sigma}$ and $\mathcal{G} = \mathcal{H} \times W$ for some W. The **ordered boundary** of $\mathcal{H}$ and $\mathcal{G}$, $\mathrm{obound}(\mathcal{H},\mathcal{G})$, is the unique node sequence that contains exactly the nodes in $\mathrm{bound}(\mathcal{H},\mathcal{G})$ and is ordered according to $<_V$.

Call semantics

We are finally ready to formalize call and return. We begin with the call semantics. We need one auxiliary function, namely for renaming the arguments of the procedure call to

[1]Allowing arbitrarily many cutpoints may unfortunately result in an infinite state space even under abstraction, because external nodes cannot be abstracted. We will come back to this point in the next section.

$$\frac{\begin{array}{c}\pi = procs(P)(p) \qquad lab' = rename(\langle x_1,\ldots,x_n\rangle, names(\pi)) \circ lab \\ \mathcal{H}_1 = \mathcal{H}[lab/lab'] \qquad\qquad \mathcal{H}_2 = reachable(\mathcal{H}_1, names(\pi)) \\ \mathcal{H}_3 = \mathcal{H}_2[ext/\,\mathrm{obound}(\mathcal{H}_1,\mathcal{H}_2)\cdot ext] \\ \mathcal{H}_4 = \mathcal{H}_3[E/E \setminus \{e \mid lab(e) \in \Sigma_V \setminus names(\pi)\}]\end{array}}{P \vdash (\textbf{call}\ p(x_1,\ldots,x_n); c, \mathcal{H}) :: stk \Rightarrow (body(\pi), \mathcal{H}_4) :: (\textbf{call}\ p(\ldots); c, \mathcal{H}) :: stk}$$

Figure 3.10.: The call semantics: We rename the parameters, compute the reachable fragment, and add the boundary to the external nodes.

the called procedure's formal parameters.

$$rename(\langle x_1,\ldots,x_n\rangle,\langle y_1,\ldots,y_n\rangle)(x) = \begin{cases} y_i, & \text{if } x = x_i \\ x, & \text{otherwise}\end{cases}$$

Now the call consists in

1. Composing the rename function with the labeling function
2. Cutting down the graph to the reachable fragment w.r.t. the formal parameters of the procedure
3. Prepending the ordered boundary between the caller heap and the reachable fragment to the reachable fragment's external nodes.
4. Removing all variables other than the parameters
5. Placing the called procedure's body together with the thus modified heap onto the call stack

R The reachable fragment may already contain external nodes that were added further down in the call hierarchy. For that reason we cannot simply replace ext but rather add the boundary in front of the external nodes. The decision to add them at the front rather than the back is arbitrary.

The inference rule for procedure calls, shown in Fig. 3.10, reflects these 4 steps, where the graphs resulting from steps 1 to 4 are named $\mathcal{H}_1$ to $\mathcal{H}_4$. (Recall that the concatenation of sequences seq_1, seq_2 is denoted $seq_1 \cdot seq_2$.)

It is worth noting several things at this point.

- Step 3 (removing excess variables) is necessary to capture static scoping rules: Only those variables are in scope (i.e. in the heap graph) that are local to the current procedure, which is correct, since PL does not have global variables.[1]
- The definition captures call-by-value semantics: Assignments to procedure parameters are possible, as such assignments are handled just like ordinary variable assignments by the intraprocedural semantics, but this reassignment is local and not reflected at call site.
- The call statement is not discarded but rather carried over to the successor state. This is necessary to be able to define the return semantics, as we shall soon see.

R There is one special case that is not covered by the call rule: If a call statement is the last statement in a block, the rule cannot be applied, because the state does not match.

[1]This is not an inherent limitation of the approach, though. If one were to incorporate global variables into PL, one could e.g. tag the corresponding variable edges and demand that tagged edges be kept as long as the nodes they point to are in the graph.

$$\frac{\begin{array}{c}\mathcal{H}_{sub} = reachable(\mathcal{H}_{caller}, \{x_1, \ldots, x_n\}) \\ \mathcal{H}_{drop} = \mathcal{H}_{sub} - (H_{sub} \times \text{bound}(\mathcal{H}, \mathcal{H}_{sub})) \\ \mathcal{H}_{cleaned} = \mathcal{H}_{exit}[E/E \setminus \{e \in E \mid lab(e) \in \Sigma_V\}] \\ \mathcal{H}_{merged} = (\mathcal{H}_{caller} - \mathcal{H}_{drop}) \cup \mathcal{H}_{cleaned} \\ \mathcal{H}_{final} = reachable(\mathcal{H}_{merged}, \{lab(e) \mid e \in E_{\mathcal{H}_{merged}} \wedge lab(e) \in \Sigma_V\})\end{array}}{P \vdash (\downarrow, \mathcal{H}_{exit}) :: (\textbf{call } p(x_1, \ldots, x_n); c, \mathcal{H}_{caller}) :: stk \Rightarrow (c, \mathcal{H}_{final}) :: stk}$$

Figure 3.11.: Return semantics

We can, of course, easily get around this by adding a rule such as the following.

$$\frac{}{P \vdash (\textbf{call } p(x_1, \ldots, x_n), \mathcal{H}) :: stk \Rightarrow (\textbf{call } p(x_1, \ldots, x_n); \textbf{skip}, \mathcal{H}) :: stk}$$

This rule allows appending a **skip** statement to a single call statement. This does not add any behavior, since the skip statement's semantics is the identity function, but makes it possible to apply the rule for call statements defined earlier.

Return semantics

Now for the return semantics. When a procedure's body has been executed completely, we end up in a state $(\downarrow, \mathcal{H}') :: stk$. If stk is empty, this signifies termination of the program. I.e., an interprocedural program P run on initial heap $\mathcal{H}$ terminates with heap $\mathcal{H}'$ if $P \vdash \langle (body(procs(P)(main)), \mathcal{H}) \rangle \overset{*}{\Rightarrow} \langle (\downarrow, \mathcal{H}') \rangle$.

If, however, the remaining stack is non-empty, we need to execute a procedure return. The key complication is the need to merge local heap views of the caller and the callee stack entries. Since the call statement is still in the second stack entry, we can recompute the reachable fragment and hence also the ordered boundary.[1]

This leads to the following return semantics, formalized in Fig. 3.11.

1. Recompute the reachable fragment w.r.t. $\langle x_1, \ldots, x_n \rangle$ ($\mathcal{H}_{sub}$)
2. Drop the boundary from the fragment. In this way we obtain the part of the caller heap that we must drop in favor of the callee heap ($\mathcal{H}_{drop}$)
3. Clean the callee heap by removing all variables ($\mathcal{H}_{cleaned}$)
4. Merge caller and callee by replacing the (outdated) caller's view of the callee's heap fragment with the result of executing the callee ($\mathcal{H}_{merged}$)
5. Finally, forget all nodes that are not reachable from a variable (H_{final})

R **Garbage collection.** Computing the reachable fragment after the hypergraph union corresponds to garbage collection: Nodes that the callee was able to access but that the caller cannot access after the return are now garbage. These are precisely the nodes that are removed through the reachable fragment computation.

This concludes the definition of the concrete interprocedural semantics of PL_{seq}. We next turn to the abstract semantics.

3.2 Abstract Interpretation of Sequential Programs

So far we have dealt with the concrete semantics of PL_{seq} programs: The heap configurations contained only variable edges and selector edges, but no nonterminal edges for

[1]When implementing the semantics it, of course, makes sense to save these rather than recomputing them at the return.

abstracting recursive data structures (such as the `L` edges from Section 2.3). Usually, we are, however, not interested in the semantics of a program on one specific input, but rather want to derive results for all sensible inputs. Even if we were only interested in the behavior on one concrete input, we would often still be out of luck, since the usual undecidability results for semantic properties apply (Rice's theorem).

We thus need abstraction both to reason about arbitrary input data and to obtain a decidable over-approximation of semantic properties such as shape safety. The abstraction I present here is based on heap abstraction grammars (HAGs) as defined towards the end of Section 2.3.3.

3.2.1 Towards an Abstract Interpretation of PL_{seq}

Our goal is to develop an abstract semantics for PL_{seq} in the spirit of abstract interpretation [CC77]. To this end, we have to define an abstract semantics that operates on sets of abstract states that safely over-approximates the concrete semantics. In this context, "safe" means that all concrete transition sequences are represented by abstract transition sequences. In general, the abstract transitions will also represent concrete transitions that are in fact impossible, which is what the term "over-approximation" refers to.

We build the abstract semantics on top of the concrete semantics: The basic idea is to wrap the concrete semantics in concretization and abstraction steps. These steps consist in the forward and backward application of production rules of an HAG. The abstract interpretation of every single command then comprises three steps:

1. Before executing a command, we use the HAG to locally concretize the abstract hypergraphs on demand to make it sufficiently concrete to apply the concrete semantics of the command—that is, to remove all violation points that stand in the way of the application of the concrete semantics. For example, we might need to access the `next` pointer of a list element that has been abstracted through an `L` nonterminal. We apply the rules of the doubly-linked list HAG to get back the `next` edge. This concretization step may result in multiple heap graphs if multiple rules are applicable.
2. We apply the concrete semantics defined in Section 3.1 to the partially concretized graphs. Thanks to the concretization step, this is well-defined.
3. We abstract the heap configurations as far as possible, i.e., we apply HAG rules backward until no rule is applicable to obtain the full abstraction of the graphs.

Unfortunately, there are several potential problems hidden in the vagueness of the above description, which is why we restrict our abstract interpretation to backward-confluent heap abstraction grammars. Let me briefly justify this restriction by pointing out why each of the properties of (backward-confluent) HAGs is necessary for a well-defined abstract interpretation of PL.

- We would like abstracted data structures to represent only concrete and consistently typed data structures, so we consider only **data structure grammar.** .
- We would like each abstract heap configuration to represent at least one concrete state, so we demand **productivity.**
- Although all PL_{seq} commands have only local effects—the maximum dereferencing depth is 1—it is not immediately clear that the concretization step consists in a fixed number of rule applications or even that it terminates at all. We demand **local concretizability** to be able to remove each violation point in a single concretization step.

- We consider only **increasing** grammars to exclude loops in the abstraction process and thus guarantee the termination of the abstraction step.
- **Backward confluence.** As I mentioned in Section 2.3.1, the abstraction result is not necessarily unique. If our grammar exhibits such ambiguity, this will not only increase the nondeterminism in the semantics, but also yield states that consist of several abstract hypergraphs that represent the same or overlapping languages, which is neither intuitive nor useful. For that reason, we restrict our attention to backward-confluent grammars.

3.2.2 Formal Definition of Abstract Semantics

Let us try to formalize the "concretization–concrete semantics–abstraction" loop. To this end, we assume a fixed HAG $\mathfrak{G} = (\sigma_i \to \mathcal{G}_i)_{i\in\{1,\dots,n\}}$ for the remainder of this section. The formal definition of on-demand-concretization and full abstraction are the key to the abstract semantics.

Concretization on demand

We define $\mathrm{conc}_{\mathfrak{G}}$, the set of **minimal admissible concretizations** of a heap graph $\mathcal{H}$ w.r.t. $\mathfrak{G}$.

For all $\sigma \in \Sigma_{NT}$, let $\mathfrak{G}^{\sigma}_{1}, \dots, \mathfrak{G}^{\sigma}_{\mathrm{rk}(\sigma)} \subseteq \mathfrak{G}^{\sigma}$ be language-preserving subgrammars of $\mathfrak{G}$ that remove violation points at attachment point $1, \dots, \mathrm{rk}(\sigma)$. (These exist because of the local concretizability of HAGs, see Def. 2.40 on page 27.)

$$\mathrm{conc}_{\mathfrak{G}}(\mathcal{H}) := \begin{cases} \{\mathcal{H}\}, & \text{if } \mathrm{vp}_{\mathfrak{G}}(\mathcal{H}) = \emptyset \\ \bigcup\{\mathrm{conc}_{\mathfrak{G}}(\mathcal{H}') \mid (e,i) \in \mathrm{vp}_{\mathfrak{G}}(\mathcal{H}) & \\ \qquad\qquad \wedge \mathcal{H}' \in concAt_{\mathfrak{G}}(e,i,\mathcal{H})\}, & \text{otherwise} \end{cases}$$
$$concAt_{\mathfrak{G}}(e,i,\mathcal{H}) := \{\mathcal{H}[e/\mathcal{G}] \mid (lab(e) \to \mathcal{G}) \in \mathfrak{G}^{\sigma}_{i}\}$$

Recall the definition of violation points on page 27. If $\mathcal{H}$ contains violation points, we pick one such point (e,i) and apply all possible concretizations from the language-preserving subgrammar $\mathfrak{G}^{\sigma}_{i}$ to remove it. Note that conc is well-defined because each concretization step at a violation point is guaranteed to reduce the number of violation points by at least one.

Full abstraction

Recall from Section 2.3.3 that abstraction is performed by applying production rules backward: Given a rule $\sigma_i \to \mathcal{G}_i$, we find a subgraph $\mathcal{H}_{sub}$ of $\mathcal{H}$ that is isomorphic to $\mathcal{G}_i$ and whose inner nodes are not external. We then perform a **partial abstraction** by replacing $\mathcal{H}_{sub}$ with the handle of σ_i, $hnd(\sigma_i)$: $\mathcal{H}[\mathcal{H}_{sub}/hnd(\sigma_i)]$. (For DSGs there is only a single type sequence τ such that $hnd(\sigma_i,\tau)$ is well-defined. Writing $hnd(\sigma_i)$ therefore does not introduce any ambiguity; see the definition of DSGs, Def. 2.33 on page 26.)

As explained before, we perform **full abstraction**, i.e., we iteratively apply abstraction steps until no rules are applicable. Formally,

$$
\begin{aligned}
applicable(\mathcal{H}, \sigma \to \mathcal{G}) &:= \begin{cases} true, & \text{if } \exists \mathcal{H}_{sub} \subseteq \mathcal{H}.(\mathcal{H}_{sub} \cong \mathcal{G} \\ & \quad \wedge ext_{\mathcal{H}_{sub}} \subseteq \mathrm{bound}(\mathcal{H}, \mathcal{H}_{sub})) \\ false, & \text{otherwise} \end{cases} \\
apply(\mathcal{H}, \sigma \to \mathcal{G}) &:= \mathcal{H}[\mathcal{H}_{sub}/hnd(\sigma_i)] \text{ where } \mathcal{H}_{sub} \in \{\mathcal{H}' \subseteq \mathcal{H} \mid \mathcal{H}' \cong \mathcal{G}\} \\
\mathrm{abst}_{\mathfrak{G}}(\mathcal{H}) &:= \begin{cases} \mathcal{H}, \text{if } \forall i \neg applicable(\mathcal{H}, \sigma_i \to \mathcal{G}_i) \\ \mathrm{abst}_{\mathfrak{G}}(apply(\mathcal{H}, \sigma_i \to \mathcal{G}_i)), \text{otherwise,} \\ \quad \text{where } i := \min\{j \mid applicable(\mathcal{H}, \sigma_j \to \mathcal{G}_j)\} \end{cases}
\end{aligned}
$$

R Note that rules are only applicable if their application does not remove external nodes, as expressed by demanding that all external nodes in the matched subgraph are on its boundary.

R *apply* is nondeterministic, and thus abst is as well. This does not matter, however, since we assume that the HRG is backward-confluent, which guarantees the uniqueness of the full abstraction, if not of the single abstraction steps. Thus we can, for example, resolve the non-determinism by always picking the first applicable rule and isomorphic subgraph that we find. This is reflected in the definition of abst.

This remark shows that, unlike concretization, full abstraction does not introduce additional non-determinism. In fact, it might even resolve some non-determinism, should two (partially) concrete heap graphs have the same abstraction.

Putting it all together

We are now ready to define the abstract operational semantics. Informally, to arrive at the abstract semantics of a command c, we want a composition "abst $\circ concrete\ semantics(c) \circ$ conc". As with the concrete semantics, we begin with an intraprocedural relation, $\xrightarrow{A}$.

$$
\frac{\mathcal{H}_0 \in \mathrm{conc}(\mathcal{H}) \quad (c, \mathcal{H}_0) \to (c', \mathcal{H}_1) \quad \mathcal{H}_2 = \mathrm{abst}(\mathcal{H}_1)}{(c, \mathcal{H}) \xrightarrow{A} (c', \mathcal{H}_2)}
$$

R $\xrightarrow{A}$ is nondeterministic, since $\mathrm{conc}(\mathcal{H})$ usually contains several partially concretized hypergraphs.

The interprocedural relation $\overset{A}{\Longrightarrow}$ exactly mirrors the concrete relation, the only difference being the additional abstraction steps. For completeness's sake, I show the modified rules in Fig. 3.12 to Fig. 3.14. Note that the call rule does not even have an additional abstraction step: We do not need an additional abstraction step, because the reachable fragment is already fully abstract. It is true that some variables might go out of scope, but these are all on the boundary of the reachable fragment and thus not subject to abstraction.

! For this semantics to be correct, it is crucial that the whole boundary between caller and callee is marked as external. This makes it impossible to abstract nodes on the boundary and thus guarantees that the boundary stays intact until the return. The return therefore works just as in the concrete case.

We write $s_1 \overset{A*}{\Longrightarrow} s_2$ for the reflexive-transitive closure of $\overset{A}{\Longrightarrow}$.

$$\frac{(c,\mathcal{H}) \xrightarrow{A} (c',\mathcal{H}')}{P \vdash (c,\mathcal{H}) :: stk \overset{A}{\Longrightarrow} (c',\mathcal{H}') :: stk} \qquad \frac{(c,\mathcal{H}) \xrightarrow{A} (c',\mathbf{err})}{P \vdash (c,\mathcal{H}) :: stk \overset{A}{\Longrightarrow} (c',\mathbf{err}) :: stk}$$

$$\frac{}{P \vdash (c,\mathbf{err}) :: stk \overset{A}{\Longrightarrow} \langle(\downarrow,\mathbf{err})\rangle}$$

Figure 3.12.: Lifted intraprocedural semantics and error handling.

$$\frac{\begin{array}{c}\pi = procs(P)(p) \qquad lab' = rename(\langle x_1,\ldots,x_n\rangle, names(\pi)) \circ lab \\ \mathcal{H}_1 = \mathcal{H}[lab/lab'] \qquad\qquad \mathcal{H}_2 = reachable(\mathcal{H}_1, names(\pi)) \\ \mathcal{H}_3 = \mathcal{H}_2[ext/\,\mathrm{obound}(\mathcal{H}_1,\mathcal{H}_2)\cdot ext] \\ \mathcal{H}_4 = \mathcal{H}_3[E/E \setminus \{e \mid lab(e) \in \Sigma_V \setminus names(\pi)\}]\end{array}}{P \vdash (\mathbf{call}\ p(x_1,\ldots,x_n); c,\mathcal{H}) :: stk \overset{A}{\Longrightarrow} (body(\pi),\mathcal{H}_3) :: (\mathbf{call}\ p(\ldots); c,\mathcal{H}) :: stk}$$

Figure 3.13.: Abstract call semantics. There is no difference to the concrete semantics, because the reachable fragment is already fully abstract.

R You may have noticed that we never formally defined reachable fragments on abstract graphs. I will omit this definition, but observe that the definition for reachability in concrete heaps (i.e., Def. 3.2) can easily be lifted to the abstract setting: Let e be a nonterminal edge and $v,w \in att(e)$. We now allow $\langle v,w\rangle$ to be contained in (abstract) access paths if there is a path from v to w in any (and hence all, Because we restricted our attention to HAGs) concrete graphs represented by e. Otherwise, we do not allow $\langle v,w\rangle$ as a subsequence of the access paths (unless, of course, there are other hyperedges connecting v and w that are concrete or that satisfy the above constraint).

Size of the abstract domain

To be able to base a static analysis on the abstract semantics, we need to guarantee the finiteness of the abstract domain—at least if we want to ensure termination of the analysis. First of all, we clearly need to identify isomorphic hypergraphs. Even if we only consider one hypergraph per isomorphism class, there are three reasons why the abstract domain of the semantics I presented here is infinite.

1. The semantics are defined in terms of stacks of arbitrary and hence potentially unbounded size.

$$\frac{\begin{array}{c}\mathcal{H}_{sub} = reachable(\mathcal{H}_{caller}, \{x_1,\ldots,x_n\}) \\ \mathcal{H}_{drop} = \mathcal{H}_{sub} - (H_{sub} \times \mathrm{bound}(\mathcal{H},\mathcal{H}_{sub})) \\ \mathcal{H}_{cleaned} = \mathcal{H}_{exit}[E/E \setminus \{e \in E \mid lab(e) \in \Sigma_V\}] \\ \mathcal{H}_{merged} = (\mathcal{H}_{caller} - \mathcal{H}_{drop}) \cup \mathcal{H}_{cleaned} \\ \mathcal{H}_{final} = reachable(\mathcal{H}_{merged}, \{lab(e) \mid e \in E_{\mathcal{H}_{merged}} \wedge lab(e) \in \Sigma_V\})\end{array}}{P \vdash (\downarrow,\mathcal{H}_{exit}) :: (\mathbf{call}\ p(x_1,\ldots,x_n); c,\mathcal{H}_{caller}) :: stk \overset{A}{\Longrightarrow} (c,\mathrm{abst}(\mathcal{H}_{final})) :: stk}$$

Figure 3.14.: Abstract return semantics. The only difference is an additional abstraction step in the end, because the cleaned heap need not be fully abstract.

2. We might be unable to bound the size of the parts of the heap that cannot be abstracted by our HRG.
3. Cutpoints cannot be abstracted, and recursive programs may contain an unbounded number of cutpoints.

As we will see in Chapter 5, the first point is actually not a problem, since we only need to consider the two top-most stack elements to define the interprocedural data-flow analysis.

The second point is often unproblematic: Most programs do not use an unbounded number of unstructured raw pointers. Instead, most data is usually organized in (possibly recursive) structures, which can often be captured completely by HRGs. Such programs may violate the (implicit) shape invariants of the data structures they use—for example, introducing a cycle into a tree—but they usually do so only locally, i.e., in an environment of constant size around the variable pointers. This environment of constant size can be used to derive a bound for the number of nodes that cannot be abstracted. If you would like to see a concrete example of the HRG-based analysis of an analysis with a bounded number of shape violations, you can have a look at the analysis of the Deutsch-Schorr-Waite algorithm in [HNR10].

The third point is the most severe. Despite the claim to the contrary in [JN14], we are not able to deal with arbitrarily many cutpoints in the data-flow analysis, because we cannot bound the size of the abstract graphs in such cases. In practice, we therefore demand an arbitrary but fixed upper bound on the number of cutpoints that the analysis is able tracks. If this bound is exceeded, the analysis fails. Since many recursive programs are in fact **effectively cutpoint-free**, our approach is still applicable to a wide range of recursive programs [Kre+13].[1]

3.2.3 Soundness of the Abstract Semantics

Intuitively, the abstract semantics is **sound** if each concrete program trace that is enabled for a set of concrete input values is included in an abstract trace that is enabled for the corresponding set of abstract input values. To formalize this intuition we use the framework of **abstract interpretation**, originally introduced by Cousot and Cousot [CC77]. We define the abstract interpretation for an arbitrary but fixed backward-confluent HAG $\mathfrak{G}$.

Recall that we defined $\Rightarrow$ and $\overset{A}{\Longrightarrow}$ as relations on states, where each state consisted of a stack of commands and (abstract) values. Observe that we can instead define the semantics via functions

$$\mathbf{nextC} : \mathbf{Cmd}^{+} \to \mathbf{Cmd}^{+} \to 2^{(\mathbf{HC}^{0} \cup \{\mathbf{err}\})^{+}} \to 2^{(\mathbf{HC}^{0} \cup \{\mathbf{err}\})^{+}}$$

in the case of the concrete semantics and

$$\mathbf{nextA} : \mathbf{Cmd}^{+} \to \mathbf{Cmd}^{+} \to 2^{(\mathbf{HC}^{0} \cup \{\mathbf{err}\})^{+}} \to 2^{(\mathbf{HC}^{0} \cup \{\mathbf{err}\})^{+}}$$

in the case of the abstract semantics. In such a formulation of the semantics,

$$\mathbf{nextC}(cs_1, cs_2, \mathfrak{H}_1) = \mathfrak{H}_2$$

[1] The notion of **effectively cutpoint-free** programs was introduced in [Kre+13]. Since I do not want to dwell on the topic of cutpoints, I refer you to that paper for an in-depth treatment of cutpoints and a discussion of how to deal with cutpoints under abstraction.

means that, if we start with call stack cs_1 and one of the stacks of heap configurations $hs \in \mathfrak{H}_1$, and end up in call stack cs_2 after a single step, then the resulting stack of heap configurations (or error information) will be one of the $hs' \in \mathfrak{H}_2$. $\mathbf{nextA}$ has the analogous meaning for stacks of abstract heaps (and error information).

Formally,

$$\begin{aligned}
\mathbf{nextC}(cs_1{,}cs_2{,}\mathfrak{H}) &:= \{hs_2 \mid hs_1 \in \mathfrak{H} \wedge \text{zip}(cs_1{,}hs_1) \Rightarrow \quad \text{zip}(cs_2{,}hs_2)\} \\
\mathbf{nextA}(cs_1{,}cs_2{,}\mathfrak{H}) &:= \{hs_2 \mid hs_1 \in \mathfrak{H} \wedge \text{zip}(cs_1{,}hs_1) \overset{A}{\Longrightarrow} \quad \text{zip}(cs_2{,}hs_2)\}
\end{aligned}$$

where

$$\begin{array}{lcl}
\text{zip}(\epsilon, \epsilon) & := & \epsilon \\
\text{zip}(\langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_n\rangle) & := & (x_1{,}y_1) :: \text{zip}(\langle x_2, \ldots, x_n\rangle, \langle y_2, \ldots, y_n\rangle)
\end{array}$$

In other words, $\mathbf{nextC}$ and $\mathbf{nextA}$ gather all the results of applying $\Rightarrow$ and $\overset{A}{\Longrightarrow}$ to any of the input configurations in $\mathfrak{H}$. For notational convenience, we define partially applied variants of the functions where the pair of control stacks has been fixed.

$$\begin{array}{lcl}
\mathbf{nextC}_{\langle cs_1{,}cs_2\rangle} & : & 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} \to 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} \\
\mathbf{nextC}_{\langle cs_1{,}cs_2\rangle}(hs) & := & \mathbf{nextC}(cs_1, cs_2, hs) \\
\mathbf{nextA}_{\langle cs_1{,}cs_2\rangle} & : & 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} \to 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} \\
\mathbf{nextA}_{\langle cs_1{,}cs_2\rangle}(hs) & := & \mathbf{nextA}(cs_1, cs_2, hs)
\end{array}$$

If cs_2 is not a possible successor call stack of cs_1, then $\mathbf{nextC}_{\langle cs_1{,}cs_2\rangle}$ will always be a constant function returning the empty set. If, however, $cs_1, cs_2, \ldots, cs_k$ is a possible sequence of call stacks when starting on a stack of heap configurations hs, then this execution trace is also reflected by the corresponding chain of semantics functions,

$$\begin{gathered}
(\mathbf{nextC}_{\langle cs_{k-1}{,}cs_k\rangle} \circ \cdots \circ \mathbf{nextC}_{\langle cs_1{,}cs_2\rangle})(\{hs\}) = \{hs'\} \\
\text{iff} \\
(cs_1{,}hs) \Rightarrow^{k-1} (cs_k{,}hs')
\end{gathered}$$

This equality holds due to the determinacy of the concrete semantics, whereas the nondeterminism of the abstract semantics also carries over to the $\mathbf{nextA}$ function.

$$\begin{gathered}
hs' \in (\mathbf{nextA}_{\langle cs_{k-1}{,}cs_k\rangle} \circ \cdots \circ \mathbf{nextA}_{\langle cs_1{,}cs_2\rangle})(\{hs\}) \\
\text{iff} \\
(cs_1{,}hs) \overset{A}{\Longrightarrow}^{k-1} (cs_k{,}hs')
\end{gathered}$$

Now observe that for each fixed pair of stacks cs_1, cs_2, the partially applied functions $\mathbf{nextC}(cs_1{,}cs_2)$ and $\mathbf{nextA}(cs_1{,}cs_2)$ are monotonic functions on the power set lattices $(2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+}, \subseteq)$ and $(2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+}, \subseteq)$, since by definition,

$$\begin{array}{ll}
 & \mathbf{nextC}(cs_1{,}cs_2)(\mathfrak{H}_1 \cup \mathfrak{H}_2) \\
= & \{hs_2 \mid hs_1 \in (\mathfrak{H}_1 \cup \mathfrak{H}_2) \wedge \text{zip}(cs_1{,}hs_1) \Rightarrow \text{zip}(cs_2{,}hs_2)\} \\
= & \{hs_2 \mid hs_1 \in \mathfrak{H}_1 \wedge \text{zip}(cs_1{,}hs_1) \Rightarrow \text{zip}(cs_2{,}hs_2)\} \\
 & \cup \{hs_2 \mid hs_1 \in \mathfrak{H}_2 \wedge \text{zip}(cs_1{,}hs_1) \Rightarrow \text{zip}(cs_2{,}hs_2)\} \\
= & \mathbf{nextC}(cs_1{,}cs_2)(\mathfrak{H}_1) \cup \mathbf{nextC}(cs_1{,}cs_2)(\mathfrak{H}_2)
\end{array}$$

(The exact same holds for $\mathbf{nextA}$.)

R If you are not familiar with the order-theoretic notions used throughout this section, you will find a brief introduction in Appendix A on page 111.

Furthermore, we can lift (full) abstraction and concretization to the power set domains, yielding a pair of functions α and γ. Recall that $\mathcal{L}(\mathfrak{G}, \mathcal{H})$ denotes all concrete hypergraphs represented by $\mathcal{H}$ (Def. 2.32 on page 25).

$$\begin{array}{lcl}
\alpha & : & 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} \to 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} \\
\alpha(\mathfrak{H}) & := & \{\langle \mathcal{G}_1, \ldots, \mathcal{G}_k \rangle \mid \langle \mathcal{H}_1, \ldots, \mathcal{H}_k \rangle \in \mathfrak{H} \wedge \bigwedge_i (\mathcal{G}_i = \mathrm{abst}(\mathcal{H}_i) \vee \mathcal{G}_i = \mathcal{H}_i = \mathbf{err})\} \\
\gamma & : & 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} \to 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} \\
\gamma(\mathfrak{H}) & := & \{\langle \mathcal{G}_1, \ldots, \mathcal{G}_k \rangle \mid \langle \mathcal{H}_1, \ldots, \mathcal{H}_k \rangle \in \mathfrak{H} \wedge \bigwedge_i (\mathcal{G}_i \in \mathcal{L}(\mathfrak{G}, \mathcal{H}_i) \vee \mathcal{G}_i = \mathcal{H}_i = \mathbf{err})\}
\end{array}$$

We obtain the abstraction of a set of concrete stacks by applying full abstraction to each heap configuration in the stack. (Note that, because we assume that $\mathfrak{G}$ is backward confluent, $\mathrm{abst}(\mathcal{H})$ is a single graph.) Conversely, we obtain the concretization of a set of abstract stacks by replacing each abstract heap configuration by a concrete one that is in the language of the abstract heap. Errors are replaced by errors.

The key observation for applying the theory of abstract interpretation to our domain is that the power set lattices together with α for abstraction and γ for concretization form a **Galois connection** (cf. Def. A.11). To show this, we first need two lemmas.

Lemma 3.8 If $\mathcal{H} \in \mathbf{HC}^0$, then $\mathcal{H} \in \mathcal{L}(\mathfrak{G}, \mathrm{abst}(\mathcal{H}))$.

Proof. Let $\langle \mathcal{R}_1, \ldots, \mathcal{R}_k \rangle$ be the sequence of rules that, applied backward, transform $\mathcal{H}$ into $\mathrm{abst}(\mathcal{H})$. Applying $\langle \mathcal{R}_k, \ldots, \mathcal{R}_1 \rangle$ forward to $\mathrm{abst}(\mathcal{H})$ yields $\mathcal{H}$. I.e., $\mathrm{abst}(\mathcal{H}) \overset{*}{\Longrightarrow} \mathcal{H}$. Hence $\mathcal{H} \in \mathcal{L}(\mathfrak{G}, \mathrm{abst}(\mathcal{H}))$. ■

Lemma 3.9 If $\mathcal{H}$ is fully abstract, then $\mathrm{abst}(\mathcal{L}(\mathfrak{G}, \mathcal{H})) = \mathcal{H}$.

Proof. Thanks to backward confluence, the abstraction result is unique. Since all graphs in $\mathcal{L}(\mathfrak{G}, \mathcal{H})$ can be abstracted to $\mathcal{H}$ by reverting the corresponding derivation sequence, this unique result must be $\mathcal{H}$. ■

Theorem 3.10 The tuple $((2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+}, \subseteq), \alpha, \gamma, (2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+}, \subseteq))$ forms a **Galois connection**.

Proof. We need to show that

1. $\forall \mathfrak{H} \in 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} . \mathfrak{H} \subseteq \gamma(\alpha(\mathfrak{H}))$
2. $\forall \mathfrak{H} \in 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+} . \alpha(\gamma(\mathfrak{H})) \subseteq \mathfrak{H}$

α and γ both map **err** to itself, so if the claims hold for stacks without error components, they clearly also hold for stacks with error components.

We show the first point for stacks without error components. Let $\mathfrak{H} \in 2^{(\mathbf{HC}^0)^+}$. Let $hs := \langle \mathcal{H}_1, \ldots, \mathcal{H}_k \rangle \in \mathfrak{H}$ be an arbitrary element of $\mathfrak{H}$. Note that $\gamma(\alpha(\{\langle \mathcal{H}_1, \ldots, \mathcal{H}_k \rangle\})) = \{\langle \mathcal{G}_1, \ldots, \mathcal{G}_k \rangle \mid \mathcal{G}_i \in \mathcal{L}(\mathfrak{G}, \mathrm{abst}(\mathcal{H}_i))\}$. Lemma 3.8 shows that for each individual stack element $\mathcal{H}_i$, $\mathcal{H}_i \in \mathcal{L}(\mathfrak{G}, \mathrm{abst}(\mathcal{H}_i))$. Hence also $\{\langle \mathcal{H}_1, \ldots, \mathcal{H}_k \rangle\} \subseteq \gamma(\alpha(\{\langle \mathcal{H}_1, \ldots, \mathcal{H}_k \rangle\}))$. Now, since $\gamma(\alpha(\mathfrak{H})) = \bigcup_{hs \in \mathfrak{H}} (\gamma(\alpha(\{hs\})))$, it follows that $\mathfrak{H} \subseteq \gamma(\alpha(\mathfrak{H}))$.

The second point is proven analogously by applying Lemma 3.9 to the individual stack components. ■

What does this characterization buy us? We can show that the abstract semantics is a **safe approximation** of the concrete semantics.

Definition 3.11 — **Safe Approximation.** Let $((L, \sqsubseteq_L),\alpha,\gamma,(M, \sqsubseteq_M))$ be a Galois connection and $f : L \to L$ be a function on the concrete domain. $g : M \to M$ is a **safe approximation** of f iff

$$\alpha(f(\gamma(m))) \sqsubseteq_M g(m)\ \forall m \in M$$

It is called the **most precise** safe approximation if the reverse also holds.

Let us think about the meaning of this definition. The abstract function is a safe approximation if it over-approximates the concrete function—that is, if the abstraction of the concrete output of the concrete function applied to the concretized input is contained in the output of the abstract function. This is illustrated by the following diagram [Nol15].

$$\begin{tikzcd}
m \arrow[rr, "\gamma"] \arrow[d, "g"] & & \gamma(m) \arrow[d, "f"'] \\
g(m) & \sqsupseteq \alpha(f(\gamma(m))) & f(\gamma(m)) \arrow[l, "\alpha"']
\end{tikzcd}$$

We are now ready to state the central theorem of this section.

Theorem 3.12 — **The abstract semantics safely approximates the concrete semantics.** For all cs_1, cs_2, $\mathbf{nextA}_{\langle cs_1,cs_2\rangle}$ safely approximates $\mathbf{nextC}_{\langle cs_1,cs_2\rangle}$.

It is intuitive that this should hold: First, the abstract semantics is just the concrete semantics applied to a set of graphs that have been concretized on demand, and on-demand concretization is language-preserving (for a locally concretizable grammar), so the abstract semantics necessarily subsumes all concrete behavior. Second, thanks to backward confluence, it does not matter whether we concretize fully (through the application of γ) or partly (in the abstract semantics), subsequent (full) abstraction, as performed both by α and the abstract semantics, must yield the same hypergraph.

To prove this formally, a series of lemmas will come in handy. In the following, I shall assume a fixed BCHAG $\mathfrak{G}$ and denote by $\mathcal{L}(hs)$ the language obtained by applying said grammar to all components of hs. I shall write $\mathrm{conc}(hs), \mathrm{abst}(hs)$ for the application of the respective function to the top element of the stack, in the former case w.r.t. the current call stack cs_1. I shall also adopt the convention to denote stacks of abstract heaps by hs, hs_i, hs', etc., and use gs, gs_j, etc., for stacks of concrete heaps.

Lemma 3.13 Let $\mathfrak{H}_1 \in 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+}$, gs_1,$gs_2 \in (\mathbf{HC}^0)^+$. If $gs_1 \in \gamma(\mathfrak{H}_1)$, $gs_1 \Rightarrow gs_2$, and $\mathbf{nextA}_{\langle cs_1,cs_2\rangle}(\mathfrak{H}_1) = \mathfrak{H}_2 \neq \emptyset$, then there exist $hs_1 \in \mathfrak{H}_1, hs_2 \in \mathfrak{H}_2$ such that $hs_1 \overset{A}{\Longrightarrow} hs_2$ and $gs_2 \in \mathcal{L}(hs_2)$.

Proof. By definition of γ, $gs_1 \in \mathcal{L}(hs)$ for some $hs \in \mathfrak{H}_1$.

Case 1: Assume the next command to be executed is intraprocedural. Let $\{hs'_1, \ldots ,hs'_k\} = \mathrm{conc}(hs)$. Clearly there exists a j such that $gs_1 \in \mathcal{L}(hs'_j)$, since $\mathfrak{G}$ is locally concretizable. Now apply the concrete semantics $\Rightarrow$ to both gs_1 and hs'_j, leading to gs_2 and hs''_j. Before that application, the parts of the graphs that are affected by this application are isomorphic, because of the local concretization.

Hence the local transformations in the two graphs caused by the application of $\Rightarrow$ are identical and the resulting subgraphs are still isomorphic after the application of $\Rightarrow$. Thus, $gs_2 \in \mathcal{L}(hs''_j)$. Subsequently, a full abstraction is applied to hs''_j, leading to stack hs^*_j. Note that $\mathcal{L}(hs^*_j) \supseteq \mathcal{L}(hs''_j)$ and $hs^*_j \in \mathfrak{H}_2$. Hence the pair $hs_1 := hs, hs_2 := hs^*_j$ has the desired property.

Case 2: The next command is a call. Clearly, language inclusion is invariant under relabeling variables, dropping variables, and marking variables as external. It is also invariant under reachable fragment computation, since $\mathfrak{G}$ is context free. The result of applying the call semantics to hs is thus the desired graph hs_2, again setting $hs_1 := hs$.

Case 3: The next case is a return. Omitted for the sake of brevity, as it can be proven along similar lines. ■

Lemma 3.14 If $\mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\{hs\}) = \emptyset$, then $\mathbf{nextC}_{\langle cs_1, cs_2 \rangle}(\gamma(\{hs\})) = \emptyset$

Proof. I show the contrapositive, i.e., if $\mathbf{nextC}_{\langle cs_1, cs_2 \rangle}(\gamma(\{hs\})) \neq \emptyset$, then $\mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\{hs\}) \neq \emptyset$.

Let gs_1,gs_2 such that $gs_1 \in \gamma(\{hs\})$ and $gs_1 \Rightarrow gs_2$. (These exist by assumption.) Note that, like in Lemma 3.13, $gs_1 \in \mathcal{L}(hs')$ for some $hs' \in \mathrm{conc}(hs)$.

Case 1: The next command in cs_1 is intraprocedural. By definition of conc, hs' does not contain violation points and the parts of gs_1 and hs' that are affected by the execution of the current command are isomorphic. Hence the concrete semantics is applicable to hs' and $hs' \Rightarrow hs''$ for some hs'', so $hs \Rightarrow hs^*$ for some hs^* and $\mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\{hs\}) \neq \emptyset$.

Case 2: The next command in cs_1 is a call statement. Then this call statement is clearly enabled for hs' (since the graphs contain the same variables as gs_1), and thus $\mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\{hs\}) \neq \emptyset$.

Case 3: The next command in cs_1 is a return statement. As hs' contains the same variables and external nodes as cs_1, the abstract return rule is enabled for hs and also in this case $\mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\{hs\}) \neq \emptyset$. ■

So far, we have not dealt with error cases. We handle them in a final lemma.

Lemma 3.15 Let $\mathfrak{H}_1 \in 2^{(\mathbf{HC}^0 \cup \{\mathbf{err}\})^+}$. If $gs_1 \in \gamma(\mathfrak{H}_1), gs_1 \Rightarrow gs_2 = \langle \mathbf{err}, \mathcal{G}_1, \dots, \mathcal{G}_k \rangle, k \geq 0$, then $\mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\mathfrak{H}_1) = \mathfrak{H}_2 \neq \emptyset$, and there exist $hs_1 \in \mathfrak{H}_1, \langle \mathbf{err}, \mathcal{H}_1, \dots, \mathcal{H}_k \rangle = hs_2 \in \mathfrak{H}_2$ such that $hs_1 \overset{A}{\Longrightarrow} hs_2$.

Proof. If $gs_1 = \langle \mathbf{err}, \mathcal{J}_1, \dots, \mathcal{J}_\ell \rangle$, the result is trivial, as both semantics yield $(\downarrow, \mathbf{err})$ in that case. Let therefore $gs_1 \in (\mathbf{HC}^0)^+$. Let $hs_1 \in \mathfrak{H}_1$ such that $gs_1 \in \mathcal{L}(hs')$ for and $hs' \in \mathrm{conc}(hs_1)$. Again, such stacks must exist for locally concretizable $\mathfrak{G}$ by definition of γ.

This time need only consider intraprocedural commands. Call and return do not produce $\mathbf{err}$ values—at worst the semantics deadlocks if the rules' preconditions are not satisfied. But clearly $hs' \overset{A}{\Longrightarrow} \langle \mathbf{err}, \mathcal{H}'_1, \dots, \mathcal{H}'_k \rangle$, since those parts of hs' and gs_1 that are relevant for the execution of cs_1 are isomorphic. By definition of $\mathbf{nextA}_{\langle cs_1, cs_2 \rangle}$, it follows that $hs_2 := \langle err, \mathrm{abst}(H'_1), \dots, \mathrm{abst}(H'_k) \rangle \in \mathfrak{H}_2$. ■

These three lemmas let us prove Theorem 3.12 in a straightforward way.

Proof of Theorem 3.12. We have to show that

$$\forall \mathfrak{H} \in 2^{(\mathbf{HC})^+}. \alpha(\mathbf{nextC}_{\langle cs_1, cs_2 \rangle}(\gamma(\mathfrak{H}))) \subseteq_M \mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\mathfrak{H})$$

Let $\mathfrak{H} \in 2^{(\mathbf{HC})^+}$ and $\mathfrak{H}' := \{hs \in \mathfrak{H} \mid \mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\{hs\}) \neq \emptyset\}$. By Lemma 3.14, it suffices to show $\alpha(\mathbf{nextC}_{\langle cs_1, cs_2 \rangle}(\gamma(\mathfrak{H}'))) \subseteq_M \mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\mathfrak{H}')$.

Consider arbitrary $hs \in \mathfrak{H}'$ and $gs_1 \in \gamma(\{hs\})$. By Lemma 3.15, we do not have to consider error cases. There are two other cases: Either there is no gs_2 such that $gs_1 \Rightarrow gs_2$. But then $\mathbf{nextA}_{\langle cs_1, cs_2 \rangle}$ is clearly a safe over-approximation of the (non-existent) concrete behavior.

Conversely, if $gs_1 \Rightarrow gs_2$, we can apply Lemma 3.13 to gs_1, gs_2, $\mathfrak{H}_1 := \{hs\}$, and $\mathfrak{H}_2 := \mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\{hs\})$ to conclude that there exists $hs_2 \in \mathfrak{H}_2$ with $gs_2 \in \mathcal{L}(hs_2)$. But then $\alpha(gs_2) = \mathrm{abst}(gs_2) = hs_2$ by backward confluence.

Hence for all $hs \in \mathfrak{H}'$ and $gs_1 \in \gamma(\mathfrak{H}')$, $\alpha(\mathbf{nextC}_{\langle cs_1, cs_2 \rangle}(\{gs_1\})) \subseteq \mathfrak{H}_2 = \mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\{hs\}) \subseteq \mathbf{nextA}_{\langle cs_1, cs_2 \rangle}(\mathfrak{H}')$. Noting that $\gamma(\mathfrak{H}') = \bigcup\{\gamma(\{hs\}) \mid hs \in \mathfrak{H}'\}$, the claim follows. ■

This justifies our use of the abstract semantics as basis for an analysis for proving the absence of memory errors.

R Note that the abstract semantics is not the most precise safe approximation w.r.t. α and γ, because the concretization function γ produces concrete stacks that cannot occur in any concrete execution. (The case where there is no gs_2 such that $gs_1 \Rightarrow gs_2$ in the proof of Theorem 3.12.) We could overcome this by adapting the call and return semantics: We could throw out the reachable fragment (apart from the boundary) already upon call rather than only at the return. Then, the various levels of the stack could be concretized individually without producing any inconsistent stacks. We would then obtain the most precise safe approximation. I opted against this to be able to define fork semantics that are very similar to call semantics—when forking, we must not throw away the reachable fragment, because it may be shared by other threads.[1]

We shall spend the remainder of this chapter exploring how we can extend both the concrete and the abstract semantics to obtain similar results for all of PL—that is, for programs that make use of fork–join parallelism.

[1] More concretely (if you have already had a look at the parallel semantics), we must not throw away the fragment because we need to retain the used, lost, and pending mappings.

CHAPTER 4

Semantics and Abstract Interpretation of Concurrent Programs

We have seen how to define a hypergraph-based semantics for a pointer language and how to define an abstract interpretation of such a semantics via hyperedge abstraction grammars. So far we have, however, focused on PL_{seq}, a sequential language. In this chapter, we explore how to extend the semantics to model fork–join parallelism.

Recall that PL supports the classical **fork–join model** of parallel computation: PL defines a **fork** statement for starting independent execution threads and a **join** statement for blocking until the forked thread has terminated. Each fork statement returns a **thread token**, to be stored in a thread token variable that was previously declared via the **thread** keyword:

$$\textbf{thread}\ t; t := \ \textbf{fork}\ p(x_1, \ldots, x_n)$$

Later, we can join the thread by referring to the thread token:

$$\textbf{join}\ t$$

We need this token mechanism (or something similar) because of the dynamic nature of PL's threading model: We do not know in advance how many threads a given program is going to create. It is thus not possible to statically limit the number of threads that are (potentially) active at the same time. We do not really care about the concrete form of the tokens, however. We shall, in fact, see that we can define a formal semantics without any reference to the tokens' values. In our semantics we will not allow aliasing of thread tokens—that is, assignments $t_2 := t_1$, where t_1 is a thread token—but I will argue later that this could be enabled easily by a slight complication of the model.

I go beyond some formulations of the fork–join model by allowing that threads be forked but never joined. I allow this to be a little closer to implementations of the fork–join model in real programming languages such as `Java` and to show that our approach is suitable for going beyond a parallel composition operator as known, for example, from process algebra [Bae05]. I do not model any bilateral synchronization mechanisms, such as Java monitors [Lea00], or more low-level synchronization constructs such as semaphores [Dow05]. This should be considered important future work. Note that we demand that a forked procedure must be joined by the same procedure instance (and hence thread) that forked it, or not at all.

Admittedly, this is a restriction compared to, for example, `Java`'s `Threads`, as `Thread` objects may be passed freely between objects.

R We have to take care to properly define what we mean by **parallel** or **concurrent** programs. In general, these terms cannot be used interchangeably. A possible definition is as follows.

> A **parallel** program is one that uses a multiplicity of computational hardware (e.g. multiple processor cores) in order to perform computation more quickly. [. . .] **concurrency** is a program-structuring technique in which there are multiple threads of control. Notionally the **threads of control** execute "at the same time"; that is, the user sees their effects interleaved. Whether they actually execute at the same time or not is an implementation detail [Mar12, Section 1.2].

On the one hand, only the potential for shared memory concurrency is of interest for the methods in this thesis, because our focus is on potential data races; hence the term "concurrent" applies. On the other hand, the adequate term for our model of concurrency is fork–join parallelism. As a result, you will encounter both terms throughout this thesis, but should be aware that the distinction is only of peripheral interest in our scenario.

The structure of this chapter mirrors the one of Chapter 3: In Section 4.1, I show how to adapt and extend the concrete semantics from Section 3.1 to take **fork–join parallelism** into account. This semantics is based on **permission accounting**. Section 4.2 develops an abstract interpretation for these extended semantics.

4.1 Concrete Semantics of Concurrent Programs

The overarching question of this chapters is: How can we define a suitable semantics for PL's variant of the fork–join model? Countless formal models and semantics for parallel and concurrent programming languages have been developed over the last four decades; see, for example, Chapter 14 in Winskel's classic book on program semantics [Win93] for an introduction.

For our purposes, the semantics should have all of the following properties.

- It should be based on interleaving of commands, rather than on true concurrency. This is the natural model when investigating data races in shared-memory concurrency: A data race is possible if different interleavings may yield different results.
- It should be defined in terms of (an extension of) our hypergraph-based heap model and be suitable for sound abstract interpretation via hyperedge replacement, so that we can reuse the semantics of the previous sections.
- We need the possibility to infer data-race freedom without looking at all (exponentially many) possible interleavings, which would be prohibitively expensive in all nontrivial cases.

To achieve all these properties, we add **access permissions** to the stack entries. Access permissions specify for each thread which memory locations it can read from and write to. The great benefit of permissions is that they allow modular reasoning about threads: If a thread holds a write permission to a memory location, it may write to that location without fearing either a concurrent write or a concurrent read from any other thread; if it holds a read permission, it knows that concurrent reads may occur, but concurrent writes are impossible [Boy03]. As long as a thread has sufficient permissions for all the memory operations it needs to perform, we can therefore reason about the thread in isolation. A permission system must, of course, be carefully designed to guarantee the desired invariants. In particular, there must never be more than one write permission for a location at a time, i.e., the duplication of permissions must be impossible. I discuss permissions in detail in Section 4.1.1.

New challenges for abstract interpretation

Ultimately, the goal of this chapter is to derive a safe approximation of a reasonable concrete semantics, building upon the results from Sections 3.1 and 3.2. In the concurrent fork–join setting, we, however, have to deal with one significant complication compared to the sequential call–return setting: The fork and join points may be arbitrarily far apart, potentially interspersed with additional forks and joins. Between the fork and the join, the heap's shape evolves locally as well as in the called thread. This was different in the sequential call–return setting, where the evolution was restricted to the called procedure. It was therefore comparatively straight-forward to replace the heap at call site with the modified heap that the procedure returned. (See Fig. 3.11 on page 44.) In the concurrent setting, it still suffices to pass the reachable fragment to the child thread, but we must still keep track of accesses to that fragment at fork site, as well as model the permission distribution between the two threads, until the join occurs at some later point—possibly with additional forks and joins in between.

Although this is not central to the current section, we must also keep in mind that this problem is further complicated through grammar-based abstraction: Between the fork and the join, abstraction and concretization steps may alter the shape of the heap fragment that the parent thread has passed to its child. For example, we might assign a different location to a variable in the parent thread, so the original location may become subject to abstraction. At the same time, the child may itself perform abstraction and concretization. Thus, the parent's and the child's views of the heap fragments may diverge, even if neither thread modifies the fragment. Inspired by the notion of spurious counterexamples in CEGAR approaches [Cla+00], we shall call this phenomenon a **spurious shape evolution**. In the presence of spurious shape evolution, we may not be able to match the parent's heap against the child's heap at the join point, unless we introduce an additional mechanism for keeping track of such evolution (which we will do in Section 4.2).

These observations influence the design space for the concurrent semantics, and we shall return to them throughout this chapter.

4.1.1 Permission Accounting

While interleaving semantics for concurrent programs are quite natural, they are problematic from the point of view of program analysis: Reasoning about a program based directly on an interleaving semantics would equate to reasoning about all possible interleavings, which does not scale well due to the inherent combinatorial explosion [Got+07].

One approach for proving data-race freedom of concurrent programs without considering all possible interleavings is **permission accounting**. The idea behind permission accounting is as follows.

There are three types of access permissions

- **Write permission**, i.e., the permission to both write to and read from a memory cell
- **Read permission**, i.e., the permission to read but not to alter a memory cell's value
- **No permission**

Whenever one thread holds a write permission to a memory location, all other threads may hold neither a write nor a read permission; whenever one thread holds a read permission, all other threads may hold read permissions, too, but not write permissions. If these invariants hold throughout the execution of the program, there cannot possibly be any data races. This is extremely useful in light of the following observation.

Observation 4.1 If a parallel program is data-race free, all possible interleavings of the program compute the same result.

In this setting we can therefore treat concurrent programs as if they were sequential and only need to consider a single (arbitrary) interleaving in the program analysis.

There are two questions that immediately come to mind:

- What representation should we choose for the permissions?
- How do we compute a valid permission distribution for data–race-free programs?

When answering the first question, we must keep in mind that we must be able to both split and recombine permissions in such a way that we can generate arbitrarily many read permissions, but must later only be able to recombine them into a write permission if we collect all the read permissions that we previously generated. To this end, using **counting permissions** and **fractional permissions** has been proposed. The former allows unbounded counting of read permissions (as is useful, for example, in the classic readers-and-writers algorithm), whereas the latter allows unbounded divisibility (as needed in concurrent divide-and-conquer schemes) [Bor+05]. Both these mechanisms were initially developed for languages with parallel composition.

More recently, Heule et al. [Heu+11] developed **permission expressions**, an adaptation of fractional permissions for modeling fork–join parallelism with dynamic thread creation and the possibility to keep track of permissions lost by threads that are never joined. Instead of explicit fractions, they use abstract read and write permissions that can be added and subtracted and which are parameterized by thread tokens. The tokens make it possible to soundly recombine permissions upon join. In addition, they introduce a $\mathbf{rd}*$ permission for modeling a read permission that can never again be used to reassemble a write permission, which happens when part of a permission is lost.

As is to be expected, we need something related to this last mechanism, as we must also handle dynamic thread allocation and lost permissions. I, however, use a different modeling mechanism: I only distinguish between read, write, and no permission, **rd**, **wt**, and **no**,, and a permission error, **er**, but keep multiple of these permissions around for each edge:

- A lower permission bound on the permission that the current thread (or any of its children) needs to guarantee execution without interference
- An upper bound on the permission that has been lost by the current thread (or its children), because of threads that were forked, but not joined and are now out of scope
- A map from **pending** threads to their required permissions: For each forked but unjoined (henceforth: pending) child thread, we keep track of the lower permission bounds that the child thread needs throughout its execution

Formally, given a set E, the permission accounting information is taken from the domain

$$\mathbf{Perms}_E := (E \to Perm) \times (E \to Perm) \times (Id \dashrightarrow (E \to Perm))$$

where $Perm := \{\mathbf{wt}, \mathbf{rd}, \mathbf{no}, \mathbf{er}\}$. For convenience, we define helper functions used, lost, and pending to access the entries of concrete values $q = (u,l,a) \in \mathbf{Perms}_E$, i.e.,

$$\begin{aligned}
&\mathrm{used} : \mathbf{Perms}_E \to (E \to Perm)\\
&\mathrm{used}((u,l,a)) = u\\
&\mathrm{lost} : \mathbf{Perms}_E \to (E \to Perm)\\
&\mathrm{lost}((u,l,a)) = l\\
&\mathrm{pending} : \mathbf{Perms}_E \to (Id \dashrightarrow (E \to Perm))\\
&\mathrm{pending}((u,l,a)) = a
\end{aligned}$$

This may sound unnecessarily complex; the reasoning behind this model is as follows. $\mathrm{pending}(q)$ contains the permissions for all threads that might possibly be running concurrently with the current thread and that are still in scope. $\mathrm{lost}(q)$ contains the aggregate of all permissions of the threads that may still be running but are no longer in scope. Consequently, if the next command in the current thread c executed on the current heap $\mathcal{H}$ needs permissions $\mathrm{requires}(c, \mathcal{H}) : E \to Perm$, we just need to check whether for all edges e,

$$\sum_t \mathrm{pending}(t)(e) + \mathrm{lost}(e) + \mathrm{requires}(c, \mathcal{H})(e) \leq \mathbf{wt}$$

where

$$\mathbf{no} \leq \mathbf{rd} \leq \mathbf{wt} \leq \mathbf{er}$$

and the addition of permissions is defined by

$$x + y = \begin{cases} x, & \text{if } y = \mathbf{no}\\ y, & \text{if } x = \mathbf{no}\\ \mathbf{rd}, & \text{if } x = y = \mathbf{rd}\\ \mathbf{er}, & \text{otherwise}\end{cases}$$

This allows us to use a greedy approach to compute a valid permission distribution: In the (sequential) execution of the concurrent program, we keep track of the minimal permissions that the thread needs (used) and the permission that have been lost throughout its execution (lost), as well as the minimal permissions needed by **pending threads**, i.e., threads that have been forked but not yet joined. (These threads' used permissions have in turn be computed greedily before.)

If at some point we encounter a command c such that

$$\sum_t \mathrm{pending}(t)(e) + \mathrm{lost}(e) + \mathrm{requires}(c, \mathcal{H})(e) = \mathbf{er}$$

the program contains a potential data race and we return an error. If, on the other hand, the greedy approach succeeds throughout the whole execution, this proves data-race freedom.

The huge advantage of this approach is that we do not need to parameterize permissions by (dynamic) thread tokens to be able to soundly recombine read permissions into write permissions: If and only if $\mathrm{lost}(e) = \mathbf{no}$, the remaining permissions may be combined into a write permission for e. Instead of the arbitrarily complex permission expressions of [Heu+11], we only need the simple set $\{\mathbf{no},\mathbf{rd},\mathbf{wt},\mathbf{er}\}$. We can therefore define a semantics without reasoning about concrete thread tokens at all.

R **Modularity.** Because of the pending map, this approach is not **thread-modular**: In a truly thread-modular analysis, each thread can be analyzed as if it were a sequential program [FFQ02; Got+07], whereas we explicitly keep track of possible interference with child processes. This is unavoidable, unless we make explicit assumptions about the execution environment, as is the case in thread-modular analyses. We do, however, preserve the same degree of modularity that we achieved in the sequential interprocedural setting: Each thread (each procedure) can be analyzed independently from the context in which it is forked (called). In particular, if a thread is run starting from the same reachable fragment multiple times, we only have to analyze it once, regardless of the forking context (see Chapter 5).[1]

4.1.2 Permission-Handling for PL_{seq} Commands

The first step on the way to a permission-based concurrent semantics is determining which access permissions are required for the execution of the various sequential commands. More specifically, let us think about which permissions are needed for the execution to progress a single step in the $\Rightarrow$ semantics.

- No permissions are involved in the execution of **skip** commands.
- The declaration **var** $x : t$ creates a new edge. No permission is needed for this operation, as no other thread is aware of this edge upon its creation.
- To execute $x := y.s$, we need read permission on $y.s$.
- To execute $x.s := y$, we need write permission on $x.s$.
- To execute $x.s_1 := y.s_2$, we need both write permission on $x.s_1$ and read permission on $y.s_2$.
- Just like with variable declaration, we do not need any permission for allocating the new edges in the execution of $x := \mathbf{new}\ t$.
- To progress a single step in $c_1; c_2$, we need the same permissions as for progressing c_1.
- To progress a single step in **if** and **while** commands, we need to evaluate their conditions, which requires read permissions on all pointers that occur in the conditions.
- Call and return can be defined just like in the sequential case. Execution of the procedure bodies, of course, requires permissions.

We formalize the required permissions by defining a function whose existence we already assumed in the previous section: A function requires, which, when applied to a command and a heap configuration $\mathcal{H}$, yields a function from $E_{\mathcal{H}}$ to $Perm$. As the definition of this function directly reflects the above observations, I just present a couple of examples instead of the whole definition.

$$\mathrm{requires}(x.s := y.t, \mathcal{H})(e) = \begin{cases} \mathbf{wt}, & \text{if } lab_{\mathcal{H}}(e) = s \wedge x \xmapsto{\mathcal{H}} att(e)(1) \\ \mathbf{rd}, & \text{if } lab_{\mathcal{H}}(e) = t \wedge y \xmapsto{\mathcal{H}} att(e)(1) \\ \mathbf{no}, & \text{otherwise} \end{cases}$$
$$\mathrm{requires}(c_1; c_2, \mathcal{H})(e) = \mathrm{requires}(c_1, \mathcal{H})(e)$$

As argued in the previous section, we can use requires in the concurrent semantics to check whether there exists a valid permission distribution. We will formalize this next.

[1]We will see in Section 4.2 that this is not entirely true for the abstract semantics, but even there a forked thread will not have any concrete knowledge about other threads and their permission allocation.

4.1.3 Deterministic Data–Race-Free Semantics

We now develop the deterministic data–race-free semantics for PL. We already observed before that

1. If we find a permission distribution for a program, this proves that the program is data-race free
2. If a program is data-race free, all interleavings of the parallel program return the same result

Hence it is sufficient to analyze an arbitrary interleaving of the program—for example by treating fork statements essentially like call statements, i.e., executing the child thread as a whole as soon as it is forked. To this end, we add a component q for permission accounting to the stack entries, which itself is a triple of functions as introduced in Section 4.1.1:

$$\mathbf{Perms}_E := (E \to Perm) \times (E \to Perm) \times (Id \dashrightarrow (E \to Perm))$$

$$\mathbf{Perms} := \bigcup_{\mathcal{H} \in \mathbf{HC}_{\mathrm{T},\Sigma}} \mathbf{Perms}_{E_\mathcal{H}}$$

$$\mathbf{Stk}_p := ((\mathbf{Cmd} \cup \{\downarrow\}) \times ((\mathbf{HC}_{\mathrm{T},\Sigma} \times \mathbf{Perms}) \cup \{\mathbf{err}\}))^+$$

How do we use these functions to define the semantics? Let us think about the various scenarios that we have to deal with in the execution of a concurrent PL program.

- When a thread is created with a heap configuration $\mathcal{H}$, the permission triple should be initialized with $\mathrm{used} = \lambda e \in E_\mathcal{H}.\mathbf{no}$, $\mathrm{lost} = \lambda e \in E_\mathcal{H}.\mathbf{no}$, and $\mathrm{pending} = \lambda t \in \mathbf{Id}.\bot$: It has not yet used any permissions, no permissions have been lost (locally) so far, and there are not yet any pending—that is, forked but unjoined—threads.
- Right at the fork point $t :=$ **fork** $p(\ldots)$, we have to compute the complete semantics of the child thread, i.e., the semantics of p. Provided the execution is successful, this will give us a used and a lost mapping for the child thread that we can use to update the local permission functions: We check for each edge whether used plus the sum of the parent's and child's lost mapping plus the sum of all pending threads in pending exceeds a write permission. If so, we go to an error state. If not, we add the child's lost value to the parent's lost value, and store the child's used mapping as $\mathrm{pending}(t)$ until the **join** t statement. In this way we denote that t now refers to a pending thread that has to be taken into account in future permission distributions until it is joined. In addition, we raise the permissions in used to the level of $\mathrm{pending}(t)$, if they are lower. This reflects that the parent thread needs at least the permissions of its children.
- If the thread associated with t is never joined, either because t is overridden or because it runs out of scope before the join (i.e., following the computation of the semantics of the thread after the fork), we must detect this and remove the entry from pending (or override it). In addition to increasing the value in used as in the ordinary join case, we must increase the values in lost to acknowledge that the thread's permissions have been lost and may never again be distributed. This is because we have no kind of progress or fairness assumption and therefore cannot derive a lower bound for the remaining run time of the lost thread. We shall call this update of the lost mapping a **ghost join**.
- Whenever the currently executed thread accesses an edge e, we have to check whether we can greedily allocate the necessary permission. To this end, we check whether the required permission plus the sum of the pending permissions on the edge plus $\mathrm{lost}(e)$

are at most a write permission. If so, we raise $\mathrm{used}(e)$ to the required permission. If not, we have a permission error and cannot find a valid distribution, so we enter an error state..

- At **join** t, we remove t from the pending threads, $\mathrm{pending}(t) := \bot$, reflecting that we may now reuse the corresponding permissions.

R I assume that a child thread may outlive its parent. If that is not the case, we do not need to perform ghost joins on a thread's pending children when it terminates. We also do not need to propagate lost values to the parent in that case, as we know that we regain the full available permission upon the thread's termination.

R Observe that the size of the domain of the $\mathrm{pending}$ mapping is bounded by the number of thread identifiers defined in the corresponding procedure and thus constant. In an implementation of this formalism, we thus only need a constant amount of additional memory per hyperedge.

We now adapt the inference rules for PL_{seq} to formalize the above points. We start with some auxiliary definitions. We write $\mathrm{proj}(stk)$ for the projection of a $\mathbf{Stk}_p$ to a $\mathbf{Stk}$, i.e.,

$$\begin{aligned}&\mathrm{proj} : \mathbf{Stk}_p \to \mathbf{Stk}\\ &\mathrm{proj}(\epsilon) := \epsilon\\ &\mathrm{proj}((c, \mathcal{H}, q) :: stk) := (c, \mathcal{H}) :: \mathrm{proj}(stk)\end{aligned}$$

For convenience, we lift addition, maximum, minimum, and comparison of permissions to functions of type $E \to Perm$:

$$\begin{array}{lcl} f \leq g & :\Longleftrightarrow & f(e) \leq f(g) \forall e \in E\\ (\max(f,g))(e) & := & \max(f(e),g(e))\\ (\min(f,g))(e) & := & \min(f(e),g(e))\\ (f+g)(e) & := & f(e)+g(e) \end{array}$$

We will also need to apply the maximum to all components of two permission mappings, which we denote $\max(q_1,q_2)$. (If $\mathrm{pending}_i(t)$ is only defined for one $i \in \{1,2\}$, we interpret this function as the maximum.)

In the inference rules below, we also need to update the domains of the functions in q whenever the rule takes a graph $\mathcal{H}$ to a graph $\mathcal{H}'$ with a different set of edges. A little sloppily, we shall write $\tilde{q}$ (without explicit reference to $\mathcal{H}$ or $\mathcal{H}'$) to mean the following transformation (where E denotes the old set of edges and E' the new set of edges):

$$\mathrm{used}(\tilde{q})(e) = \begin{cases}\mathrm{used}(q)(e), & \text{if } e \in E \cap E'\\ \mathbf{no}, & \text{if } e \in E' \setminus E\\ \bot, & \text{if } e \in E \setminus E'\end{cases}$$
$$\mathrm{lost}(\tilde{q})(e), \mathrm{pending}(\tilde{q})(t)(e) \text{ analogously}$$

Finally, we also need to formalize **ghost joins**. We denote the ghost join of a set of thread identifiers $\{t_1, \ldots, t_k\}$ w.r.t. q by $\mathrm{gjoin}(q,\{t_1, \ldots, t_k\})$.

$$\begin{aligned}&\mathrm{gjoin}(q,\emptyset) = q\\ &\mathrm{gjoin}(q,\{t_1,\ldots,t_k\}) = \begin{cases}\mathrm{gjoin}(q,\{t_2,\ldots,t_k\}), & \text{if } t_1 \notin Dom(\mathrm{pending}(q))\\ \mathrm{gjoin}(q',\{t_2,\ldots,t_k\}), & \text{otherwise}\end{cases}\\ &\quad \text{where } q' = (\mathrm{used}(q), \mathrm{lost}(q) + \mathrm{pending}(q)(t_1),\\ &\qquad\qquad \mathrm{pending}(q) \restriction (Dom(\mathrm{pending}(q)) \setminus \{t_1\}))\end{aligned}$$

We use these definitions to formalize the concurrent semantics, $\overset{P}{\Longrightarrow}$.

- **Sequential commands.** Figs. 4.1 to 4.4 show how to reuse the sequential semantics $\Rightarrow$ by coercing the stacks to their sequential representation. In addition, we make sure that we have the necessary permissions for each sequential command by comparing the sums against write permissions We log the permissions that are necessary to perform the sequential command by updating used. If we lack the required permissions, we transition to an error case. We also lift the error cases of the sequential semantics to the parallel semantics, and achieve error propagation in the same way (Fig. 4.5).
  Note the use of $\tilde{q}$ to reflect the appropriate changes to the domain; this way we take care of new edges as introduced, for example, by memory allocation.
- **Fork.** We treat a fork just like a call. Consequently, the fork semantics is split in two, Figs. 4.6 and 4.7. We reuse the call and return semantics to perform the reachable fragment computation, parameter renaming, merging etc.
  In the first rule, the forked thread is placed on top of the call stack and initialized with an empty permission distribution. It can therefore be analyzed in a modular way without taking the parent thread into account. The second rule merges the result of executing the child thread with the heap of the parent thread. The child thread is now a pending thread and thus added to the $\mathrm{pending}$ mapping.
  Of course we also need to handle permission distribution errors, which we do in a third rule (Fig. 4.8), in the same way as in Fig. 4.2.
  All fork rules use the auxiliary gjoin defined before to account for permissions that are lost when thread identifiers run out of scope: In the first rule, a ghost join occurs when a thread is assigned to the identifier t, even though t already represents a pending thread. In the second and third rule, a ghost join occurs if the child thread's execution ends with non-empty $\mathrm{pending}$, since all thread identifiers used in the child thread run out of scope.
- **Join.** The join operation is trivial in this semantics, because we already computed the child thread's effect and updated the heap at the fork point. We just remove the thread from the pending threads, i.e., remove the corresponding entry from $\mathrm{pending}$ (Fig. 4.9).

The semantics is obviously deterministic, because at most one rule of the $\overset{P}{\Longrightarrow}$ semantics can be a match and the underlying sequential semantics $\Rightarrow$ is deterministic as well.

Additionally, because it is sufficient to check the permissions for intraprocedural commands and for the return-from-fork, which we do in Fig. 4.8, the semantics transitions to an error case whenever there is no valid permission distribution among the threads that are currently running. As observed in Section 4.1.1, whenever there is a valid permission distribution, the program is data-race free.

Observation 4.2 If $P \vdash \langle(body(procs(P)(main)), \mathcal{H}, q_{init})\rangle \overset{P*}{\Longrightarrow} \langle(\downarrow, \mathcal{H}', q')\rangle$, then P is data-race free.

The above argument is, of course, not a formal proof. What would constitute a proof of this claim? We would need to track our permission tuples across the execution of all possible commands, showing that a thread always allocates all required permissions and proving the invariant that write permissions indeed provide mutual exclusion. It is quite possible that an attempt at such a proof would unveil a minor error in my semantics. I hope, however, that my elaborations have convinced you that my general approach does the trick, and that I can thus spare the both of us from a dozen or so pages of mindless technical proof.

$$\frac{\begin{array}{c} P \vdash (c,\mathcal{H}) :: \text{proj}(stk) \Rightarrow (c',\mathcal{H}') :: \text{proj}(stk) \\ \text{requires}(c) + \text{lost}(q) + \sum_t access(q)(t) \leq \lambda e.WT \\ q' = (\max(\text{used}(\tilde{q}), \text{requires}(c)), \text{lost}(\tilde{q}), \text{pending}(\tilde{q})) \end{array}}{P \vdash (c,\mathcal{H},q) :: stk \overset{P}{\Longrightarrow} (c',\mathcal{H}',q') :: stk}$$

Figure 4.1.: Lifting the intraprocedural semantics into the concurrent setting. The second precondition ensures that a permission distribution is possible. The third precondition expresses that we adapt the minimal used permissions to reflect the requirements of c.

$$\frac{\begin{array}{c} P \vdash (c,\mathcal{H}) :: \text{proj}(stk) \Rightarrow (c',\mathcal{H}') :: \text{proj}(stk) \\ \exists e.\ \text{requires}(c)(e) + \text{lost}(q)(e) + \sum_t \text{pending}(q)(t)(e) = \mathbf{er} \end{array}}{P \vdash (c,\mathcal{H},q) :: stk \overset{P}{\Longrightarrow} (c',\mathbf{err}) :: stk}$$

Figure 4.2.: Lifting the intraprocedural semantics into the concurrent setting. If the permission distribution is impossible for at least one edge, we transition to an error state.

$$\frac{P \vdash (\mathbf{call}\ p(\vec{x}); c, \mathcal{H}) :: \text{proj}(stk) \Rightarrow (c', \mathcal{H}_{callee}) :: (\mathbf{call}\ p(\vec{x}); c, \mathcal{H}) :: \text{proj}(stk)}{P \vdash (\mathbf{call}\ p(\vec{x}); c, \mathcal{H}, q) :: stk \overset{P}{\Longrightarrow} (c', \mathcal{H}_{callee}, \tilde{q}) :: (\mathbf{call}\ p(\vec{x}); c, \mathcal{H}, q) :: stk}$$

Figure 4.3.: Lifting the call semantics into the concurrent setting. No permissions are involved, we just have to cut the domain of q down to the reachable fragment.

$$\frac{\begin{array}{c} P \vdash (\downarrow, \mathcal{H}_{exit}) :: (\mathbf{call}\ p(\vec{x}); c, \mathcal{H}_{caller}) :: \text{proj}(stk) \Rightarrow (c, \mathcal{H}_{final}) :: \text{proj}(stk) \\ q_{max} = \max(q_{caller}, q_{exit}) \end{array}}{P \vdash (\downarrow, \mathcal{H}_{exit}, q_{exit}) :: (\mathbf{call}\ p(\vec{x}); c, \mathcal{H}_{caller}, q_{caller}) :: stk \overset{P}{\Longrightarrow} (c, \mathcal{H}_{final}, \tilde{q_{max}}) :: stk}$$

Figure 4.4.: Lifting the return semantics into the concurrent setting. q_{max} propagates the permissions used by the callee to the caller, including lost permissions and pending permissions.

$$\frac{P \vdash (c,\mathcal{H}) :: \text{proj}(stk) \Rightarrow (c',\mathbf{err}) :: \text{proj}(stk)}{P \vdash (c,\mathcal{H},q) :: stk \overset{P}{\Longrightarrow} (c',\mathbf{err}) :: stk} \qquad \frac{}{P \vdash (c,\mathbf{err}) :: stk \overset{P}{\Longrightarrow} (\downarrow, \mathbf{err})}$$

Figure 4.5.: Sequential error cases and propagation of errors are achieved in the obvious way.

$$\frac{\begin{array}{c} P \vdash (\textbf{call}\ p(\vec{x}); c, \mathcal{H}) :: \text{proj}(stk) \Rightarrow (c', \mathcal{H}_{fork}) :: (\textbf{call}\ p(\vec{x}); c, \mathcal{H}) :: \text{proj}(stk) \\ q_{par} = \text{gjoin}(q, t) \\ q_{fork} = (\lambda e.0, \text{lost}(q), \lambda t'.\bot, \lambda t'.\bot) \end{array}}{\begin{array}{l} P \vdash (t :=\ \textbf{fork}\ p(\vec{x}); c, \mathcal{H}, q) :: stk \\ \quad \overset{P}{\Longrightarrow} (c', \mathcal{H}_{fork}, q_{fork}) :: (t :=\ \textbf{fork}\ p(\vec{x}); c, \mathcal{H}, q_{par}) :: stk \end{array}}$$

Figure 4.6.: Fork semantics, part 1: Initializing the thread. We refer to the call semantics, as the reachable fragment computation is the same. In addition, we initialize the permission accounting for the child thread and perform a ghost join in the parent thread if t is overridden.

$$\frac{\begin{array}{c} (\downarrow, \mathcal{H}_{exit}) :: (\textbf{call}\ p(x_1, \ldots, x_n); c, \mathcal{H}_{par}) :: \text{proj}(stk) \Rightarrow (c, \mathcal{H}_{final}) :: \text{proj}(stk) \\ q_{gjoined} = \text{gjoin}(q_{exit}, Dom(q_{exit})) \\ \text{used}_{final} = \min(\text{used}(q_{gjoined}), \text{used}(q_{par})) \\ \text{lost}_{final} = \text{lost}(q_{par}) + \text{lost}(q_{gjoined}), \\ \text{pending}_{final} = \text{pending}(q_{par}) \cup (t \mapsto \lambda e.\, \text{used}(q_{gjoined})(e)) \\ q_{final} = (\text{used}_{final}, \text{lost}_{final}, \text{pending}_{final}) \\ \text{lost}(q_{final}) + \sum_t \text{pending}(q_{final})(t) \leq \lambda e.WT \end{array}}{\begin{array}{l} P \vdash (\downarrow, \mathcal{H}_{exit}, q_{exit}) :: (t :=\ \textbf{fork}\ p(\vec{x}); c, \mathcal{H}_{par}, q_{par}) :: stk \\ \quad \overset{P}{\Longrightarrow} (c, \mathcal{H}_{final}, q_{\tilde{final}}) :: stk \end{array}}$$

Figure 4.7.: Fork semantics, part 2: Return from the child thread. We reuse the return-from-call semantics to merge the heaps. In addition, we perform permission accounting: Any remaining children of the child process are ghost joined. We note that the parent thread needs at least the permissions of the child thread, that permissions lost in the child thread are also lost from the point of view of the parent, and add the child thread itself to the set of pending threads.

$$\frac{\begin{array}{c} (\downarrow, \mathcal{H}_{exit}) :: (\textbf{call}\ p(x_1, \ldots, x_n); c, \mathcal{H}_{par}) :: \text{proj}(stk) \Rightarrow (c, \mathcal{H}_{final}) :: \text{proj}(stk) \\ q_{gjoined} = \text{gjoin}(q_{exit}, Dom(q_{exit})) \\ \text{lost}_{final} = \text{lost}(q_{caller}) + \text{lost}(q_{gjoined}), \\ \text{pending}_{final} = \text{pending}(q_{par}) \cup (t \mapsto \lambda e.\, \text{used}(q_{gjoined})(e)) \\ \exists e.\, \text{lost}_{final}(e) + \sum_t \text{pending}_{final}(t)(e) = \textbf{er} \end{array}}{P \vdash (\downarrow, \mathcal{H}_{exit}, q_{exit}) :: (t :=\ \textbf{fork}\ p(\vec{x}); c, \mathcal{H}_{par}, q_{par}) :: stk(c, \textbf{err}) :: stk}$$

Figure 4.8.: Fork semantics, error case: We check whether at least one distribution requirement is violated upon the return. Other than the slightly reduced bookkeeping, this rule is identical to Fig. 4.7.

$$\frac{q' = (\text{used}(q), \text{lost}(q), \text{pending}(q) \restriction (Dom(\text{pending}(q)) \setminus \{t\}))}{P \vdash (\textbf{join}\ t; c, \mathcal{H}, q) :: stk \overset{P}{\Longrightarrow} (c, \mathcal{H}, q') :: stk}$$

Figure 4.9.: Join semantics. t is removed from the pending threads. $\mathcal{H}$ is not modified, as the heap views were already merged by the second fork rule.

R By treating the fork like a call, we also immediately apply the child's modifications of the heap to the parent heap. In a sense the parent thread therefore always has the maximally progressed view of the fragment of the heap that it can access. This may not sound intuitive, as we might expect that we update the heap only at the join point. This is barely a matter of taste, however; from a correctness point of view, it does not matter when we update the parent's heap: If it did matter, we would have a potential data race.

4.2 Abstract Interpretation of Concurrent Programs

In this section we will informally develop an abstract interpretation of the data–race-free semantics for PL. The goal is to integrate permission accounting into the abstract interpretation of PL_{seq}. In particular, we will, of course, want to find an abstract semantics that safely approximates the concrete permission-based semantics. We will apply techniques similar to those from Section 3.2 to the semantics from Section 4.1. Due to the complexity and sheer size of the semantics in Section 4.1, I will not attempt a complete and rigorous proof of the safe approximation.

So how do we go about defining an abstract semantics for PL? In principle, there is nothing stopping us from assigning permissions to nonterminal edges and then use abstraction and concretization in a way that respects the permissions. We must, however, take care to do this in a way that does not lose information, let alone soundness. For this reason we should only allow abstraction of subgraphs with uniform permissions. Conversely, when we apply a production rule to a nonterminal edge with a certain permission, the right-hand side should also be consistently assigned the same permission information.

R Note that this restriction will lead to an infinite state space even under abstraction, as in general, we cannot bound the number of **permission alternations** that occur in concrete data structures. I will return to this problem towards the end of this section.

We will formalize this notion of permission-preserving abstraction and concretization in Section 4.2.1. Before we get there, I would like to mention one additional problem, which motivates the definitions in this section. Remember how the call and return semantics were designed in a way that we were able to easily replace the outdated caller's view of the reachable fragment with the result of executing the procedure. This worked for both concrete and abstract graphs. The concrete fork and return-from-fork rules in Section 4.1 then simply reused this machinery. In the presence of abstraction, this may unfortunately invalidate the $\mathrm{pending}$ mappings: If the set of edges has changed because of abstraction and/or concretization during the execution of the child thread, we lose edges that are in the domain of the $\mathrm{pending}(t)$ functions and add new edges that are not in the domain, even in cases where the language of the abstract heap(s) remains unchanged! This makes it impossible to recombine permissions upon return-from-fork, because we are simply unable to match the correct edges against each other.

One way to circumvent this would be to pass along the $\mathrm{pending}$ functions to the child process, but that is undesirable for two reasons. First, we would have to deal with aliasing of thread variables t, but that would merely be a technical inconvenience. Second, and more importantly, we would lose modularity, because the analysis of the child thread would then depend on the analysis of the parent thread.

Instead we introduce an additional component into the stack entries that keeps track of the **shape evolution** of the local heap configuration. That is, given $\mathcal{H}, \mathcal{H}' \in \mathbf{HC}_{\mathrm{T},\Sigma}$, where $\mathcal{H}$ was the heap when the thread started its execution and $\mathcal{H}'$ was obtained from $\mathcal{H}$ through a series of transformations, we define a function $\eta : E_{\mathcal{H}'} \to 2^{E_{\mathcal{H}}}$ that maps each new edge e to the set of original edges from which e was derived through concretization and abstraction. (Or $\emptyset$ for edges that were added by the thread itself.) This shape evolution function will allow us to update the functions in q correctly after returning from the fork.

R We need to allow multiple target edges because abstraction steps that occur during the abstract execution may lead to a merging of initially separate edges.

4.2.1 Abstract Semantics with Shape Evolution

Throughout this section, we once again assume the presence of a backward-confluent heap abstraction grammar, 𝔊Recall that the concrete semantics for (the successful) execution of PL programs was defined on stacks of commands, hypergraphs, and permissions:

$$\mathbf{Stk}_p := (\mathbf{Cmd} \times ((\mathbf{HC}_{\mathrm{T},\Sigma} \times \mathbf{Perms}) \cup \{\mathbf{err}\}))^+$$

We further extend the stack entries with a fourth component to keep track of shape evolution:

$$\mathbf{Evo}_{\mathcal{H},\mathcal{H}'} := E_{\mathcal{H}'} \to E_{\mathcal{H}}$$
$$\mathbf{Evo} := \bigcup_{\mathcal{H},\mathcal{H}' \in \mathbf{HC}_{\mathrm{T},\Sigma}} \mathbf{Evo}_{\mathcal{H},\mathcal{H}'}$$
$$\mathbf{Stk}_a := (\mathbf{Cmd} \times ((\mathbf{HC}_{\mathrm{T},\Sigma} \times \mathbf{Perms} \times \mathbf{Evo}) \cup \{err\}))^+$$

We shall denote individual shape evolution functions by $\eta : E_{\mathcal{H}'} \to 2^{E_{\mathcal{H}}}$. Intraprocedural commands thus transform stack frames of the form $(c,\mathcal{H},q,\eta)$, interprocedural (and interthread) commands operate on two such stack frames.

Each step in the abstract semantics becomes (compare Section 3.2.1):

1. On-demand permission-preserving concretization with corresponding update of q and η
2. Application of concrete data–race-free semantics, with additional update of η where applicable
3. Permission-preserving abstraction (as fully as possible)

Just like we did in Section 3.2, we begin with a formalization of concretization and abstraction and then wrap these steps around the concrete semantics. To this end, it will be convenient to have a shorthand for composing two shape evolution functions.

Definition 4.3 — **Composition of shape evolution functions.** Let $\eta_1 : E_1 \to 2^{E_0}, \eta_2 : E_2 \to 2^{E_1}$. We define the composition of shape evolution functions

$$\eta_2 \bullet \eta_1 : E_2 \to 2^{E_0}$$
$$(\eta_2 \bullet \eta_1)(e) := \bigcup_{e' \in \eta_2(e)} \eta_1(e')$$

Concretization with permissions and shape evolution

To define concretization in the parallel setting, we need to make precise the notion of permission-preserving hyperedge replacement.

$$\mathrm{ppconc}_{\mathfrak{G}}(\mathcal{H},q,\eta) := \begin{cases} \{(\mathcal{H},q,\eta)\}, & \text{if } \mathrm{vp}_{\mathfrak{G}}(\mathcal{H}) = \emptyset \\ \bigcup\{\mathrm{ppconc}_{\mathfrak{G}}(\mathcal{H}',q',\eta') \mid (e,i) \in \mathrm{vp}_{\mathfrak{G}}(\mathcal{H}) & \\ \qquad \wedge (\mathcal{H}',q',\eta') \in \mathit{concAt}_{\mathfrak{G}}(e,i,(\mathcal{H},q,\eta))\}, & \text{otherwise} \end{cases}$$

$$\mathit{concAt}_{\mathfrak{G}}(e,i,(\mathcal{H},q,\eta)) := \{(\mathcal{H}[e/\mathcal{G}],\hat{q},\eta_{\langle\mathcal{H},e,\mathcal{G}\rangle} \bullet \eta) \mid (\mathit{lab}(e) \to \mathcal{G}) \in \mathfrak{G}_i^\sigma \wedge \hat{q} \text{ preserves permissions}\}$$

Figure 4.10.: Permission-preserving on-demand concretization, ppconc. Each concretization step *concAt* updates η to keep track of shape evolution.

Definition 4.4 — **Permission-preserving hyperedge replacement.** Let $\mathcal{H},\mathcal{G}$ be hypergraphs and $e \in E_{\mathcal{H}}$ be a hyperedge with $|\mathit{ext}_{\mathcal{G}}| = \mathrm{rk}(\mathit{lab}(e))$. Let $V_{\mathcal{H}} \cap V_{\mathcal{G}} = \emptyset$. Let further $\mathcal{H}[e/\mathcal{G}] =: \mathcal{J}$ denote the replacement of e by $\mathcal{G}$ in $\mathcal{H}$ as per Def. 2.26 on page 23.

We say that $(\mathcal{J},\hat{q})$ is a **permission-preserving hyperedge replacement** w.r.t. $q \in \mathbf{Perms}_{E_{\mathcal{H}}}$ if

- $\hat{q} \in \mathbf{Perms}_{E_{\mathcal{J}}}$
- $\hat{q}$ and q agree on all $e \in E_{\mathcal{H}} \cap E_{\mathcal{J}}$
- For all $e' \in E_{\mathcal{G}}$ and all t, $\mathrm{used}(\hat{q})(e') = \mathrm{used}(q)(e)$, $\mathrm{lost}(\hat{q})(e') = \mathrm{lost}(q)(e)$, $\mathrm{pending}(\hat{q})(t)(e') = \mathrm{pending}(q)(t)(e)$

Each such replacement must additionally be reflected in the shape evolution function. To this end we define a shape evolution function that precisely reflects the changes induced by the hyperedge replacement. This function can then be composed with the shape evolution function prior to concretization to obtain the updated shape evolution.

Definition 4.5 — **Shape evolution for hyperedge replacement.** Given the hyperedge replacement $\mathcal{H}[e/\mathcal{G}]$, we define $\eta_{\langle\mathcal{H},e,\mathcal{G}\rangle} \in \mathbf{Evo}_{\mathcal{H},\mathcal{H}[e/\mathcal{G}]}$ by:

$$\eta_{\langle\mathcal{H},e,\mathcal{G}\rangle}(e') = \begin{cases} \{e'\}, & \text{if } e' \notin E_{\mathcal{G}} \\ \{e\}, & \text{otherwise} \end{cases}$$

This finally allows us to extend the definition of on-demand concretization from page 46 to include permissions and shape evolution. We denote this extended function by ppconc. It is defined in Fig. 4.10. (We again write $\mathfrak{G}_1^\sigma, \ldots, \mathfrak{G}_{\mathrm{rk}(\sigma)}^\sigma \subseteq \mathfrak{G}^\sigma$ for the language-preserving subgrammars of $\mathfrak{G}$ that remove violation points at the k attachment points.)

While the definitions are admittedly becoming increasingly unwieldy, the underlying intuition is quite simple: We concretize on demand just like before, but additionally, we make sure to preserve permissions and to update the shape evolution function (through function composition) to keep track of the graph transformation.

Abstraction with permissions and shape evolution

We lift abstraction to the parallel setting completely analogously, obtaining a function $\mathrm{ppabst}_{\mathfrak{G}}(\mathcal{H},q,\eta)$ that iterates permission-preserving subgraph abstraction. Having seen the full definition of ppconc, the formal definition of ppabst will not provide any insight. I shall therefore omit it and only provide the necessary underlying definitions.

Definition 4.6 — **Permission-preserving subgraph abstraction.** Let $\mathcal{H} \in \mathbf{HG}_{\mathrm{T},\Sigma}$, $q \in \mathbf{Perms}_E$, $W \subseteq V_{\mathcal{H}}$ and $\mathcal{G} = \mathcal{H} \times W$ be a section hypergraph such that for all $e_1, e_2 \in E_{\mathcal{G}}$

- $\mathrm{used}(q)(e_1) = \mathrm{used}(q)(e_2)$
- $\mathrm{lost}(q)(e_1) = \mathrm{lost}(q)(e_2)$
- $\forall t.\, \mathrm{pending}(q)(t)(e_1) = \mathrm{pending}(q)(t)(e_2)$

Let further $\sigma \in \Sigma_{NT}$, and $ext_{match} \in (V_{\mathcal{G}})^{\mathrm{rk}(\sigma)}$ be a sequence of nodes such that $ext_{\mathcal{G}} \setminus \mathrm{bound}(\mathcal{G},\mathcal{H}) = \emptyset$. Let $\mathcal{J} := \mathcal{H}[\mathcal{G}\&ext_{match}/\sigma]$ (as defined in Def. 2.27 on page 24).

$(\mathcal{J},\hat{q})$ is a **permission-preserving abstraction** of $\mathcal{G}$ by σ if

- $\hat{q} \in \mathbf{Perms}_{E_{\mathcal{J}}}$
- $\hat{q}$ and q agree on all edges in $\mathcal{H} \setminus \mathcal{G}$
- $\hat{q}$ assigns to the new abstract edge the unique permission information of the abstracted edges

Definition 4.7 — **Shape evolution for subgraph abstraction.** Given the subgraph abstraction $\mathcal{H}[\mathcal{G}\&ext_{match}/\sigma]$, we define $\eta_{\langle \mathcal{H},\mathcal{G},ext_{match},\sigma\rangle} \in \mathbf{Evo}_{\mathcal{H},\mathcal{H}[\mathcal{G}\&ext_{match}/\sigma]}$ by:

$$\eta_{\langle \mathcal{H},\mathcal{G},ext_{match},\sigma\rangle}(e') = \begin{cases} \{e'\}, & \text{if } e' \in E_{\mathcal{H}} \\ E_{\mathcal{G}}, & \text{otherwise} \end{cases}$$

$\mathrm{ppabst}_{\mathfrak{G}}$ is then defined as the iterated application of permission-preserving abstraction, where each step updates η through composition with $\eta_{\langle \mathcal{H},\mathcal{G},ext_{match},\sigma\rangle}$.

Lifted semantics

We now turn to the formalization of the semantics. We denote the new relation $\overset{\eta}{\Longrightarrow}$, where the η indicates that we track shape evolution. We first lift the sequential semantics to our extended setting. The only interesting rule for this subset is the one for the intraprocedural semantics, where the extended wrapping is done; it is shown in Fig. 4.11.

Beside wrapping the concrete semantics in ppconc and ppabst, there is just one special case: Upon memory allocation, we need to extend the shape evolution mapping and map the new edges to the empty set, as they have no counterpart in the original hypergraph. This is achieved using the following auxiliary definition:

$$\mathrm{instantiate}(\eta, E_{new})(e) = \begin{cases} \eta(e), & \text{if } e \notin E_{new} \\ \emptyset, & \text{otherwise} \end{cases}$$

Note that I will omit all rules for error cases in this section. They are a completely straightforward adaptation from the underlying semantics, $\overset{P}{\Longrightarrow}$. Lifting call and return is also straightforward: We only add the η bookkeeping and are done. Since they can also be regarded as simpler special cases of the fork rules, I omit the corresponding rules for the sake of brevity. The most interesting rules are, of course, the rules for fork and return-from-form, given in Figs. 4.12 and 4.13.

The fork rule, Fig. 4.12, starts by marking all nodes of $\mathcal{H}$ as external, denoted by externalize (formal definition omitted). This may seem rather drastic, and it is. We need to have a look at the return-from-fork rule to make sense of this step, so I will return to it later. Apart from the use of externalize, the fork rule is a straightforward adaptation of the corresponding rule in the concrete semantics (cf. Fig. 4.6 on page 65). The new thread has to keep track of

$$\frac{\begin{array}{cc}(\mathcal{H}_0,q_0,\eta_0) \in \mathrm{ppconc}(\mathcal{H},q,evo) & P \vdash (c,\mathcal{H}_0,q_0) :: stk \overset{P}{\Longrightarrow} (c',\mathcal{H}_1,q_1) :: stk \\ \eta_1 = \mathrm{instantiate}(\eta_0,E_{\mathcal{H}_1} \setminus E_{\mathcal{H}_0}) & (\mathcal{H}_2,q_2,\eta_2) = \mathrm{ppabst}(\mathcal{H}_1,q_1,\eta_1)\end{array}}{P \vdash (c,\mathcal{H},q,\eta) :: stk \overset{\eta}{\Longrightarrow} (c',\mathcal{H}_2,q_2,\eta_2) :: stk}$$

Figure 4.11.: The lifted intraprocedural semantics that form the basis of the abstract parallel semantics. Beside wrapping the concrete semantics in (extended) concretization and abstraction, we also add new edges to η via the instantiate auxiliary function.

$$\frac{\begin{array}{c}\mathcal{H}_{ext} = \mathrm{externalize}(\mathcal{H}) \\ (\textbf{call } p(\vec{x}); c, \mathcal{H}_{ext}) :: \mathrm{proj}(stk) \overset{A}{\Longrightarrow} (c', \mathcal{H}_{fork}) :: (\textbf{call } p(\vec{x}); c, \mathcal{H}) :: \mathrm{proj}(stk) \\ q_{par} = \mathrm{gjoin}(q,t) \\ q_{fork} = (\lambda e.0, \mathrm{lost}(q), \lambda t'.\bot, \lambda t'.\bot) \\ \eta_{fork} = \lambda e.\{e\}\end{array}}{\begin{array}{l}P \vdash (t := \textbf{ fork } p(\vec{x}); c, \mathcal{H}, q, \eta) :: stk \\ \quad \overset{\eta}{\Longrightarrow} (c', \mathcal{H}_{fork}, q_{fork}, \eta_{fork}) :: (t := \textbf{ fork } p(\vec{x}); c, \mathcal{H}, q_{par}, \eta) :: stk\end{array}}$$

Figure 4.12.: Abstract fork semantics, part 1: Initializing the thread. We mark all nodes as external to avoid the extreme over-approximation of permissions through abstraction that would occur after the erasure (q_{fork}) of permission information. For formal reasons, we switch over to the abstract semantics of call statements. We also initialize the shape evolution for the new thread. As the shape evolution function always maps back to the start of the thread, it is initially the identity function.

$$\frac{\begin{array}{c}(\downarrow, \mathcal{H}_{exit}) :: (\textbf{call } p(x_1, \ldots ,x_n); c, \mathcal{H}_{par}) :: \mathrm{proj}(stk) \overset{A}{\Longrightarrow} (c, \mathcal{H}_{final}) :: \mathrm{proj}(stk) \\ q_{gjoined} = \mathrm{gjoin}(q_{exit}, Dom(q_{exit})) \\ q_{evo} = \mathrm{apply}(\eta_{exit}, q_{par}) \\ \mathrm{used}_{final} = \max(\mathrm{used}(q_{gjoined}), \mathrm{used}(q_{evo})) \\ \mathrm{lost}_{final} = \mathrm{lost}(q_{evo}) + \mathrm{lost}(q_{gjoined}), \\ \mathrm{pending}_{final} = \mathrm{pending}(q_{evo}) \cup (t \mapsto \lambda e.\, \mathrm{used}(q_{gjoined})(e)) \\ q_{final} = (\mathrm{used}_{final}, \mathrm{lost}_{final}, \mathrm{pending}_{final}) \\ \mathrm{lost}(q_{final}) + \sum_t access(q_{final})(t) \leq \lambda e.WT\end{array}}{\begin{array}{l}P \vdash (\downarrow, \mathcal{H}_{exit}, q_{exit}, \eta_{exit}) :: (t := \textbf{ fork } p(\vec{x}); c, \mathcal{H}_{par}, q_{par}, \eta_{par}) :: stk \\ \quad \overset{P}{\Longrightarrow} (c, \mathcal{H}_{final}, q_{fi\tilde{n}al}, \mathrm{extend}(\eta_{exit},\mathcal{H}_{final}) \bullet \eta_{par}) :: stk\end{array}}$$

Figure 4.13.: Abstract fork semantics, part 2: Return from the child thread. On top of the concrete semantics, we progress the set of edges in the parent thread according to η_{exit}. This makes it possible to take the maximum over permissions in a sound way. We continue with the composed shape evolution.

its own shape evolution, so that we can later merge its permissions with the parent thread. As no shape evolution has occurred initially, we start with the identity function.

Fig. 4.13, which specifies the return from forked threads, is the most involved rule in my entire thesis. Once again, the starting point is the rule from the concrete semantics. But now we have to use the shape evolution function η_{exit} to be able to match the permission accounting information.

To this end, we introduce a couple of auxiliary definitions. First, we need to be able to extend the domain of a shape evolution function to the entire set of edges of a hypergraph.

$$\text{extend}(\eta,\mathcal{H})(e) := \begin{cases} \eta(e), & \text{if } e \in \mathit{Dom}(\eta) \\ \{e\}, & \text{if } e \in E_{\mathcal{H}} \setminus \mathit{Dom}(\eta) \end{cases}$$

This is used to define the resulting shape evolution function: The execution continues with $\text{extend}(\eta_{exit},\mathcal{H}_{final}) \bullet \eta_{par}$ rather than η_{par}. This makes sense: To obtain $\mathcal{H}_{final}$, we replaced a heap fragment at call site with the returned heap $\mathcal{H}_{exit}$, and thus we must not forget the shape evolution that led us to $\mathcal{H}_{final}$. extend ensures that the resulting shape evolution function is defined for all edges in $\mathcal{H}_{final}$ rather than only those from $\mathcal{H}_{exit}$.

Second, we need to progress the permission accounting functions of the parent thread q_{par} according to the shape evolution η_{exit} to align their domains, captured in the following definition.

Definition 4.8 — **Application of shape evolution to permission accounting information.** Let q be permission accounting information for a set of edges E and $\eta \in \mathbf{Evo}_{\mathcal{H},\mathcal{H}'}$ be a shape evolution that maps each $e \in E'$ to a subset of E. We say that the application of η to q leads to q', written $q' = \text{apply}(\eta,q)$, if

- $\mathit{Dom}(\text{used}(q')) = (E \setminus \mathit{Cod}(\eta)) \cup \mathit{Dom}(\eta)$
- $\text{used}(q')(e) = \text{used}(q)(e)$ for all $e \notin \mathit{Dom}(\eta)$
- $\text{used}(q')(e) = \max\{\text{used}(q)(e') \mid e' \in \eta(e)\}$
- The same holds for $\text{lost}(q'), \text{pending}(q')$

In other words, we create a new permission mapping which is defined for all edges that resulted from the shape evolution as well as all edges that were not involved in the shape evolution (i.e., which were not in the reachable fragment of the thread that performed the shape evolution). The permissions of those edges that were not passed to the child thread ($e \notin \mathit{Dom}(\eta)$) are not changed. If, on the other hand, $e \in \mathit{Dom}(\eta)$, we perform η lookups to gather all permission information of the edges in the parent thread from which e was derived. This can then be compared against the permission information in the child thread to find out whether a permission error may have occurred.

The reasoning behind externalize

Taking the maximum over all $\eta(e)$ values is of course an over-approximation, but it is the only thing we can do, given that the edge e represents all the original edges in $\eta(e)$.

Things would be much worse if we did not externalize the nodes of the graph: Recall that, upon forking, we throw away all permission information of the parent thread to retain modularity (expressed in the precondition for q_{fork} in Fig. 4.12). We are therefore suddenly able to abstract all those concrete parts that we previously could not abstract because of the constraint that abstraction be permission preserving. Say we obtain an abstract edge e in this fashion. Upon return from the forked thread, we have to recombine the forked thread with the parent thread, thus use apply. apply will transform the entire set of edges that was abstracted into e into the edge(s) derived from e: Again, the only way to combine permissions in a sound way is to take the maximum of all permissions in the concrete subgraph and add it to the permissions of e to see if the distribution is valid. This would aggressively over-approximating the permission accounting information of the parent thread.

On the downside, marking all nodes as external obviously quickly leads to an explosion in the size of the graphs that occur in the analysis for algorithms with non-uniform permission allocation pattern. (The size of the graphs is not bounded in this case.) To alleviate this, we could compromise and define a constant bound on the number of nodes that are made external in this fashion—just like the bound for the number of cutpoints. The remainder of the graph would then be subject to aggressive over-approximation.

Size of the abstract domain

Recall from page Section 3.2.2 that, in the sequential setting, we needed two conditions to guarantee that the abstract state space was finite: First, we had to find an HRG that allowed abstracting all but a bounded number of nodes at each step in the program execution. Second, we also had to be able to bound the number of cutpoints by a constant. This still holds in the parallel setting of this chapter, but now the first requirement is much harder to achieve, because we only abstract data structures that have a uniform permission distribution. To guarantee a finite abstract state space, we therefore have to prove that throughout the program execution, only a constant number of nodes cannot be abstracted because of differences in the permission distribution.

For many algorithms, this is actually easy. Parallel divide-and-conquer algorithms will, for example, usually split their input list (or tree, array, etc.) into a constant number of chunks of roughly equal size, each of which can be fully abstracted in our approach. If we instead write an algorithm that needs a write permission on every other element of a list, our approach fails: We get an unbounded chain of permission alternations on the list.

Note that there are ways to get around this problem—for example by introducing permission-valued parameters into the grammars to be able to abstract data structures with non-uniform permissions without the need to over-approximate the permissions; see Section 6.2.

4.2.2 Soundness of the Abstract Semantics

The inference rules have now become so involved as to be nearly unintelligible—at least without the careful reading of the previous sections. Crucially, however, the abstract semantics are once again quite a thin wrapper around the concrete semantics: Compared to the concrete setting, we have to do some additional bookkeeping (η) to be able to track permissions, but at its core, the abstract semantics once again merely wraps the concrete semantics in concretization and abstraction steps.

Intuitively, we should therefore be able to prove that abstract semantics is a safe approximation of the concrete semantics in a similar way to Section 3.2.3. As a first step, we would need to come up with abstraction and concretization functions,

$$\begin{aligned}
\alpha &: 2^{((\mathbf{HC}^0\times\mathbf{Perms})\cup\{\mathbf{err}\})^+} \to 2^{((\mathbf{HC}\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+}\\
\gamma &: 2^{((\mathbf{HC}\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+} \to 2^{((\mathbf{HC}^0\times\mathbf{Perms})\cup\{\mathbf{err}\})^+}
\end{aligned}$$

Unfortunately, the additional η component makes it difficult to do this directly, as the abstraction function would need to generate a suitable η function, even though we do not keep any explicit history information in the concrete domain $(\mathbf{HC}^0 \times \mathbf{Perms}) \cup \{\mathbf{err}\}$. In the concrete setting, we can identify the original set of edges of the child thread by taking the intersection of the edge sets of child and parent, but even if we did that in the definition of the abstraction function, we would still get an extremely convoluted mapping between concrete and abstract states that would complicate the proofs.

So what can we do instead? The key idea is that we can define a variant of the concrete semantics that also uses a shape evolution function, but whose projection onto the concrete domain agrees with the concrete semantics for all program traces. Formally, we define a concrete semantics $\overset{e}{\Longrightarrow}$ on the extended domain

$$\mathbf{Stk}_e := (\mathbf{Cmd} \times ((\mathbf{HC}^0_{\mathtt{T},\Sigma} \times \mathbf{Perms} \times \mathbf{Evo}) \cup \{err\}))^+$$

such that if

$$\langle(c_1, \mathcal{H}_1, q_1, \eta_1), \ldots, (c_k, \mathcal{H}_k, q_k, \eta_k)\rangle \overset{e}{\Longrightarrow} \langle(c'_1, \mathcal{H}'_1, q'_1, \eta'_1), \ldots, (c'_j, \mathcal{H}'_j, q'_j, \eta'_j)\rangle$$

then

$$\langle(c_1, \mathcal{H}_1, q_1), \ldots, (c_k, \mathcal{H}_k, q_k)\rangle \overset{P}{\Longrightarrow} \langle(c'_1, \mathcal{H}'_1, q'_1), \ldots, (c'_j, \mathcal{H}'_j, q'_j)\rangle$$

with identical $\mathcal{H}'_k$ and q'_k. Once we have defined such a semantics $\overset{e}{\Longrightarrow}$, we can show that $\overset{\eta}{\Longrightarrow}$ safely approximates $\overset{e}{\Longrightarrow}$. Since the projection of the result of applying $\overset{e}{\Longrightarrow}$ onto $(\mathbf{HC}^0 \times \mathbf{Perms}) \cup \{\mathbf{err}\}$ yields the exact same result as applying $\overset{P}{\Longrightarrow}$, we can then conclude that $\overset{\eta}{\Longrightarrow}$ also safely approximates $\overset{P}{\Longrightarrow}$. Giving the full definition and the full proofs would be rather tedious; I will instead try to convince you with brief arguments that both the definition of $\overset{e}{\Longrightarrow}$ and the subsequent safe approximation proof are possible.

The bookkeeping semantics $\overset{e}{\Longrightarrow}$

Recall how η was used in the abstract semantics:

1. Abstraction and concretization were translated into corresponding updates of η (Fig. 4.11).
2. New edges were mapped to $\perp$ via the $\mathrm{instantiate}$ function (Fig. 4.11).
3. η was initialized with the identity function at thread creation (Fig. 4.12).
4. The composition operator $\bullet$ was applied upon return from fork to combine the evolution functions of the two topmost stack entries in a meaningful way (Fig. 4.13).
5. We also used an auxiliary function apply that combined child and parent permissions based on η (Fig. 4.13).

In the concrete setting, we do not have the concretization and abstraction steps, so we do not need step 1. If we simply add steps 2 to 4 to the rules of $\overset{P}{\Longrightarrow}$, we end up with a function that maps all edges e either to the singleton set $\{e\}$ (if e exists at thread creation) or to $\perp$ (if the edge was created by the child). Other set-valued results can only arise through abstraction and concretization steps, so they will not occur in the concrete semantics. Crucially, we therefore do not need step 5, which is the only step in which the η component influences the value of the q component.

In other words, adding steps 2 to 4 to $\overset{P}{\Longrightarrow}$ results in a semantics that behaves identically to $\overset{P}{\Longrightarrow}$ for the $\mathcal{H}$ and q component, but additionally keeps track of an independent η

component that does not interfere with the values of the other components. We have thus found a candidate for the semantics $\overset{e}{\Longrightarrow}$. In essence, the η component does nothing but tracking explicitly which edges have been added throughout the lifetime of the thread (by mapping them to $\perp$). This will be enough to easily relate the semantics to $\overset{\eta}{\Longrightarrow}$, which was not possible for $\overset{P}{\Longrightarrow}$.

Safe approximation of $\overset{e}{\Longrightarrow}$ by $\overset{\eta}{\Longrightarrow}$

By using the new bookkeeping semantics $\overset{e}{\Longrightarrow}$ as concrete semantics, the safe approximation proof can now be carried out in the same way as in Section 3.2.3. It is, of course, more technical and involved because of the additional complexity of the stack components. At the same time, the proof does not provide any additional insights. I will therefore not give the full proof, but instead only show that the same theoretical framework is applicable by defining extended abstraction and concretization functions that yield a Galois connection on the extended stacks. Since we use $\overset{e}{\Longrightarrow}$ as the concrete semantics, the concrete domain is extended by an η component, yielding abstraction and concretization functions α and γ of the following type.

$$\begin{array}{lcl} \alpha & : & 2^{((\mathbf{HC}^0\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+} \rightarrow 2^{((\mathbf{HC}\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+} \\ \gamma & : & 2^{((\mathbf{HC}\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+} \rightarrow 2^{((\mathbf{HC}^0\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+} \end{array}$$

Now that we have the additional η component in the concrete semantics, these functions can be defined by applying ppabst and ppconc instead of abst and conc (cf. page 51). We would like to define abstraction and concretization component-wise once again, but this will unfortunately not be possible because of the η component.

Let us first think about abstraction. To simplify matters, we will assume that each stack frame corresponds to a thread fork rather than a procedure call.[1] Let $(\mathcal{H}_1,q_1,\eta_1)$ be a concrete stack entry and $(\mathcal{H}_2,q_2,\eta_2) = \mathrm{ppabst}(\mathcal{H}_1,q_1,\eta_1)$. The result is a fully abstract graph $\mathcal{H}_2$ with correspondingly updated permission and evolution components. Is this enough? Not quite. The codomain of η_2 is still concrete.

It is tempting to arbitrarily abstract it, but recall that this is not done in the abstract semantics. Instead, the abstract semantics externalizes all nodes that cannot be abstracted by the parent thread. To replicate this behavior, we have to use the (partial) abstraction of the parent thread as codomain of η_2. This leads to a recursive definition of α. (I define it for single concrete stacks, the extension to sets of stacks is obtained by taking the union.)

$$\begin{array}{l} \alpha(\langle(\mathcal{H}_1,q_1,\eta_1)\rangle) = \mathrm{ppabst}(\mathcal{H}_1,q_1,\eta_1) \\ \alpha(\langle(\mathcal{H}_1,q_1,\eta_1),(\mathcal{H}_2,q_2,\eta_2),\ldots,(\mathcal{H}_k,q_k,\eta_k)\rangle) = \langle(\mathcal{H}'_1,q'_1,\eta'_1),(\mathcal{H}'_2,q'_2,\eta'_2),\ldots,(\mathcal{H}'_k,q'_k,\eta'_k)\rangle \\ \textbf{where } \langle(\mathcal{H}'_2,q'_2,\eta'_2),\ldots,(\mathcal{H}'_k,q'_k,\eta'_k)\rangle = \alpha(\langle(\mathcal{H}_2,q_2,\eta_2),\ldots,(\mathcal{H}_k,q_k,\eta_k)\rangle) \\ \qquad\quad (\mathcal{H}^*_1,\eta^*_1) = \mathrm{replacecodomain}(\eta_1,\mathcal{H}_2,q_2) \\ \qquad\quad (\mathcal{H}'_1,q'_1,\eta^*_1) = \mathrm{ppabst}(\mathcal{H}^*_1,q_1,\eta^*_1) \end{array}$$

where $\mathrm{replacecodomain}(\eta,\mathcal{H},q)$ is a function that

- Takes the intersection of $\mathcal{H}$ with the codomain of η
- Abstracts this intersection while respecting q (using ppabst)
- Marks all concrete nodes in the result as external
- Replaces the intersection with this new (partially) abstract graph to obtain $\mathcal{H}^*$

[1]It is straightforward to integrate calls, but this further complicates the definitions, since we need to handle η components differently for calls. I focus on forks, because they are the more interesting case w.r.t. η.

- Uses the abstraction as the new codomain of η by replacing each edge of the original codomain with the corresponding edge in the abstracted graph, yielding η^*
- Returns $(\mathcal{H}^*, \eta^*)$

I omit the formalization of $\mathrm{replacecodomain}$ for the sake of brevity. By defining α in this way, we keep the same nodes concrete that are kept concrete by the abstract semantics.

γ is defined in a similar way. Recall that in the sequential setting, we replaced each abstract graph $\mathcal{H}$ by the language of concrete graphs that it represents, $\mathcal{L}(\mathfrak{G}, \mathcal{H})$, where the language was defined as all concrete graphs that can be obtained from $\mathcal{H}$ via iterated concretization. It is straightforward to adapt the definition of languages to keep track of q and η—that is, to define languages $\mathcal{L}(\mathfrak{G}, \mathcal{H}, q, \eta)$, which contain the same graphs as $\mathcal{L}(\mathfrak{G}, \mathcal{H})$, but concretizes the graphs using ppconc, thereby keeping q and η in sync with the concretization of $\mathcal{H}$.

In addition, we need to make sure that the codomain of η matches a subgraph of the corresponding parent thread's heap. This can again be achieved through a recursive definition, but this time recursing on the initial part rather than the tail of the stack. (Again, I only give the definition for a single stack.)

$$\begin{aligned}
&\gamma(\langle(\mathcal{H}_1,q_1,\eta_1)\rangle) = \mathcal{L}(\mathfrak{G}, \mathcal{H}_1,q_1,\eta_1)\\
&\gamma(\langle(\mathcal{H}_1,q_1,\eta_1),\ldots,(\mathcal{H}_{k-1},q_{k-1},\eta_{k-1}),(\mathcal{H}_k,q_k,\eta_k)\rangle)\\
&\quad = \{\langle(\mathcal{H}'_1,q'_1,\eta'_1),\ldots,(\mathcal{H}'_{k-1},q'_{k-1},\eta'_{k-1}),(\mathcal{H}'_k,q'_k,\eta'_k)\rangle\}\\
&\textbf{where } \langle(\mathcal{H}'_1,q'_1,\eta'_1),\ldots,(\mathcal{H}'_{k-1},q'_{k-1},\eta'_{k-1})\rangle \in \gamma(\langle(\mathcal{H}_1,q_1,\eta_1),\ldots,(\mathcal{H}_{k-1},q_{k-1},\eta_{k-1})\rangle)\\
&\qquad\quad \eta^*_k = \mathrm{insertconcrete}(\eta_k, \mathcal{H}_{k-1}, \mathcal{H}'_{k-1})\\
&\qquad\quad (\mathcal{H}'_k,q'_k,\eta'_k) \in \mathcal{L}(\mathcal{H}_k,q_k,\eta^*_k)
\end{aligned}$$

Here, $\mathrm{insertconcrete}$ uses the concretization result of $\mathcal{H}_{k-1}$, which is available in $\mathcal{H}'_{k-1}$, as the codomain of η_k. (Formalization omitted.)

Proposition 4.9 $((2^{((\mathbf{HC}^0\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+}, \subseteq), \alpha, \gamma, (2^{((\mathbf{HC}\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+}, \subseteq))$ forms a **Galois connection**.

To prove this, we would need to show that

1. $\forall \mathfrak{H} \in 2^{((\mathbf{HC}^0\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+} . \mathfrak{H} \subseteq \gamma(\alpha(\mathfrak{H}))$
2. $\forall \mathfrak{H} \in 2^{((\mathbf{HC}\times\mathbf{Perms}\times\mathbf{Evo})\cup\{\mathbf{err}\})^+} . \alpha(\gamma(\mathfrak{H})) \subseteq \mathfrak{H}$

This is, of course, not possible in any meaningful way without formalizing $\mathrm{replacecodomain}$ and $\mathrm{insertconcrete}$, but informally, the conditions hold for the following reasons.

1. Abstraction is still unique. $\mathrm{replacecodomain}$ ensures that the evolution components are consistent and that nodes are externalized in the same way as in the abstract semantics. Hence α does not over-approximate the permission information q. Consequently, when subsequently applying γ, we will obtain the same concrete graph (plus many more) through the application of $\mathcal{L}$. Hence the first inequality holds.
2. In the application of γ, $\mathrm{insertconcrete}$ ensures that we do not generate any concrete stacks that are inconsistent. Hence the subsequent application of α to each concrete stack will return the original abstract stack, and the second inequality follows.

Thus, we again obtain a Galois connection. This reasoning is admittedly vague and incomplete, but I hope it is enough to convince you that it is possible to connect the domains of $\overset{e}{\Longrightarrow}$ and $\overset{\eta}{\Longrightarrow}$ via a Galois connection. I also hope that, given the complexity even in this partly informal setting, my choice to omit part of the formalization is excusable.

Having established the Galois connection the safe approximation proof itself is now a mere technicality, just like in Section 3.2.3: The abstract semantics is again just a wrapper, and γ

is defined in such a way that any concrete stack (and hence any concrete behavior in the unwrapped semantics) corresponds to an abstract stack and behavior.

Proposition 4.10 $\xRightarrow{\eta}$ safely approximates $\xRightarrow{e}$ and hence safely approximates $\xRightarrow{P}$.

This proposition concludes my theoretical study of programming language semantics. I will now turn to the application of these semantics in program analysis.

CHAPTER 5

Modular Computation of Procedure and Thread Contracts

In the previous chapter, we developed concrete and abstract programming language semantics based on hypergraphs and hyperedge replacement grammars. We also saw how these two semantics could be formally related using the theory of abstract interpretation. While this was insightful in its own right, it is in fact a means rather than an end: We are interested in the static analysis of PL programs. To make this effective, we need a way to only reason about finitely many executions of any given program. This was the purpose of developing an abstract semantics that provides a finite over-approximation of the program's semantics. (We saw that this was not always possible in the parallel case; I will return to this point later in this chapter.) By showing that the abstract semantics safely approximates the concrete semantics using the theory of abstract interpretation, we were able to show that our abstract semantics is a sound approximate of our concrete semantics. This justifies the use of the abstract semantics as basis for program analysis.[1]

In this chapter we construct such an analysis by instantiating a standard data-flow analysis framework for interprocedural analysis. More specifically, we will see how to generate abstract graph-based **procedure summaries**, as the title of Jansen and Noll's paper [JN14] puts it. A procedure summary is a form of **procedure contract** that relates preconditions to possible postconditions. In our case, both preconditions and postconditions will be abstract heap graphs. In other words, we will develop an analysis that computes for each procedure (and thread) and for each invocation of the procedure (or thread) with an abstract heap graph the possible heap graphs at the end of the execution. By doing this in a demand-driven way (i.e., by only considering preconditions that actually occur in the symbolic execution), we will obtain an effective algorithm based on fixed point computations.

This chapter is based on Jansen and Noll's paper [JN14], but the presentation is at times quite different for a couple of reasons. First, we build upon our prior formalization of the semantics, whereas the cited paper does not explicitly define a formal semantics. Second, I also incorporate the parallel semantics, whereas Jansen and Noll where operating only in the

[1]You could argue that the concrete semantics also needs justification: It is concrete in the terminology of abstract interpretation, but it is, in fact, already quite abstract in that it abstracts from the concrete memory layout. It would thus be interesting to show that it corresponds to a **store-based semantics** that closely reflects the program execution on a real machine. I took care to justify my concrete semantics, but it would still be interesting to try to relate it to a store-based semantics. See [BIL03] for more information on store-based vs. storeless semantics.

sequential setting. The underlying framework for **interprocedural data-flow analysis** (IPA) itself is adapted from Knoop and Steffen's work [KS92].

Let us first review some basic terminology and approaches. As the name suggests, **data-flow analysis** is a technique for analyzing how data flows through a given program. Such analyses can deal with all kinds of data as long as the underlying domain forms a complete lattice (defined in Appendix A for readers unfamiliar with order theory), as we will argue shortly. Canonical examples include available expressions analysis, live variable analysis, and constant propagation [NNH99]. As these examples suggest, data-flow analyses are often used in optimizing compilers. Our application, however, is in program correctness rather than program optimization. In this context, data-flow analyses can be useful in two ways:

- **Error detection.** The immediate purpose of our data-flow analysis will be to find possible memory errors such as null pointer dereferences and data races, and prove the absence of such errors.
- **State space generation.** The data-flow analysis implicitly computes an abstract state space for the program that, when explicitly generated, can be used as input for model checkers.

The emphasis of this thesis is on the former, though the latter is one possible direction for future work, as I suggest in Section 6.2. Due to the abstraction-induced over-approximation, we can, of course, not prove the absence of all memory errors for all programs: We will sometimes report errors that are only possible in an abstract trace, but not in any concrete program execution. On the other hand, if the analysis runs through without reporting errors, we can be sure that there are no errors—we can only have false positives, not false negatives.

The basic work-flow when designing a data-flow analysis for a programming language is as follows [NNH99].

1. Define a transformation of programs to **control flow graphs** (CFGs).
2. Derive an **equation system** that describes how the analysis information is propagated and transformed between the nodes of the CFG.
3. Perform **fixed point iteration** to solve the equation system.

Given the right mathematical properties—namely a monotone fixed point operator on a complete lattice of finite height—a fixed point exists and is reached after a finite number of iterations. A data-flow analysis is called **interprocedural** if it tracks data-flow information across procedure boundaries. Interprocedural analyses are both more complex and yield more complete analysis information [Aho+06]. Over the course of this chapter, I will develop such an interprocedural data-flow analysis for heap analysis based on HRGs, adapting the abstract semantics from Sections 3.2 and 4.2.

The remainder of this chapter is structured as follows. In Section 5.1 I describe the translation of PL programs to control flow graphs. I then develop a general interprocedural data-flow analysis framework in This framework was adapted from Knoop and Steffen [KS92] and underlies our analysis of PL. I instantiate this framework to our domain in the following section, Section 5.3. I discuss algorithmic issues in Section 5.4, before concluding with a short comparison of the hypergraph-based approach to separation logic in Section 5.5. If you find it difficult to make sense of the interprocedural analysis framework, you may first want to have a look at Appendix B to familiarize yourself with the intraprocedural case.

5.1 Compilation to Control Flow Graphs

As mentioned in the introduction to this section, we will compile PL into a control flow graph (CFG) representation and then later define our data-flow analysis in terms of this representation.

Definition 5.1 — **Control flow graph.** A directed graph $G = (N,F,s,e)$, where

- N is the set of nodes
- $F \subseteq N \times N$ is the **flow relation**, i.e., the set of edges of the CFG
- $s \in N$ is the **start node**
- $e \in N$ is the **end node**

is a **control flow graph** (CFG) if

- s does not have any incoming edge and e does not have any outgoing edge
- every node in N is reachable from s

We instantiate this generic definition for PL programs as follows. We compile each procedure into a CFG consisting of several types of nodes. Each node will be labeled with the PL command or Boolean expression it represents (if any) and a numerical label $i \in \mathbb{N}$ to guarantee uniqueness. In the following, we write $name(p)$ (or, somewhat sloppily, $name(c)$, where c is a procedure body) to refer to the name of a procedure.

- A unique **entry** and **exit** node for each procedure p, $entry_{name(p)}$, $exit_{name(p)}$. The purpose of these nodes is to guarantee isolated entry and exit and simplify the interprocedural semantics.
- One **call** and one **return** node per call statement c, $call_c^i$, $return_c^i$
- One **fork** and one **return from fork** node per fork statement c, $fork_c^i$, $rfork_c^i$
- **filter** nodes for dealing with the branching of **if** and **while** induced by their Boolean conditions b, $filter_b^i$ and $filter_{\neg b}^i$
- Generic **procedure-local** nodes, loc_c^i, for all atomic statements c, i.e., skip, variable declarations, assignments, and memory allocation, as well as for each **join** node. (In our deterministic semantics join statements are local in the sense that all non-local information was already computed at the return-from-fork point.)

Splitting the call and fork nodes reflects the corresponding split in the program semantics. Compiling branches into filters for the positive and negative condition also makes sense, as the analysis will generate sets of postconditions. (Just like the $\mathbf{nextC}$ and $\mathbf{nextA}$ functions were operating on sets of heaps.)

Let us now formalize the definition of CFGs in terms of these types of nodes. To be able to do so recursively, we will use two auxiliary functions in the definition of the flow relation: An $init$ function that collects the incoming nodes of a (block) statement c and a $final$ function that collects the outgoing nodes.[1] We thus define four functions: $nodes(c)$ determines the set of all nodes of the CFG for command c, $flow(c)$ defines the set of all edges, and uses the auxiliary functions $init(c)$ and $final(c)$ to do so.

Let p be a procedure with body $c \in \mathbf{Cmd}$ and let $\ell : \mathbf{Cmd} \to \mathbb{N}$ be a labeling function that assigns a unique number (label) to each local statement in c.[2] Then the CFG for p with

[1] I adapted this formalization of control-flow graphs from [Nol15].

[2] For example obtained by traversing the abstract syntax tree and assigning consecutive numbers.

respect to ℓ is

$$\begin{aligned}
&G = (N, F, entry_p, exit_p)\\
&\textbf{where}\\
&\quad N = \{entry_p, exit_p\} \dot\cup nodes(c)\\
&\quad F = flow(c) \dot\cup \{(entry_p, n) \mid n \in init(c)\} \dot\cup \{(m, exit_p) \mid m \in final(c)\}
\end{aligned}$$

where $nodes$, $init$, $final$, and $flow$ are defined recursively on the structure of c as follows.

$$nodes(c) := \begin{cases}
\{loc_c^{\ell(c)}\}, & \text{if } c \text{ is local}\\
\{call_c^{\ell(c)}, return_c^{\ell(c)}\}, & \text{if } c = \textbf{call } p(x_1, \ldots, x_n)\\
\{fork_c^{\ell(c)}, rfork_c^{\ell(c)}\}, & \text{if } c = \textbf{ fork } p(x_1, \ldots, x_n)\\
nodes(c_1) \dot\cup nodes(c_2), & \text{if } c = c_1; c_2\\
\{filter_b^{\ell(c)}, filter_{\neg b}^{\ell(c)}\} & \\
\dot\cup nodes(c_1) & \text{if } c = \textbf{while } b \textbf{ do } c_1\\
\{filter_b^{\ell(c)}, filter_{\neg b}^{\ell(c)}\} & \\
\dot\cup nodes(c_1) \dot\cup nodes(c_2), & \text{if } c = \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2
\end{cases}$$

$$init(c) := \begin{cases}
nodes(c), & \text{if } c \text{ is local}\\
\{call_c^{\ell(c)}\}, & \text{if } c = \textbf{call } p(x_1, \ldots, x_n)\\
\{fork_c^{\ell(c)}\}, & \text{if } c = \textbf{ fork } p(x_1, \ldots, x_n)\\
init(c_1), & \text{if } c = c_1; c_2\\
\{filter_b^{\ell(c)}, filter_{\neg b}^{\ell(c)}\} & \text{if } c = \textbf{while } b \textbf{ do } c_1\\
\{filter_b^{\ell(c)}, filter_{\neg b}^{\ell(c)}\}, & \text{if } c = \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2
\end{cases}$$

$$final(c) := \begin{cases}
nodes(c), & \text{if } c \text{ is local}\\
\{return_c^{\ell(c)}\}, & \text{if } c = \textbf{call } p(x_1, \ldots, x_n)\\
\{rfork_c^{\ell(c)}\}, & \text{if } c = \textbf{ fork } p(x_1, \ldots, x_n)\\
final(c_2), & \text{if } c = c_1; c_2\\
\{filter_{\neg b}^{\ell(c)}\} & \text{if } c = \textbf{while } b \textbf{ do } c_1\\
final(c_1) \dot\cup final(c_2) & \text{if } c = \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2
\end{cases}$$

Now we define the flow relation $flow : \mathbf{Cmd} \to nodes(c) \times nodes(c)$ in terms of $init$ and $final$.

$$flow(c) := \begin{cases}
\emptyset, & \text{if } c \text{ is local}\\
\emptyset, & \text{if } c = \textbf{call } p(x_1, \ldots, x_n)\\
\emptyset, & \text{if } c = \textbf{ fork } p(x_1, \ldots, x_n)\\
\{(n, m) \mid n \in final(c_1), m \in init(c_2)\}, & \text{if } c = c_1; c_2\\
\{(filter_b^{\ell(c)}, init(c_1))\} \dot\cup flow(c_1) & \\
\dot\cup \{(n, filter_\beta^{\ell(c)}) \mid n \in final(c), \beta \in \{b, \neg b\}\} & \text{if } c = \textbf{while } b \textbf{ do } c_1\\
\{(filter_b^{\ell(c)}, init(c_1)), (filter_{\neg b}^{\ell(c)}, init(c_2))\} & \\
\dot\cup flow(c_1) \dot\cup flow(c_2), & \text{if } c = \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2
\end{cases}$$

There are a couple of things to note. First, there is no edge from call or fork nodes to the corresponding return nodes. This makes sense: They will only be connected via the called procedure in the interprocedural flow graphs, which we will define in Section 5.2.

Second, the control flow for while loops seems strange at first. This is because—just like for if branches—we introduce two filter nodes, one for the condition and one for its negation. At the end of the loop body, we therefore have two outgoing edges, one to each filter node. We leave the loop only via the negated filter, i.e, when the loop condition is violated. In many formalizations of control flow graphs, only a single node is introduced for each condition. In that case, however, we either need to add semantics to the outgoing edges to (such as a "Yes" or "No" label indicating whether the edge corresponds to positive evaluation of the condition) or do not evaluate conditions in their nodes at all but rather instrument the code with assertion statements at the beginning of each branch [Nol15]. I chose a different approach to avoid both explicit assert statements and the need to add semantics to edges.

We will need a way to combine the individual CFGs for each procedure into a CFG for interprocedural and also for concurrent programs. Especially in the concurrent setting, many different flavors of control-flow graphs have been proposed, see for example [LC89; DS91; NAC99]. Thanks to our deterministic semantics for concurrent programs, we do not need to take any special care to model concurrency, however; a formalism for sequential interprocedural programs suffices. When we compile each procedure independently via the above compilation scheme, taking care not to reuse labels, we get a system of control flow graphs.

Definition 5.2 — **System of control flow graphs (SCFG).** A **system of control flow graphs** or **SCFG** is a tuple $(G_0, \dots, G_{n-1})$ for a program $P = (p_0, \dots, p_{n-1})$, where $N_i \cap N_j = \emptyset$ for all $i \neq j$.

To obtain an operational interpretation of an SCFG, we need to connect the individual CFGs, which we do as follows.

Definition 5.3 — **Interprocedural flow graph (IFG).** Let $(G_0, \dots, G_{n-1})$ be an SCFG, where for each i, $G_i = (N_i, F_i, s_i, e_i)$ and where G_0 represents the main procedure. The **interprocedural flow graph** (IFG) for $(G_0, \dots, G_{n-1})$ is the tuple

$$G^* = (N^*, F^*, s^*, e^*)$$

where the components are computed from the SCFG as follows

- $N^* := \bigcup_{i \in \{0,\dots,n-1\}} N_i$ (Combined node set)
- $$\begin{aligned} F^* := &\bigcup\{F_i \mid i \in \{0, \dots, n-1\}\} \cup \\ &\bigcup\{(call^i_c, entry_{name(c)}) \mid call^i_c \in N^*\} \cup \\ &\bigcup\{(fork^i_c, entry_{name(c)}) \mid fork^i_c \in N^*\} \cup \\ &\bigcup\{(exit_{name(c)}, return^i_c) \mid return^i_c \in N^*\} \cup \\ &\bigcup\{(exit_{name(c)}, rfork^i_c) \mid rfork^i_c \in N^*\} \end{aligned}$$
  (Interprocedural flow relation)
- $s^* := s_0, e^* := e_0$ (Main entry and exit)

In other words, we combine the SCFG into a single interprocedural flow graph through an interprocedural flow relation F^* that extends the individual flow relations with interprocedural edges.

R Depending on the analysis we are interested in, it may be necessary to distinguish between local procedure calls and thread forks for the same procedure p. In that case it would not be allowed to indiscriminately connect exit nodes to call and fork nodes in the interprocedural flow relation; rather, we would need two copies of each flow graph, one for local calls and one for forks.

At first it seems like this holds for our analysis, too: In thread forks, $\mathrm{pending}$ and η are reset, whereas they remain unchanged in local procedure calls. Upon we return we must therefore not merge fork analysis information with call analysis information. Since we compute procedure contracts, we will fortunately not have this problem in our analysis: Upon return (from either call or fork), we will only select contracts with preconditions that have the expected $\mathrm{pending}$ and η information and thus automatically discard analysis information of the wrong type. I will return to this point in Section 5.3.3.

We will often have to refer to specific parts of the IFG, so we introduce some additional notation (which you do not have to memorize to follow the remainder of the chapter).

- $N_C := \{n \in N^* \mid \exists i \exists c.n = call_c^i \vee n = fork_c^i\}$ (Set of call and fork nodes)
- $N_R := \{n \in N^* \mid \exists i \exists c.n = return_c^i \vee n = rfork_c^i\}$ (Set of return and return-from-fork nodes)

Finally, it will be useful to have some notation for looking up specific relations between nodes. We define:

$$\begin{array}{llllll}
callee & : & N_C \to N_E & \text{as} & callee(call_c^i) := entry_{name(c)} \\
 & & & & callee(fork_c^i) := entry_{name(c)} \\
return & : & N_C \to N_R & \text{as} & return(call_c^i) := return_c^i \\
 & & & & return(fork_c^i) := rfork_c^i \\
caller & : & N_X \to 2^{N_R} & \text{as} & caller(exit_p) := \{call_c^i \in N_C, \mid name(c) = p\} \\
 & & & & \quad \dot\cup \{fork_c^i \in N_C, \mid name(c) = p\} \\
pred & : & N^* \to 2^{N^*} & \text{as} & pred(n) := \{m \mid (m,n) \in F^*\} \\
succ & : & N^* \to 2^{N^*} & \text{as} & succ(n) := \{m \mid (n,m) \in F^*\}
\end{array}$$

R The definitions in these section are quite similar to the ones in [KS92]. Apart from the additional nodes for fork and join, the only difference is that I already split call (and fork) nodes in two in the compilation scheme. They are therefore already split in the SCFG and do not need to be split in the IFG.

With all these definitions in place, we can finally formalize our data flow analysis framework.

5.2 A Formalization of Interprocedural Data-flow Analysis

Before I dive in, I would like to remind you that Appendix B develops a data-flow analysis framework for the simple setting of intraprocedural analyses. If you are not familiar with data-flow analyses, you may want to read that appendix before reading on.

In the following, let $(D, \sqsubseteq)$ be a complete lattice with LUB operator $\sqcup$, least element $\bot$, and top element $\top$. In the following, we will use D as the domain for the analysis information. Let $G^* = (N^*, F^*, s^*, e^*)$ be the interprocedural flow graph of the program that we would like to analyze. Our goal thus is to analyze how data flows through this graph. Intuitively, the most precise solution that any such data-flow analysis can possibly compute is a solution that takes into account for each CFG node all execution paths that lead to the node but nothing else. Formally, we would like to compute the **meet over all paths** solution.

Paths

To make this precise, we must first formalize the paths through the IFG.

Definition 5.4 — **Interprocedural path.** A sequence of nodes $\pi = \langle n_1, \ldots, n_k \rangle$ is an **interprocedural path** through G^* iff $(n_i, n_{i+1}) \in F^*$ $\forall i \in \{1, \ldots, k-1\}$.

Analogously to the intraprocedural case, we also define the set of all paths as well as the set of all paths between a pair of nodes.

$$\begin{aligned} \mathbf{IP} &:= \{\langle x_1, \ldots, x_j \rangle \in N^* \mid \forall i \in \{1, \ldots, j-1\}.(x_i, x_{i+1}) \in F^*\} \\ \mathbf{IP}_n^m &:= \{\pi \in \mathbf{IP} \mid \pi = \langle n, x_2, \ldots, x_{j-1}, m \rangle\} \end{aligned}$$

Note that not every interprocedural path actually corresponds to an execution of the program. In particular, it is not enforced that we return to the caller upon method return. This leads to the notion of **valid interprocedural paths** that respect the nesting of calls.

Definition 5.5 — **Valid interprocedural path.** An interprocedural path $\pi = \langle n_1, \ldots, n_k \rangle$ is called **valid** if the **node sequence** $\langle n_1, \ldots, n_k \rangle$ is **well-formed**. $\langle n_1, \ldots, n_k \rangle$ is well-formed if either of the following conditions holds

1. if $\langle n_1, \ldots, n_k \rangle$ does not contain any $return$ or $rfork$ node, i.e., $n_i \notin N_R \forall i \in \{1, \ldots, k\}$, $\langle n_1, \ldots, n_k \rangle$ is well-formed
2. if $\langle n_1, \ldots, n_k \rangle$ does contain a $return$ or $rfork$ node, let j be the smallest j such that n_j is such a node. Further, let $i < j$ be the largest index such that $n_i \in N_C$. If n_i exists and **matches** n_j (i.e., $n_i \in caller(n_j)$), then $\langle n_1, \ldots, n_k \rangle$ is well-formed iff

$$\langle n_1, \ldots, n_{i-1}, n_{i+1}, \ldots, n_{j-1}, n_{j+1}, \ldots, n_k \rangle$$

is well-formed.

In accordance with the notational conventions used so far, we write $\mathbf{VIP}$ and $\mathbf{VIP}_n^m$ for the set of all valid paths and the set all valid paths between n and m, respectively.

R Observe that in Def. 5.5, the shorter sequence is in general **not** an interprocedural path as per Def. 5.4 because of the gaps we introduce in the sequence. Hence the distinction between well-formed **node sequences** and valid **paths**.

R Note that Def. 5.5 only works because we assumed that all procedures are either exclusively called locally or exclusively forked; otherwise we would have to take care not to match $call$ nodes with $rfork$ nodes or $fork$ nodes with $return$ nodes.

Finally, to properly define the semantics of procedure calls and thread forks, we are interested in paths with an equal number of calls and returns, i.e., paths where all invoked procedures and threads have returned. We call such paths **complete**.[1]

Definition 5.6 — **Complete interprocedural path (CIP) [KS92].** A valid interprocedural path $\pi = \langle n_1, \ldots, n_k \rangle$ is called complete if $n_1 \in N_E$ and

$$|\{i \mid n_i \in N_C\}| = |\{i \mid n_i \in N_R\}|$$

We write $\mathbf{CIP}$ and $\mathbf{CIP}_n^m$ for the set of all CIPs and the set all CIPs between n and m.

[1]Even if a thread is forked but never joined, the execution corresponds to a complete path in our semantics, because the thread's semantics will nevertheless be computed in its entirety at the return-from-fork.

Stacks

Recall that we assume that the analysis information is taken from a complete lattice $(D, \sqsubseteq)$. In the interprocedural case, we have to track this information across procedure boundaries. To this end, we will use stacks of elements as domain for the interprocedural data-flow information.

Definition 5.7 — **Stacks over lattices.** Let $(D, \sqsubseteq)$ be a complete lattice. We then define the stack over D, $\mathbf{Stk}(D) := (D^S, \sqsubseteq^S)$, where

- $D^S := D^+ \cup \{\bot^S\} = \bigcup_{i \in \mathbb{N}_{>0}} D^i \cup \{\bot^S\}$
- Let $d,c \in D^S$. $d \sqsubseteq^s c$ if and only if at least one of the following conditions holds
  1. $d = \bot^s$
  2. $d = c$
  3. $d = \langle d_1, \ldots, d_i \rangle \wedge c = \langle c_1, \ldots, c_j \rangle \wedge j = 1 \wedge d_1 \sqsubseteq c_1$

Lemma 5.8 — **Stacks over lattices are complete lattices.** If $(D, \sqsubseteq)$ is a complete lattice, then $\mathbf{Stk}(D) := (D^S, \sqsubseteq^S)$ is a complete lattice.

Proof. Clearly, $\sqsubseteq^S$ is a partial order. To show that $\mathbf{Stk}(D)$ is a complete lattice, it suffices to show that all subsets of D^S have a least upper bound. (This is a standard result from order theory, cf. Appendix A.) Clearly, $\bigsqcup \emptyset = \bot^S$ and $\bigsqcup\{d\} = d$.

For the remaining cases, consider $X \subseteq D^S$ with $|X| \geq 2$. W.l.o.g., $\bot^S \notin X$, because $\bigsqcup(\{\bot^S\} \cup X) = \bigsqcup X$ for all $X \sqsubset D^S$. Let

$$d^{lub} := \bigsqcup\{d_1 \mid \langle d_1, \ldots \rangle \in X\}$$

This bound exists, because D is a complete lattice. Note that

- For each $\langle d_1, \ldots \rangle \in X$, $d_1 \sqsubseteq d^{lub}$ and hence, by definition of $\sqsubseteq^S$, $\langle d_1, \ldots \rangle \sqsubseteq^S \langle d^{lub} \rangle$
- Conversely, let $\langle c_1, \ldots, c_k \rangle$ be an upper bound of X. Clearly, $k = 1$, because X contains at least two different stacks, and stacks are only comparable to themselves and to one-element stacks. But if $k = 1$, we have $d^{lub} \sqsubseteq c_1$, because $\langle c_1 \rangle$ is an upper bound for X and hence $d_1 \sqsubseteq c_1$ for all $\langle d_1, \ldots \rangle \in S$. Consequently, $d^{lub} = \bigsqcup\{d_1 \mid \langle d_1, \ldots \rangle \in S\} \sqsubseteq c_1$. Thus, $\langle d^{lub} \rangle \sqsubseteq^S \langle c_1, \ldots, c_k \rangle$.

Hence $\langle d^{lub} \rangle$ is the least upper bound of X. ■

In other words, the least upper bound of a set of (at least two) stacks is the one-element stack consisting of the least upper bound over the top elements of the stacks. Defining the stack lattice in this way makes sense: Throughout our analysis, the topmost stack element will contain the relevant analysis information at any given point. Other stack entries are just necessary for bookkeeping across procedure boundaries—for example, to update the analysis information at procedure return. I will return to this point throughout this section.

R Throughout the remainder of this chapter, I will always assume that the result of the LUB operation is a one-element stack, although this is not true when we apply it to a singleton set, where the LUB is the stack itself. We can formally deal with such cases by introducing a widening operator ∇ that projects the LUB onto the one-element stack consisting of the first element. This is sound, because for all stacks $d = \langle d_1, \ldots, d_k \rangle$, it holds that $\langle d_1, \ldots, d_k \rangle \sqsubseteq^S \langle d_1 \rangle$. To simplify the exposition, I gloss over this detail.

Before we go on, however, we define the following (partial) functions on stacks for convenience.

$$\begin{array}{lllllll}
\mathit{newstack} & : & D \to D^S & \text{as} & \mathit{newstack}(d) & := & \langle d \rangle \\
\mathit{top} & : & D^S \to D & \text{as} & \mathit{top}(\langle d_1, \dots, d_k \rangle) & := & d_1 \\
\mathit{push} & : & D^S \to D \to D^S & \text{as} & \mathit{push}(\langle d_1, \dots, d_k \rangle, d) & := & \langle d, d_1, \dots, d_k \rangle \\
\mathit{pop} & : & D^S \to D^S & \text{as} & \mathit{pop}(\langle d_1, d_2, \dots, d_k \rangle) & := & \langle d_2, \dots, d_k \rangle
\end{array}$$

Interprocedural data-flow systems

We can now lift the standard notion of data-flow systems (as, for example, developed in Appendix B) to stack lattices and define data-flow analyses on interprocedural flow graphs.

Definition 5.9 — **(Distributive) interprocedural data-flow system.** An **interprocedural data-flow system** over D is a tuple $S = (G^*, \Phi, \Xi, \Psi, \iota)$, where

- G^* is an IFG
- $\Phi = \{\varphi_n : D \to D \mid n \in N^* \setminus (N_C \cup N_R)\}$ is a set of monotonic transfer functions for the procedure-local nodes
- $\Xi = \{\xi_n : D \to D \mid n \in N_C\}$ is a set of monotonic call (and fork) transfer functions
- $\Psi = \{\psi_n : D \times D \to D \mid n \in N_R\}$ is a set of monotonic transfer functions for procedure (and fork) return
- $\iota : D$ is the extremal value

If all individual φ_n, ρ_n, and ξ_n are distributive, we call S distributive.

The φ_n functions define the meaning of intraprocedural statements (which only transform the current top element and in our case include join statements), the ξ_n functions define how the current top element of the stack is transformed into analysis information for the called procedure, and the ψ_n functions define how analysis information is merged upon procedure return. Such a data-flow system thus gives rise to a local semantic functional that operates on stacks of analysis information, $\mathit{local} : N^* \to \mathbf{Stk}(D) \to \mathbf{Stk}(D)$.

$$\mathit{local}(n)(\mathit{stk}) = \begin{cases} \mathit{push}(\mathit{pop}(\mathit{stk}), \varphi_n(\mathit{top}(\mathit{stk}))), & \text{if } n \in N^* \setminus (N_C \cup N_R) \\ \mathit{push}(\mathit{stk}, \xi_n(\mathit{top}(\mathit{stk}))), & \text{if } n \in N_C \\ \mathit{push}(\mathit{pop}(\mathit{pop}(\mathit{stk})), & \\ \qquad \psi_n(\mathit{top}(\mathit{pop}(\mathit{stk})), \mathit{top}(\mathit{stk}))) & \text{if } n \in N_R \end{cases}$$

local defines how stacks of data-flow information flow through the interprocedural flow graph. Based on this definition, we can adapt the standard notions of **meet over all paths** (MOP) and **maximal fixed point** solution to our interprocedural setting.

Interprocedural meet over all paths (IMOP) solution

Given the local semantic functional, it is straightforward to define a functional that captures the effect of paths through function composition.

$$\mathit{pathlocal}(\pi) := \begin{cases} \mathit{id}, & \text{if } \pi = \epsilon \\ \mathit{local}(x_j) \circ \mathit{pathlocal}(\langle x_1, \dots, x_{j-1} \rangle), & \text{if } \pi = \langle x_1 \dots, x_j \rangle \end{cases}$$

This function is still local in the sense that it only defines the semantics of a single path. Our goal is, however, to reason about all paths leading up to a node. This leads to the definition of the **interprocedural meet over all paths** (IMOP) solution.

$$\mathrm{imop}(\Phi)(n) := \bigsqcup\{\mathit{pathlocal}(\pi)(\mathit{newstack}(\iota)) \mid \pi \in \mathbf{VIP}^n_{s*}\}$$

We take the least upper bound over all paths leading up to the node of interest. Crucially, the stack lattice was defined in such a way that this is the least upper bound over all top elements of all the $\mathit{local}(\pi)(\mathit{newstack}(\iota))$ stacks.[1] Hence we only consider the relevant topmost analysis information.

Unsurprisingly, it is in general not possible to compute the IMOP solution effectively.

Corollary 5.10 — **The IMOP solution is undecidable.**

Proof. The IMOP solution degenerates to the MOP solution in the intraprocedural case, which is already undecidable [NNH99]. ■

Interprocedural maximal fixed point (IMFP) solution

Just like in the intraprocedural setting, we can perform fixed-point iteration to approximate the IMOP solution. Our goal is to perform a fixed-point iteration for the analysis domain $\mathbf{Stk}(D)$. To this end, we need to provide a transformer of type $\mathbf{Stk}(D) \to \mathbf{Stk}(D)$ for all nodes in the CFG. We already have such transformers for all procedure-local nodes of the CFG. We do, however, not have them for call or fork statements, since their meaning is global in the sense that we have to compute (or approximate) the meaning of the entire corresponding procedure body. More specifically, the meaning of a procedure is a (partial) function from stacks to stacks whose top element is changed. We shall call this domain $\mathcal{F}_{\mathbf{Stk}(D)}$.

$$\mathcal{F}_{\mathbf{Stk}(D)} := \{f : \mathbf{Stk}(D) \dashrightarrow \mathbf{Stk}(D) \mid \mathit{pop}(f(\mathit{stk})) = \mathit{pop}(\mathit{stk})\ \forall \mathit{stk} \in \mathbf{Stk}(D)\}$$

Computing partial functions is sufficient: As long as the procedure's meaning is defined for all inputs that occur in the program's execution, the result is well-defined.

We will thus define two equation systems:

1. An equation system operating on the function domain $\mathcal{F}_{\mathbf{Stk}(D)}$, whose fixed point will assign meaning to procedures
2. An equation system operating on $\mathbf{Stk}(D)$ that uses the transfer functions Φ of the data-flow system for local nodes and the fixed point of the first equation system as transfer functions for call and fork nodes

The idea behind the first system will be to assign to each node in the CFG the function of the containing procedure's effect up to that node. The function at the procedure's exit node then defines the semantics of the entire procedure. The equation system takes the

[1]Recall that this is technically only true if there are at least two different stacks at node n, but we assume that a widening operator for projecting onto the first component of the LUB is applied otherwise.

following form.

$$local^*(n) = \begin{cases} local(return(n)) \circ global(end(callee(n))) \circ local(n), & \text{if } n \in N_C \\ \bot, & \text{if } n \in N_R \\ local(n), & \text{otherwise} \end{cases}$$

$$global(n) = \begin{cases} id, & \text{if } n \in N_E \\ \bigsqcup\{local^*(m) \circ global(m) \mid m \in pred(n)\}, & \text{otherwise} \end{cases}$$

Here $\sqcup$ (applied to functions) represents the point-wise application of $\sqcup$ to the stacks in the codomain.

$global(n)$ represents the procedure effect up to node n, while $local^*(n)$ approximates the "local" semantics of procedure calls with respect to the current approximation of the global semantics. Note that we want a local function in the sense that we want a function that does not modify the stack height. This is achieved by computing the whole procedure effect for call nodes, while leaving the function undefined for return nodes.

Since $\mathcal{F}_{\mathbf{Stk}(D)}$ inherits the lattice structure from $\mathbf{Stk}(D)$ (by pointwise application of $\sqsubseteq^S$ to the codomain), it makes sense to ask for the least fixed point of this mutually recursive equation system. The least fixed point, once found, determines the procedure semantics: $global(exit_p)$ becomes the meaning of procedure p.

We define a second equation system to compute the combined effect of all procedure calls starting from the initial data-flow information ι. To this end, we set up a second equation system that refers back to the first equation system to determine the semantics of call statements.

$$\alpha_{n_i} = \begin{cases} newstack(\iota), & \text{if } n_i = s^* \\ \bigsqcup\{local(m)(\alpha_m) \mid m \in caller(n_i)\} & \text{if } n_i \in N_E \setminus \{s^*\} \\ \bigsqcup\{local^*(m)(\alpha_m) \mid m \in pred(n_i)\}, & \text{otherwise} \end{cases}$$

The **interprocedural maximal fixed point** (IMFP) solution is the least fixed point of this second equation system. We write $\mathrm{ifix}(S)$ for this fixed point.

Theorem 5.11 The least fixed point of α is effectively computable for finite lattices D.

Proof. Given a solution for $local^*$, the solution for α is obviously reached after finitely many iterations if D is finite.

Note that all calls to $local^*$ in α are made for stacks of height 1, as the analysis starts from a stack of height 1 ($newstack(\iota)$) and the analysis information in α always remains at height 1 by definition of the least upper bound operator $\sqcup$.

If we compute the fixed points of $local^*$ and $global$ on demand—that is, only for input values that occur in the fixed-point iteration α—we will therefore only need to look at stacks of height at most 2 throughout the fixed-point iteration for $local^*$ and $global$: A second element is added to stacks when $local$ is applied to a call node, but since the functions in the domain $\mathcal{F}_{\mathbf{Stk}(D)}$ do not change the stack height, no stacks with height larger than 2 can occur. But if D is finite, there are only finitely many functions in $\mathcal{F}_{\mathbf{Stk}(D)}$ that only involve stacks of height at most 2. (We can, for example, view each such function as a finite sets of key–value pairs, and the size of these sets is bounded by the size of D.) Hence the on-demand fixed-point iteration of $local^*$ and $global$ terminates as well. ■

Interprocedural Coincidence Theorem

We have thus developed two solutions for interprocedural data-flow analysis: The undecidable but optimal IMOP solution and the effectively computable but approximate IMFP solution. We shall now investigate the relationship between the two solutions. It makes sense that the IMFP solution approximates the IMOP solution: It is based on the same (monotonic) transfer functions, but we perform additional LUB computations and only operate on stacks of height at most two, so we must expect to lose precision. Perhaps surprisingly, we can show that the IMFP and IMOP coincide if all transfer functions in the data-flow system $S = (G^*,\Phi,\Xi,\Psi,\iota)$ are distributive. This is summarized in the following theorems.

Theorem 5.12 — **Interprocedural safety theorem [KS92].** Let S be an interprocedural data-flow system (with monotonic transfer functions as per Def. 5.9). Then

$$\forall n \in N^*.\, \text{ifix}(S)(n) \sqsubseteq \text{imop}(S)(n)$$

Theorem 5.13 — **Interprocedural coincidence theorem [KS92].** Let S be a distributive interprocedural data-flow system. Then the IMFP and IMOP solutions coincide, i.e.,

$$\forall n \in N^*.\, \text{ifix}(S)(n) = \text{imop}(S)(n)$$

Both results are quite technical but not difficult. If you are interested in the details, have a look at [KS92].[1]

5.3 Adaptation of the IPA Framework to the Analysis of PL_{seq}

In this section we will see how to define an interprocedural data-flow system for computing **procedure contracts** based on the abstract semantics of PL_{seq}. I start with PL_{seq}, because the abstract state space from Section 3.2 is much simpler and because it is finite. It will therefore be relatively straightforward to use the IPA framework for the abstract interpretation of PL_{seq}. We turn to parallel PL programs in the next section, Section 5.3.3.

5.3.1 The (Sequential) Procedure Contract Domain

The first step when designing a data-flow analysis is the definition of a suitable domain. The basic goal of our analysis will be to compute **procedure contracts**. A procedure contract is a mathematical object that relates a procedure's preconditions, i.e., some specification of the calling environment, to its postconditions, i.e., the (possible) cumulative effect(s) of the procedure on the precondition. Contract-based reasoning has been used extensively in the area of formal verification to facilitate modular verification techniques; consider, for example, rely/guarantee reasoning [Sta85], where we rely on properties of the environment

[1] Here are a few pointers to help you map my definitions to their setting. I opted for naming my functionals $local$, $local^*$, $global$ rather than $[\![\cdot]\!]$, $[\![\cdot]\!]^*$, $[\![[\cdot]]\!]$. I was hoping to increase readability in this way, but am not sure that I actually succeeded.

By introducing interprocedural data-flow systems as basis for $local$, I can make do without explicitly defining s-monotonicity and s-distributivity.

I deal with fork and join statements as if they were call statements and intraprocedural statements, respectively. The IPA framework is therefore still applicable—as long as I manage to define an interprocedural data-flow system with appropriate monotonic (or distributive) functions for fork and join.

(precondition) and make guarantees about how the procedure will affect said environment (postconditions). Similar techniques have also found their way into the software engineering domain as **design by contract** [Mey88] .

R The term **contract** is ambiguous. Let me emphasize that in this thesis, contracts are **not** annotations written by humans, as is usually the case in design by contract. Instead, a contract is an auto-generated summary of a procedure's effect (expressed as a transformation of abstract hypergraphs).

Which form should the pre- and postconditions take in our setting? Our program semantics are defined in terms of abstract hypergraphs. A procedure's effect can therefore be viewed as a (possibly non-deterministic) transformation of abstract heap configurations. Our contracts reflect this point of view. As preconditions, we simply use heap configurations that characterize the (abstract) calling context. In case of successful execution of the procedure, the corresponding postcondition is the set of heap configurations that may result from executing the procedure on the precondition heap. There are two reasons why the postcondition is a set of heaps rather than a single heap configuration. First, the abstract semantics that underlies our analysis is already non-deterministic. Second, when we compute our contracts via the IPA framework, we will have to perform least upper bound computations. This constitutes an additional source of over-approximation that may also introduce non-determinism.

In addition to a representation of successful computations, we also need to take the possibility of errors into account. After all, our main motivation for using data-flow analysis is our desire to find errors in the analyzed program. If a given precondition led to a (potential) error, the corresponding postcondition will consist exclusively of this error, even if there are successful paths through the procedure that start with the precondition. (This is not strictly necessary for soundness as long as we retain the information that an error occurred throughout the remaining analysis, but it both simplifies and speeds up the analysis.)

In general, there may, of course, be many valid preconditions for any given procedure. We therefore model procedure contracts as partial functions from preconditions to postconditions. A procedure contract thus is an element of the domain $\mathfrak{C} := \mathbf{HC}_{\mathrm{T},\Sigma} \dashrightarrow 2^{\mathbf{HC}_{\mathrm{T},\Sigma}} \cup \{\mathbf{err}\}$. The reason for opting for partial instead of total functions will become clear shortly; this design choice is due to our aim of computing contracts on demand.

Let $\mathfrak{c} : \mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}$ be a contract. (I shall from now on omit the T and Σ parameters). We write $\mathrm{pre}(\mathfrak{c})$ to refer to the domain of $\mathfrak{c}$, i.e., the set of all preconditions for which the contract is defined. Since $\mathfrak{c}$ is a function, we simply write $\mathfrak{c}(\mathcal{H})$ to access the postcondition for the precondition $\mathcal{H}$. We will sometimes need to refer to the set of all postconditions of a contract, i.e., its codomain. We then write $\mathrm{post}(\mathfrak{c})$. Note that $\mathrm{pre}(\mathfrak{c}) \in \mathbf{HC}$ and $\mathrm{post}(\mathfrak{c}) \in 2^{\mathbf{HC}} \cup \{\mathbf{err}\}$.

For the IPA to be applicable to contracts, we need to find a partial order $\sqsubseteq$ such that $(\mathfrak{C}, \sqsubseteq)$ forms a complete lattice. First, we need a partial order on the postconditions: Let $s_1,s_2 \in 2^{\mathbf{HC}} \cup \{\mathbf{err}\}$. We define

$$s_1 \sqsubseteq_P s_2 :\Longleftrightarrow s_2 = \mathbf{err} \vee s_1 \subseteq s_2$$

This order corresponds to our expectations of a **may analysis**—that is, an analysis where we take the union rather than the intersection off the predecessor information: A larger set of postconditions constitutes more complete analysis information, and, because errors must be propagated, they should be regarded as the largest possible analysis information. Since $(2^{\mathbf{HC}}, \subseteq)$ is a complete lattice (cf. Appendix A), so is $(2^{\mathbf{HC}} \cup \{\mathbf{err}\}, \sqsubseteq_P)$.

The induced least upper bound $\sqcup_p$ is simply union (if the compared elements are sets of hypergraphs) or **err** (if at least one of the operands of $\sqcup_p$ is **err**).

By applying the partial order point-wise, we obtain a complete lattice characterization of the contract domain.

$$\mathfrak{c}_1 \sqsubseteq \mathfrak{c}_2 :\Longleftrightarrow \forall \mathcal{H} \in \mathbf{HC}. \mathcal{H} \notin \mathrm{pre}(\mathfrak{c}_1) \vee (\mathcal{H} \in (\mathrm{pre}(\mathfrak{c}_1) \cap \mathrm{pre}(\mathfrak{c}_2)) \wedge \mathfrak{c}_1(\mathcal{H}) \sqsubseteq_P \mathfrak{c}_2(\mathcal{H}))$$

The bottom element of this lattice is the empty contract, i.e., the unique contract $\mathfrak{c}_\bot$ with $\mathrm{pre}(\mathfrak{c}_\bot) = \emptyset$. The top element is the (infinite) contract $\mathfrak{c}_\top$ with $\forall \mathcal{H} \in \mathbf{HC}. \mathfrak{c}_\top(\mathcal{H}) = \mathbf{err}$. We can easily see that the least upper bound operator for this domain is given by

$$(\mathfrak{c}_1 \sqcup \mathfrak{c}_2)(\mathcal{H}) = \begin{cases} \mathfrak{c}_1(\mathcal{H}) \sqcup_P \mathfrak{c}_2(\mathcal{H}), & \text{if } \mathcal{H} \in \mathrm{pre}(\mathfrak{c}_1) \cap \mathrm{pre}(\mathfrak{c}_2), \\ \mathfrak{c}_1(\mathcal{H}), & \text{if } \mathcal{H} \in \mathrm{pre}(\mathfrak{c}_1) \\ \mathfrak{c}_2(\mathcal{H}), & \text{if } \mathcal{H} \in \mathrm{pre}(\mathfrak{c}_2) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Again, this is in accordance with the intuition behind may analyses, since contracts with a larger domain are regarded as more defined.

There is one problem with this domain, however: It is infinite, because $\mathbf{HC}_{\mathrm{T},\Sigma}$ is already infinite. At first sight, it may therefore seem as though we cannot guarantee that we can compute a fixed point in finite time if we base our IPA on the contract domain. Fortunately, the abstract semantics keeps the hypergraphs fully abstract. As observed on page 48, there are only finitely many fully abstract hypergraphs for a given program if we can bound the size and number of concrete parts of the graph as well as the number of cutpoints (and hence the number of external nodes) If these requirements hold, only finitely many contracts can be generated by the analysis.

Definition 5.14 — **Abstract contract.** A procedure contract is called **abstract** w.r.t. a BCHAG $\mathfrak{G}$ if both its precondition and its postconditions are fully abstract w.r.t. $\mathfrak{G}$.

Observation 5.15 Given a program P with a bounded number of cutpoints and a BCHAG $\mathfrak{G}$ that can always abstract all but a bounded number of nodes in the program's heap, the set of abstract contracts for P modulo isomorphism is finite; in particular, it satisfies the ascending chain condition (cf. Def. A.6).

This is a highly useful property: If the abstract contract domain is finite, we can effectively compute the IMFP solution of Section 5.2 according to Theorem 5.11 on page 87.

R The proviso "modulo isomorphism" is important. In our notion of hypergraphs, nodes and edges are just arbitrary objects in a universe. Hence there are infinitely many hypergraphs in each isomorphism class. Note, however, that from a practical point of view, we do not want to distinguish between isomorphic graphs, because they represent the exact same heap! To simplify the presentation, I ignore the problem of generating only one hypergraph per isomorphism class for the time being and address it separately in Section 5.3.4.

5.3.2 An Interprocedural Data-Flow System for Contracts

The IPA of Section 5.2 was defined on interprocedural data-flow systems over some lattice D (cf. Def. 5.9 on page 85). We now develop such a system for generating procedure contracts by using the contract lattice $\mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}$ as domain.

What is the goal of our analysis? At the end of the analysis, the full procedure contracts should be available at the respective exit nodes. That is, the analysis information at each exit node should be a contract that maps each possible precondition of the procedure to the possible postconditions after the execution. For efficiency reasons, only preconditions that can actually occur throughout the main program's execution should be taken into account. Each contract that maps a heap $\mathcal{H}$ to a set of heaps proves the absence of memory errors for the precondition $\mathcal{H}$. Conversely, a contract that maps $\mathcal{H}$ to an error signifies that an error may occur during the execution of the procedure when the heap at the start of the procedure was $\mathcal{H}$.[1]

We formalize this by defining the Φ, Ξ, Ψ, and ι components of the DFS. An informal discussion will reveal how these components should be defined.

First, we need to assign an identity contract $\mathcal{H} \mapsto \{\mathcal{H}\}$ to each entry node of a procedure: At the entry node, no computation has taken place, and hence the identity contract correctly relates the precondition with the (empty) computation up until the entry node. This identity contract will, however, only be defined on those graphs $\mathcal{H}$ that occur throughout the analysis. We only explicitly define this identity contract for the main procedure (in form of the extremal value ι), the other identity contracts will be generated on demand by the call transfer functions when the procedures are actually invoked.

The transfer functions describe how the contracts evolve away from the initial identity contracts. To this end, each intraprocedural transfer function $\varphi \in \Phi$ updates the postcondition of the predecessor contracts by applying the semantics from Chapter 3 to each hypergraph in the postcondition (or propagating the error value, if applicable). The preconditions are not modified, because they refer to the precondition of the procedure execution as a whole.

Each call transfer function $\xi \in \Xi$ generates a precondition for the called procedure based on the call inference rule and then adds a corresponding identity contract to the analysis information of the called procedure's entry node. This is how the on-demand computation of the identity contracts is realized: Whenever an abstract heap is generated by the actual application of a call rule, it is added to the identity contract at the called procedure's entry node. This change in the contract at the entry node is propagated to the other nodes of the procedure through the fixed-point iteration.

Each return transfer function $\psi \in \Psi$ combines the caller and the callee contract as follows. For each postcondition of the caller contract, we pick the isomorphic precondition of the callee contract. This must exist, because it was generated in the call transfer function. To simulate procedure return, we construct two element stacks out of the selected caller postcondition and each callee postcondition for the matching precondition. We then simply apply the return inference rule.

This informal discussion shows that we can directly use the inference rules to define the transfer functions of our contract data-flow system. Let us briefly formalize this process. We start with the extremal value $\iota : \mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}$. ι is assigned to the entry node of the main procedure. As such, it should be the identity contract that maps each of the possible preconditions of the main procedure to itself. I.e., given a set of global preconditions $\{\mathcal{H}_1, \ldots, \mathcal{H}_k\}$, we define

$$\iota(\mathcal{G}) := \begin{cases} \{\mathcal{G}\}, & \text{if } \exists i. \mathcal{G} \cong \mathcal{H}_i \\ \bot, & \text{otherwise} \end{cases}$$

[1]As discussed before, this error might, however, be a false positive.

The global preconditions must be specified by the user if the analysis should begin with a non-empty initial heap.[1] Usually, the global preconditions will be a fully abstract representation of all program inputs for which we want to analyze the program. A list reversal program will, for example, have a fully abstract list as sole global precondition. Note that the global precondition is the only pre- or postcondition that is user-defined. All other contracts are generated automatically by solving the data-flow system.

As we just saw, the transfer functions can be adapted directly from the semantics in Section 3.2: Given a node n for command c, the transfer function for n receives one or two contracts as inputs and applies the inference rule for c to the contracts' postconditions to compute the updated contracts. Formally, we have to define

- $\Phi = \{\varphi_n : (\mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}) \to (\mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}) \mid n \in N^* \setminus (N_C \cup N_R)\}$
- $\Xi = \{\xi_n : (\mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}) \to (\mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}) \mid n \in N_C\}$
- $\Psi = \{\psi_n : (\mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}) \times (\mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}) \to (\mathbf{HC} \dashrightarrow 2^{\mathbf{HC}} \cup \{\mathbf{err}\}) \mid n \in N_R\}$

To be able to make sense of their definition, you might want to have another look at their use in the definition of $local$ on page 85. First, let n be a procedure-local node representing command c.

$$\varphi_n(\mathfrak{c})(\mathcal{H}) := \{\mathcal{G}' \mid \exists c' \in \mathbf{Cmd} \exists \mathcal{G} \in \mathfrak{c}(\mathcal{H}).(c,\mathcal{G}) \xrightarrow{A} (c',\mathcal{G}')\}$$

We consider each of the postconditions $\mathcal{G}$ of $\mathcal{H}$ and collect the results of applying the intraprocedural abstract semantics for the node's command c to the postconditions.

Second, for entry and exit nodes, we set $\varphi_n(\mathfrak{c})(\mathcal{H}) := \mathfrak{c}(\mathcal{H})$, because they do not have any effect of the contracts.

Third, for filter nodes $filter_b^i$, we define

$$\varphi_n(\mathfrak{c})(\mathcal{H}) := \{\mathcal{G} \in \mathfrak{c}(\mathcal{H}) \mid \wedge condEval(\mathcal{G}, b) = true\}$$

That is, we only keep those graphs in the postcondition for which the filter condition b evaluates to true. (See page 35 for the definition of $condEval$.)

Fourth, let n be a call node for a call statement c in program P.

$$\xi_n(\mathfrak{c})(\mathcal{H}) := \begin{cases} \{\mathcal{H}\} & \text{if } \exists c' \in \mathbf{Cmd} \exists \mathcal{J} \in \mathbf{HC} \exists \mathcal{G} \in \mathfrak{c}(\mathcal{J}). \\ & P \vdash \langle (c,\mathcal{G})\rangle \overset{A}{\Longrightarrow} \langle (c',\mathcal{H}), (c,\mathcal{G})\rangle \\ \bot, & \text{otherwise} \end{cases}$$

Recall that ξ_n generates a new initial contract for the called procedure, i.e., an identity contract. This contract has to be defined precisely on those heaps that occur when the procedure is called on any of the caller contract's postconditions. These heaps are extracted from the results of applying the abstract semantics to each of the postconditions $\mathcal{G}$.

Fifth, let n be a return node for a call statement **call** $p(x_1, \ldots, x_n)$ in program P. In this case, the new contract is computed based on two previous contracts, the caller contract, $\mathfrak{c}_c$,

[1]Recall that we allow parameters for the `main` procedure. This makes sense, because we usually want to apply our analysis to algorithms that exhibit some input–output behavior. Since we do not have any form of I/O in PL, we instead define initial heaps and pass them as arguments to `main`.

and the callee contract at procedure return, $\mathfrak{c}_r$.

$$\begin{aligned}\psi_n(\mathfrak{c}_c,\mathfrak{c}_r)(\mathcal{H}) := \{\mathcal{G}' \mid\ & \exists c' \in \mathbf{Cmd} \exists \mathcal{J} \in \mathfrak{c}_c(\mathcal{H}) \\ & \exists \mathcal{G} \in \mathfrak{c}_r(\mathit{reachable}(\mathcal{J},\{x_1,\dots,x_n\})). \\ & P \vdash \langle(\downarrow,\mathcal{G}),(\mathbf{call}\ p(x_1,\dots,x_n);c',\mathcal{J})\rangle \overset{A}{\Longrightarrow} \langle(c',\mathcal{G}')\rangle\}\end{aligned}$$

This definition is a little more involved. The problem is that we need to extract the right precondition–postcondition pair from the return contract $\mathfrak{c}_r$: The precondition must match the reachable fragment of the caller graph $\mathcal{J}$ w.r.t. the call parameters. The postconditions, here assigned to $\mathcal{G}$, are the possible results of executing the called procedure on that reachable fragment of the caller graph. Having identified the right pairs of heaps in this way, ψ just applies the return rule to the pairs and gathers the results.

By combining these transfer functions, we obtain a data-flow system that computes contracts by closely mirroring the abstract semantics $\overset{A}{\Longrightarrow}$. In this way, a sequence of iterations of the fixed-point analysis corresponds to a sequence of rule applications to the contracts' postconditions—the postconditions approximate the computation of $\overset{A*}{\Longrightarrow}$.

Distributivity

As noted in Section 5.2, the MOP and MFP solutions coincide if and only if all transfer functions are distributive. Let us assure ourselves that this is true for the transfer functions that we just defined. Conveniently, we obtain this result nearly for free, because our contracts are functions. I will show distributivity only for Φ; the other cases work analogously.

To show distributivity, we need to show that (cf. Def. A.8) for each of the transformers $\varphi \in \Phi$, $\varphi(\mathfrak{c}_1 \sqcup \mathfrak{c}_2) = \varphi(\mathfrak{c}_1) \sqcup \varphi(\mathfrak{c}_2)$ holds. Equivalently, we can show that for all $\mathcal{H}$, $\varphi(\mathfrak{c}_1 \sqcup \mathfrak{c}_2)(\mathcal{H}) = \varphi(\mathfrak{c}_1)(\mathcal{H}) \sqcup_p \varphi(\mathfrak{c}_2)(\mathcal{H})$, because $\sqcup$ was defined as the point-wise application of $\sqcup_p$ to the postconditions. More specifically, recall that

$$(\mathfrak{c}_1 \sqcup \mathfrak{c}_2)(\mathcal{H}) = \begin{cases} \mathfrak{c}_1(\mathcal{H}) \sqcup_P \mathfrak{c}_2(\mathcal{H}), & \text{if } \mathcal{H} \in \mathrm{pre}(\mathfrak{c}_1) \cap \mathrm{pre}(\mathfrak{c}_2), \\ \mathfrak{c}_1(\mathcal{H}), & \text{if } \mathcal{H} \in \mathrm{pre}(\mathfrak{c}_1) \\ \mathfrak{c}_2(\mathcal{H}), & \text{if } \mathcal{H} \in \mathrm{pre}(\mathfrak{c}_2) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

The only interesting case is the first one. Let therefore $\mathcal{H} \in \mathrm{pre}(\mathfrak{c}_1) \cap \mathrm{pre}(\mathfrak{c}_2)$. First, assume that $\mathfrak{c}_1(\mathcal{H}) \neq \mathbf{err} \neq \mathfrak{c}_2(\mathcal{H})$. Then the following holds.

$$\begin{aligned} & \varphi(\mathfrak{c}_1 \sqcup \mathfrak{c}_2)(\mathcal{H}) \\ = \ & \{\mathcal{G}' \mid \exists c' \in \mathbf{Cmd} \exists \mathcal{G} \in (\mathfrak{c}_1 \sqcup \mathfrak{c}_2)(\mathcal{H}).(c,\mathcal{G}) \overset{A}{\to} (c',\mathcal{G}')\} \\ = \ & \{\mathcal{G}' \mid \exists c' \in \mathbf{Cmd} \exists \mathcal{G} \in (\mathfrak{c}_1(\mathcal{H}) \sqcup_P \mathfrak{c}_2(\mathcal{H})).(c,\mathcal{G}) \overset{A}{\to} (c',\mathcal{G}')\} \\ = \ & \{\mathcal{G}' \mid \exists c' \in \mathbf{Cmd} \exists \mathcal{G} \in (\mathfrak{c}_1(\mathcal{H}) \cup \mathfrak{c}_2(\mathcal{H})).(c,\mathcal{G}) \overset{A}{\to} (c',\mathcal{G}')\} \\ = \ & \{\mathcal{G}' \mid \exists c' \in \mathbf{Cmd} \exists \mathcal{G} \in (\mathfrak{c}_1(\mathcal{H})).(c,\mathcal{G}) \overset{A}{\to} (c',\mathcal{G}')\} \cup \\ & \{\mathcal{G}' \mid \exists c' \in \mathbf{Cmd} \exists \mathcal{G} \in (\mathfrak{c}_2(\mathcal{H})).(c,\mathcal{G}) \overset{A}{\to} (c',\mathcal{G}')\} \\ = \ & \varphi(\mathfrak{c}_1)(\mathcal{H}) \cup \varphi(\mathfrak{c}_2)(\mathcal{H}) \\ = \ & \varphi(\mathfrak{c}_1)(\mathcal{H}) \sqcup_p \varphi(\mathfrak{c}_2)(\mathcal{H}) \end{aligned}$$

Conversely, if $\mathfrak{c}_1(\mathcal{H}) = \mathbf{err}$ (or, equivalently, $\mathfrak{c}_2(\mathcal{H}) = \mathbf{err}$), then $\varphi(\mathfrak{c}_1 \sqcup \mathfrak{c}_2)(\mathcal{H}) = \mathbf{err}$ and also $\varphi(\mathfrak{c}_1)(\mathcal{H}) \sqcup_p \varphi(\mathfrak{c}_2)(\mathcal{H}) = \mathbf{err} \sqcup_p \varphi(\mathfrak{c}_2)(\mathcal{H}) = \mathbf{err}$.

5.3.3 Extension to Concurrent Programs

As you may well have realized already, the techniques in the previous section are not restricted to sequential programs. I decided to develop the DFS for sequential programs only to make it simpler, but we can easily adapt it to concurrent programs.

R We exploit here that the concrete semantics of Chapter 4 are deterministic. This made it possible to treat fork and join essentially like call and return, and thus only model a single call stack, rather than one call stack per active thread. This in turn allows us to reuse the framework for (sequential) interprocedural analysis, in which we only have one stack of analysis information, for contract generation in the concurrent setting.

As a first step, we extend the contract domain to

$$\mathbf{HC} \times \mathbf{Perms} \times \mathbf{Evo} \dashrightarrow 2^{\mathbf{HC}\times\mathbf{Perms}\times\mathbf{Evo}} \cup \{\mathbf{err}\}$$

Both preconditions and postconditions now include permissions and shape evolution information. ι is modified accordingly: Given a global precondition $\{\mathcal{H}_1, \ldots, \mathcal{H}_k\}$, the initial contract becomes

$$\iota((\mathcal{G}, q_0, \eta_0)) := \begin{cases} \{(\mathcal{G},q_0,\eta_0)\}, & \text{if } \exists i.\mathcal{G} \cong \mathcal{H}_i \\ \bot, & \text{otherwise} \end{cases}$$
$$\textbf{where } \eta_0 = \lambda e.\{e\}$$
$$q_0 = (\lambda e.\mathbf{no}, \lambda e.\mathbf{no}, \lambda t.\bot)$$

The transfer functions now apply the inference rules from $\xRightarrow{\eta}$ rather than $\xRightarrow{A}$. For example, for a node n that represents the intraprocedural command c,

$$\varphi_n(\mathfrak{c})(\mathcal{H},q,\eta) := \{(\mathcal{G}',q',\eta') \mid \begin{array}{l} \exists c' \in \mathbf{Cmd} \exists \mathcal{G} \in \mathfrak{c}(\mathcal{H},q,\eta). \\ \langle (c, \mathcal{G}, q, \eta)\rangle \xRightarrow{\eta} (c',\mathcal{G}',q',\eta')\} \end{array}$$

This is less readable, but at its core, not more complex than the sequential case: It still boils down to rule application to the contract's postconditions. We omit the adaptation of the remaining transfer functions to the parallel setting—it could easily be achieved in the same way.

5.3.4 Handling Isomorphic Graphs

As pointed out in the remark on page 90, it is important that we do not distinguish between isomorphic graphs: If we have already computed the contract for precondition $\mathcal{H}$, and $\mathcal{H} \cong \mathcal{G}$, we do not need to compute a contract for $\mathcal{G}$, because $\mathcal{H}$ and $\mathcal{G}$ represent the same heap. In such cases, we should just use the contract for $\mathcal{H}$ instead. It is therefore desirable to define contracts on isomorphism classes (w.r.t. Def. 2.23 on page 22) or on the canonical representatives of isomorphism classes rather than on the full set $\mathbf{HC}$.

The result is obviously still a complete lattice, because we just exchanged the underlying set. The transfer functions become a little more complicated, however, since we cannot just apply inference rules any longer, but must perform additional isomorphism checks. In the return transfer functions, for example, we have to replace

$$\exists \mathcal{G} \in \mathfrak{c}_r(\mathit{reachable}(\mathcal{J},\{x_1, \ldots, x_n\}))$$

with

$$\exists \mathcal{J}'.reachable(\mathcal{J},\{x_1,\ldots,x_n\}) \cong \mathcal{J}' \wedge \exists \mathcal{G} \in \mathfrak{c}_r(\mathcal{J}')$$

I felt that constantly worrying about isomorphisms throughout the entire section would be too much of a distraction from the key ideas, but in an implementation, we definitely must not distinguish between isomorphic graphs.

You may, however, object that it is prohibitively expensive from a computational point of view to have to perform that many isomorphism checks. Surprisingly, it is not: We can compute canonical representatives of isomorphism classes very cheaply in our context, so the isomorphism check can be performed efficiently by computing said representatives and comparing them for equality. I shall further discuss this, as well as other algorithmic issues, next.

5.4 Algorithmic Complexity

Up until now I have completely neglected the question of efficiency, which is, of course, one of central factors for the applicability of the analysis to real programs.[1]

I will start with a rough complexity of the analysis of sequential programs and then discuss the overhead incurred by including permission information.

There are computations at several levels of abstraction that we have to analyze: The computation of each transfer function consists in applying on-demand concretization, a rule from the concrete semantics, and subsequent abstraction to each graph in the postcondition. This cost has to be multiplied by the number of postconditions, i.e., grows linearly with the size of the contract. As discussed in the previous section, we also need to keep only one representative of each isomorphism class, so we need to analyze the cost of computing hypergraph isomorphisms in our setting. Adding all these costs gives us the cost for a single step in the fixed point iteration. At a higher level, we are also interested in the number of steps until termination of the fixed-point iteration.

There are also more low-level aspects to consider in a concrete implementation, such as the data structures used for representing the hypergraphs, but these are not the focus of this section. I will therefore not discuss the cost of individual operations on nodes or hyperedges; the purpose of this section is not to provide sharp complexity bounds, but rather to convey a feeling for the complexity of the proposed methods.

Canonical Representatives

Finding canonical representatives will be useful at multiple points, so I would like to discuss this issue first. The basic idea is that, because we assume garbage collection, all nodes in a heap are reachable from at least one variable edge. We already exploited this in Def. 3.6 on page 42, where we defined a total order on the nodes $<_V$. Using this order, it is straightforward to define a canonical representative for a given heap $\mathcal{H} = (V, E, att, lab, ext, typ, isnull)$. Let $pos(v) \in \{1, \ldots, |V|\}$ denote the position of node $v \in V$ in the order $<_V$. We then define the canonical representative $\mathcal{H}'$ as follows.

- $V' := \{1, \ldots, |V|\}$
- $E := \{(\sigma,\langle pos(att(e)(1)), \ldots ,pos(att(e)(k))\rangle) \mid e \in E \wedge lab(e) = \sigma \wedge \mathrm{rk}(\sigma) = k\}$

[1]The other major one being the precision of the abstraction.

- $lab'((\sigma,\langle n_1,\ldots,n_k\rangle)) := \sigma$
- $att'((\sigma,\langle n_1,\ldots,n_k\rangle)) := \langle n_1,\ldots,n_k\rangle$
- $ext', typ', isnull'$ are obtained by renaming all nodes using pos

We always use an initial fragment of the positive integers as set of nodes and to encode each edge's label and attachment directly into the edge object. Clearly, this representation is the same for a pair of graphs $\mathcal{G}$ and $\mathcal{H}$ if and only if $\mathcal{G} \cong \mathcal{H}$.

Once we have the order (and hence the pos function), the representative can be computed in $\mathcal{O}(|E| + |V|)$. The order itself can, for example, be computed by applying Dijkstra's algorithm once for each of the (constantly many) variable edges, i.e., in time $\mathcal{O}(|E| + |V|\log(V))$. We can obviously compare canonical representative against each other in $\mathcal{O}(|E| + |V|)$. Perhaps surprisingly, we can therefore check whether a pair of heaps is isomorphic in time $\mathcal{O}(|E| + |V|\log(V))$, by first computing each canonical representative and then comparing them for equality.

The cost of concrete symbolic execution

At the heart of all transfer functions lies the application of one inference rule of the concrete semantics to a concrete subgraph of the heap. First, all local operations (assignments, memory allocation, variable declaration, filters) are cheap: We only need to manipulate and create a constant number of nodes and edges.

The main complexity of call statements lies in the computation of reachable fragments. To this end, we can use standard algorithms to identify all nodes that are reachable from any of the (constantly many) procedure parameters. This can be done in time linear in the number of hyperedges in the reachable fragment, since each hyperedge will be processed only a constant number of times.

Executing procedure return is more interesting. Recall that we must replace a part of the caller heap with the exit heap of the callee. To this end, we must reattach the edges on the boundary between the caller and the callee heap. This takes time linear in the size of the boundary. Embedding the callee heap is rather cheap if we make the assumption that the sets of nodes and edges of the unmodified part of the caller graph and the callee graph are disjoint. If we only store canonical representatives, this will obviously not be the case, but can be achieved by shifting the nodes in one of the graphs that need to be merged—an operation that is linear in the size of the graph.[1] Having embedded the returned fragment, we then perform a reachable fragment computation (starting from the variable edges) to get rid of the (now unreachable) fragment of the caller graph that we have just replaced. The total number of operations on edges and nodes is thus linear in the size of the involved hypergraphs.

The cost of concretization and abstraction

Recall that we have to perform on-demand concretization to be able to apply the concrete semantics, and perform full abstraction after applying the concrete semantics.

Let n be the number of rules in G, m be the maximum size of a right-hand side, ℓ be the size of $\mathcal{H}$. Concretization is only necessary when executing assignments or evaluating conditions. In the former case, only at most two violation points are relevant. In the second

[1] When actually implementing this, other approaches are possible. In my implementation I keep unique node identifiers in addition to the identifiers in the canonical representative and merge the two graphs based on the unique identifiers, not on the identifiers in the canonical representative.

case, the number of relevant violation points is bounded by the number of dereferenced pointers in the condition. Let p denote the number of relevant violation points. In the worst case we have to perform n^p concretizations. Note, however, that both n and p will usually be small in practice.

For the subsequent abstraction, we have to find subgraphs that are isomorphic to the right-hand side of a production rule. In the worst case, this is exponential in m. (Subgraph isomorphism is NP complete.) In practice, m is usually very small—certainly in the single digits—and $\mathcal{H}$ has enough structure (thanks to the types and labels) to further reduce the complexity, since even a brute force back-tracking algorithm for finding subgraphs can quickly rule out most edge combinations.

Costs at the contract level

The main additional cost at the contract level is caused by the necessity to identify isomorphic hypergraphs: We have to merge precondition–postcondition pairs that have isomorphic preconditions and we also have to remove isomorphic graphs from each postcondition. Pairwise isomorphism is checked by computing canonical representatives. Canonical representatives can be both cached and hashed so that the total cost of removing all duplicates (modulo isomorphism) from a set of hypergraphs will be not much higher than the cost of computing each canonical representative once. (Comparison of hash values is possible in $\mathcal{O}(()1)$ and, given a good hash function, comparing hash values of canonical representatives will usually suffice.)

Solving the equation system

A standard worklist algorithm can be used to optimize the fixed-point iteration [NNH99]. Such an algorithm computes one transfer function at a time, rather than updating the values for all equations simultaneously. Changes are propagated by adding successor nodes to a list of nodes for which the fixed-point iteration has not yet converged—the worklist. This usually leads to much faster termination than the naive solution of the equation system.

In the worst case we do, however, still have to perform $n \cdot m$ iterations, n being the number of nodes and m being the height of the contract lattice. In general, $m = \infty$—we cannot bound the size of the heaps that occur throughout the analysis. But if we assume that we can always abstract all but a bounded number of nodes ℓ, because we have a suitable BCHAG for the program and only boundedly many permission alternations occur (in the parallel case), m still is a huge number:

- The number of heaps modulo isomorphism is exponential in the number of variables in the program, in the number of cutpoints, the number of nonterminals in the grammar, and in ℓ.
- The number of postconditions is exponential in the number of heaps modulo isomorphism, because postconditions are arbitrary subsets of the set of possible heaps.
- The number of contracts is equal to the product of the number of heaps modulo isomorphism and the number of postconditions, as there may be one postcondition per heap.

In total, the height of the domain is thus triply exponential in the number of variables, cutpoints, nonterminals, and ℓ. While these worst-case guarantees are truly terrible, we have reason to hope that we will never come near them in practice: Each procedure will usually only be called for a small number of different preconditions, each postcondition for a given precondition will only consist of a few hypergraphs, and only a subset of variables and nonterminals is relevant for each procedure. Additionally there are often implicit invariants that greatly reduce the number of possible heaps that may occur. (For example, a tail pointer always points to a node that is after the node pointed to by the head pointer.)

Still, only experiments with a mature implementation can show whether the contracts are small enough in practice to make the proposed analysis viable.

5.5 Comparison with Separation Logic

Before concluding this thesis, I would like to briefly address one question that I have been asked several times: "Why not separation logic?" After all, I am trying to solve exactly the same problem and there is already good tool support for several separation logics.[1] My perspective is as follows.

- Separation logic formulas are often difficult to read and comprehend. Graphical models are much more intuitive. This is aided by our aggressive approach to abstraction, which leads to small and hence easily comprehensible abstract graphs. As such, procedure contracts based on (hyper)graphs rather than separation logic formulas can be much more useful, especially for manual inspection and as basis for debugging.
- Entailment is undecidable for full separation logic. To enable tool support, various decidable fragments have been proposed, most prominently the restriction to linked lists [BCO05a] that has served as the basis of many tool implementations. Each time we want to incorporate additional data structures we must, however, develop new decision procedures (assuming the extended logic is still decidable).
  This is completely different for our approach: We just have to design a suitable heap abstraction grammar for our data structure. The analysis itself remains unchanged and decidable. This enables us to easily model custom object graphs within our formalism, which is not easily (if at all) possible in the framework of separation logic. To my knowledge, completely automatic procedure contract generation in such a general setting has not been proposed before—neither in the separation logic community nor elsewhere.
- It should also be noted that separation logic and our hypergraph-based approach are quite closely related. This correspondence between separation logic and the hypergraphs approach was first noted in [Dod08] and subsequently extended in [JGN14]. The details are out of the scope of this work; the key idea is that hyperedge replacement grammars mirror recursive separation logic predicates.
  This correspondence has potentially far-reaching implications: We can translate back and forth and use results from both worlds. In this way we can, for example, determine more easily whether entailment of a given separation logic fragment is decidable [Mat14].

[1]Countless variants have been proposed, hence the plural; see [Par10].

CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

Heap analysis based on hypergraphs and grammar-based abstraction has now been studied for several years [Hei+15; JGN14; Jan+11; HNR10; Rie09; Dod09]. In 2014, Jansen and Noll proposed to use abstract hypergraphs as building blocks for procedure contracts, and to automatically generate such contracts by means of an interprocedural data-flow analysis [JN14]. This work, however, exclusively focused on the analysis of sequential programs. It also lacked a thorough formalization of the underlying program semantics.

I set out to address both of these issues; as such, the contributions of this thesis can be viewed from two orthogonal perspectives: First, I integrated the hypergraph-based heap analysis of interprocedural programs into the framework of abstract interpretation [CC77], thereby developing a more formal justification for the analysis proposed by Jansen and Noll [JGN14]. This made it necessary to define an operational hypergraph-based semantics, which was also not done with a similar degree of rigor in previous work such as [Hei+15; JN14; HBJ12].

Second, I endeavored to extend the interprocedural analysis to incorporate (limited) support for the analysis of concurrent programs. To this end, I developed a permission-based operational semantics for a language with fork and join, and briefly argued that this semantics enables the analysis of concurrent programs within the same framework for (sequential) interprocedural analysis [KS92]. In the process, I developed a novel approach to permission accounting, in which I modeled the current thread's permissions, the globally lost permissions, and the permissions of unjoined children separately, rather than keeping track of fractional permissions [Boy03; Bor+05], counting permissions [Bor+05], or dynamic thread tokens [Heu+11]. This resulted in very simple permission arithmetic and made it possible to define a data–race-free semantics for our limited fork–join model.

Both the analyses presented here and the original analysis of [JN14] are appealing for a number of reasons. The resulting analyses are modular—each procedure is analyzed independent of its calling context, each thread independent of its forking context. The approach is also highly automatic: We need to provide only a graph grammar and an initial precondition for the main thread; the contract generation itself does not require any human intervention. In addition, we are able to deal with arbitrary data structures of bounded tree width out of the box—we just need to provide a suitable grammar. This is, for example, not true for separation logic, where entailment quickly becomes undecidable. Many tools for separation logic are therefore restricted to fixed decidable fragments, such as the restriction to singly-linked lists presented by Berdine et al. [BCO05a].

We saw, however, that the worst-case complexity of our method may be an impediment to practical applicability. In Appendix C, I shall briefly introduce my prototypical implementation of the analysis. Developing case studies for this implementation should reveal whether the worst-case complexity is indeed a problem for actual programs.

We were also unable to guarantee termination in the parallel setting, unless each thread in the analyzed program assigned different permissions to only a bounded number of sub-heaps. We also noted that full abstraction using hyperedge abstraction grammars yields a very coarse over-approximation of the state space, which may result in a large number of false positives in practice.

6.2 Future Work

I shall conclude this thesis by proposing a few directions for future work, which may help overcome the mentioned shortcomings of my method.

Implementation

This thesis approached the exploration of hypergraph-based program semantics and dataflow analysis mainly from a theoretical perspective. In my opinion, the first focus on future work should therefore be on tool development; to this end, my prototype, which I briefly introduce in Appendix C, can serve as a possible starting point.

Improved abstraction and automation

Performing full HRG-based abstraction leads to a very coarse abstraction. Beside that, HRGs as defined here are also not powerful enough to capture non-uniform permission distribution patterns. On top of that, HRGs have to be developed manually. These limitations motivate various lines of future work:

- We can change the semantics to always leave a constant neighborhood of each variable edge concrete to improve the precision of the analysis; this approach was successfully employed in the Juggrnaut tool [HBJ12].
- Conversely, at the expense of precision, we can allow the application of HRG rules even for non-uniform distributions. The only sound way to do this is to take the maximum over all permissions that occur in the abstracted subgraph. We should perform experiments to find out whether we can still prove useful properties of parallel programs given this additional over-approximation.
- The abstraction mechanism itself can be refined or replaced without ramifications for the rest of the semantics or the analysis framework, as long as the modified mechanism still allows a safe approximation of the concrete semantics.
  One could, for example, experiment with parameterized HRGs. One way to do this is to add a fixed number of permission-valued parameters to each nonterminal in the grammar to be able to abstract data structures with non-uniform, but regular permission distribution. The nonterminal $L(\mathbf{rd},\mathbf{wt})$ could, for instance, represent a list for which the thread needs a read permission on all elements with odd index and a write permission for all elements with even index—which is precisely the kind of permission distribution that would cause the divergence of our current analysis.

- To achieve further automation as well as to find the right degree of abstraction, it would be interesting to explore mechanisms for automatically deriving HRGs, possibly coupled with counterexample-guided abstraction refinement (CEGAR) techniques [Cla+00]. To this end, a possible starting point is [Wei12], where automatic inference of heap abstraction grammars is explored.

Support for synchronization mechanisms

The fork–join model presented in this thesis is rather limited. In particular, it completely lacks bilateral synchronization mechanisms such as locks or monitors. Our current formalization of the semantics cannot easily be adapted to incorporate synchronization: We can no longer pretend that a fork is like a call, because we can only execute the forked thread up to its first synchronization point.

This is not surprising: In the presence of synchronization, even data–race-free programs can exhibit nondeterministic behavior, but they do so in a safe way by encapsulating statements that can cause interference within critical sections. It is therefore impossible to define a fully deterministic semantics for programs with synchronization. In the future work we must therefore find a way to incorporate this (limited form of) nondeterminism in our semantics and analysis framework.

Model checking

Last but not least, it is possible to extract an abstract state space from the fixed point of the interprocedural analysis: Each abstract heap configuration in the analysis result corresponds to a state in the state space. This gives rise to the following idea: We can

1. Annotate the abstract heaps with properties expressed in a logical formalism
2. Extract the abstract state space as well as the property
3. Pass the state space to a model checker such as SPIN [Hol04]

This would allow us to perform verification of additional properties of sequential as well as concurrent PL programs. A similar approach was taken in the Juggrnaut tool [HBJ12].

Bibliography

[Aho+06] Alfred V Aho, Monica Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, And Tools.* Pearson Education Inc., 2006 (cit. on pp. 6, 38, 78).

[Bae05] Jos CM Baeten. “A brief history of process algebra”. In: *Theoretical Computer Science* 335.2 (2005), pp. 131–146 (cit. on p. 55).

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. “SLAyer: Memory safety for systems-level code”. In: *Computer Aided Verification.* Springer. 2011, pp. 178–183 (cit. on p. 3).

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W O’Hearn. “A decidable fragment of separation logic”. In: *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science.* Springer, 2005, pp. 97–109 (cit. on pp. 98, 100).

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W O’Hearn. “Symbolic execution with separation logic”. In: *Programming Languages and Systems.* Springer, 2005, pp. 52–68 (cit. on p. 3).

[BCO06] Josh Berdine, Cristiano Calcagno, and Peter W O’Hearn. “Smallfoot: Modular automatic assertion checking with separation logic”. In: *Formal Methods for Components and Objects.* Springer. 2006, pp. 115–137 (cit. on p. 3).

[BH14] Stefan Blom and Marieke Huisman. “The VerCors Tool for verification of concurrent programs”. In: *FM 2014: Formal Methods.* Springer, 2014, pp. 127–131 (cit. on p. 4).

[BIL03] Marius Bozga, Radu Iosif, and Yassine Laknech. “Storeless semantics and alias logic”. In: *ACM SIGPLAN Notices* 38.10 (2003), pp. 55–65 (cit. on pp. 13, 31, 77).

[BNR08] Anindya Banerjee, David A Naumann, and Stan Rosenberg. “Regional logic for local reasoning about global invariants”. In: *ECOOP 2008–Object-Oriented Programming.* Springer, 2008, pp. 387–411 (cit. on p. 2).

[Bor+05] Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. “Permission accounting in separation logic”. In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 259–270 (cit. on pp. 3, 58, 99).

[Boy03] John Boyland. “Checking interference with fractional permissions”. In: *Static Analysis.* Springer, 2003, pp. 55–72 (cit. on pp. 3, 56, 99).

[CC77] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977, pp. 238–252 (cit. on pp. 1, 45, 49, 99, 112).

[Cla+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-guided abstraction refinement”. In: *Computer Aided Verification*. Springer. 2000, pp. 154–169 (cit. on pp. 57, 101).

[Con63] Melvin E Conway. “A multiprocessor system design”. In: *Proceedings of the November 12-14, 1963, fall joint computer conference*. ACM. 1963, pp. 139–146 (cit. on p. 2).

[DGT93] John Darlington, M Ghanem, and Hing Wing To. “Structured parallel programming”. In: *Programming Models for Massively Parallel Computers*. IEEE. 1993, pp. 160–169 (cit. on p. 2).

[Dij68] Edsger W Dijkstra. “Co-operating sequential processes”. In: *Programming Languages* (1968) (cit. on p. 2).

[DKH97] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. “Hyperedge Replacement Graph Grammars”. In: Grzegorz Rozenberg. *Handbook of Graph Grammars*. Ed. by Grzegorz Rozenberg. Vol. 1. World scientific Singapore, 1997, pp. 95–162 (cit. on pp. 14, 18, 19, 22).

[Dod08] Mike Dodds. “From separation logic to hyperedge replacement and back”. In: *Graph Transformations*. Springer, 2008, pp. 484–486 (cit. on pp. 3, 98).

[Dod09] Mike Dodds. “Graph transformation and pointer structures”. PhD thesis. University of York, 2009 (cit. on pp. 3, 12, 17, 99).

[Dow05] Allen B Downey. *The Little Book of Semaphores*. Published online. 2005 (cit. on p. 55).

[DS91] Evelyn Duesterwald and Mary Lou Soffa. “Concurrency analysis in the presence of procedures using a data-flow framework”. In: *Proceedings of the symposium on Testing, analysis, and verification*. ACM. 1991, pp. 36–48 (cit. on p. 81).

[FFQ02] Cormac Flanagan, Stephen N Freund, and Shaz Qadeer. “Thread-modular verification for shared-memory programs”. In: *Programming Languages and Systems*. Springer, 2002, pp. 262–277 (cit. on p. 60).

[FGL96] Pascal Fradet, Ronan Gaugne, and Daniel Le Métayer. “Static detection of pointer errors: an axiomatisation and a checking algorithm”. In: *Programming Languages and Systems (ESOP’96)*. Springer, 1996, pp. 125–140 (cit. on p. 1).

[Gos00] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000 (cit. on pp. 2, 8).

[Got+07] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. “Thread-modular shape analysis”. In: *ACM SIGPLAN Notices* 42.6 (2007), pp. 266–277 (cit. on pp. 57, 60).

[Hab+12] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. “Forest automata for verification of heap manipulation”. In: *Formal methods in system design* 41.1 (2012), pp. 83–106 (cit. on p. 2).

[HBJ12] Jonathan Heinen, Henrik Barthels, and Christina Jansen. “Juggrnaut–An Abstract JVM”. In: *Formal Verification of Object-Oriented Software*. Springer, 2012, pp. 142–159 (cit. on pp. 3, 99–101).

[Hei+15] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. “Verifying pointer programs using graph grammars”. In: *Science of Computer Programming* (2015) (cit. on pp. 3, 12, 17, 99).

[Heu+11] Stefan Heule, K Rustan M Leino, Peter Müller, and Alexander J Summers. “Fractional permissions without the fractions”. In: *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*. ACM. 2011, p. 1 (cit. on pp. 3, 58, 60, 99).

[HHH11] Christian Haack, Marieke Huisman, and Clément Hurlin. “Permission-based separation logic for multithreaded Java programs”. In: *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica* 15 (2011), pp. 13–23 (cit. on p. 3).

[HNR10] Jonathan Heinen, Thomas Noll, and Stefan Rieger. “Juggrnaut: Graph grammar abstraction for unbounded heap structures”. In: *Electronic Notes in Theoretical Computer Science* 266 (2010), pp. 93–107 (cit. on pp. 12, 49, 99).

[Hol04] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley, 2004 (cit. on p. 101).

[IEEE95] *Information Technology–Portable Operating System Interface*. IEEE 1003.1c-1995. Geneva, Switzerland: Institute of Electrical and Electronics Engineers, 1995 (cit. on p. 2).

[Jan+11] Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, and Thomas Noll. “A local Greibach normal form for hyperedge replacement grammars”. In: *Language and Automata Theory and Applications* (2011), pp. 323–335 (cit. on pp. 12, 14, 15, 20, 24–26, 28, 29, 99).

[JGN14] Christina Jansen, Florian Göbe, and Thomas Noll. “Generating inductive predicates for symbolic execution of pointer-manipulating programs”. In: *Graph Transformation* (2014), pp. 65–80 (cit. on pp. 14, 98, 99).

[JHC13] Cliff B Jones, Ian J Hayes, and Robert J Colvin. “Balancing expressiveness in formal approaches to concurrency”. In: *Formal Aspects of Computing* (2013), pp. 1–23 (cit. on p. 2).

[JN14] Christina Jansen and Thomas Noll. “Generating Abstract Graph-Based Procedure Summaries for Pointer Programs”. In: *Graph Transformation* (2014), pp. 49–64 (cit. on pp. 3, 4, 12, 14, 17, 19, 29, 49, 77, 99, 117).

[Jon12] Cliff B Jones. “Abstraction as a unifying link for formal approaches to concurrency”. In: *Software Engineering and Formal Methods* (2012), pp. 1–15 (cit. on pp. 2, 14).

[Jon81] Henricus BM Jonkers. *Abstract storage structures*. Mathematisch Centrum, Afdeling Informatica, 1981 (cit. on p. 31).

[JP08] Bart Jacobs and Frank Piessens. “The VeriFast program verifier”. In: *CW Reports* (2008) (cit. on p. 3).

[Kas11] Ioannis T Kassios. “The dynamic frames theory”. In: *Formal Aspects of Computing* 23.3 (2011), pp. 267–288 (cit. on p. 2).

[Kre+13] Jörg Kreiker, Thomas Reps, Noam Rinetzky, Mooly Sagiv, Reinhard Wilhelm, and Eran Yahav. “Interprocedural shape analysis for effectively cutpoint-free programs”. In: *Programming Logics*. Springer, 2013, pp. 414–445 (cit. on pp. 42, 49).

[KS92] Jens Knoop and Bernhard Steffen. “The interprocedural coincidence theorem”. In: *Compiler Construction*. Springer Berlin Heidelberg. 1992, pp. 125–140 (cit. on pp. 3, 78, 82, 83, 88, 99, 113).

[KU77] John B Kam and Jeffrey D Ullman. “Monotone data flow analysis frameworks”. In: *Acta Informatica* 7.3 (1977), pp. 305–317 (cit. on p. 115).

[LC89] Douglas L Long and Lori A Clarke. “Task interaction graphs for concurrency analysis”. In: *Proceedings of the 11th international conference on Software engineering*. ACM. 1989, pp. 44–52 (cit. on p. 81).

[Lea00] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000 (cit. on pp. 2, 55).

[LMS09] K Rustan M Leino, Peter Müller, and Jan Smans. “Verification of concurrent programs with Chalice”. In: *Foundations of Security Analysis and Design V*. Springer, 2009, pp. 195–222 (cit. on p. 3).

[Mag+08] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. “THOR: A tool for reasoning about shape and arithmetic”. In: *Computer Aided Verification*. Springer. 2008, pp. 428–432 (cit. on p. 3).

[Mar12] Simon Marlow. “Parallel and concurrent programming in Haskell”. In: *Central European Functional Programming School*. Springer, 2012, pp. 339–401 (cit. on p. 56).

[Mat14] Christoph Matheja. *Reconciling Decidability of Separation Logic Entailment and Graph Grammar Language Inclusion*. Master Thesis. RWTH Aachen University. 2014 (cit. on p. 98).

[McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft press, 2004 (cit. on p. 1).

[Mey88] Bertrand Meyer. “Eiffel: A language and environment for software engineering”. In: *Journal of Systems and Software* 8.3 (1988), pp. 199–246 (cit. on p. 89).

[NAC99] Gleb Naumovich, George S Avrunin, and Lori A Clarke. “An efficient algorithm for computing MHP information for concurrent Java programs”. In: *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’99)*. Springer. 1999, pp. 338–354 (cit. on p. 81).

[NNH99] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999 (cit. on pp. 78, 86, 97, 111–115).

[Nol15] Thomas Noll. *Lecture Notes on Static Program Analysis*. Available at `http://moves.rwth-aachen.de/teaching/ws-1415/spa/`. 2015 (cit. on pp. 52, 79, 81, 111–113).

[OG76] Susan Owicki and David Gries. “An axiomatic proof technique for parallel programs I”. In: *Acta informatica* 6.4 (1976), pp. 319–340 (cit. on p. 2).

[OHe07] Peter W O’Hearn. “Resources, concurrency, and local reasoning”. In: *Theoretical computer science* 375.1 (2007), pp. 271–307 (cit. on p. 3).

[Pac11] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011 (cit. on p. 2).

[Par10] Matthew Parkinson. “The next 700 separation logics”. In: *Verified Software: Theories, Tools, Experiments* (2010), pp. 169–182 (cit. on p. 98).

[Plo04] Gordon D Plotkin. “A structural approach to operational semantics”. In: *J. Log. Algebr. Program.* 60.61 (2004). Originally published as DAIMI FN-19, Dept. of Computer Science, Univ. of Aarhus, 1981, pp. 17–139 (cit. on pp. 31, 35).

[Plu10] Detlef Plump. “Checking graph-transformation systems for confluence”. In: *Electronic Communications of the EASST* 26 (2010) (cit. on p. 28).

[PWZ13] Ruzica Piskac, Thomas Wies, and Damien Zufferey. “Automating separation logic using SMT”. In: *Computer Aided Verification*. Springer. 2013, pp. 773–789 (cit. on p. 3).

[Rey02] John C Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74 (cit. on pp. 2, 3, 12).

[Rie09] Stefan Rieger. “Verification of Pointer Programs”. PhD thesis. RWTH Aachen University, 2009 (cit. on pp. 3, 12, 17, 99).

[Rin+05] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. “A semantics for procedure local heaps and its abstractions”. In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 296–309 (cit. on pp. 31, 42).

[Roe01] W-P de Roever. *Concurrency Verification: Introduction to Compositional and Non-compositional Methods*. Vol. 54. Cambridge University Press, 2001 (cit. on p. 2).

[Roz99] Grzegorz Rozenberg. *Handbook of graph grammars and computing by graph transformation*. Vol. 1. World scientific Singapore, 1999 (cit. on pp. 22, 23).

[Rus95] John Rushby. *Formal methods and their role in the certification of critical systems*. Springer, 1995 (cit. on pp. 1, 12, 14).

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. “Parametric shape analysis via 3-valued logic”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24.3 (2002), pp. 217–298 (cit. on pp. 1, 2, 12).

[Sta85] Eugene W Stark. “A proof technique for rely/guarantee properties”. In: *Foundations of software technology and theoretical computer science*. Springer. 1985, pp. 369–391 (cit. on p. 88).

[Wei12] Alexander Dominik Weinert. *Inferring Heap Abstraction Grammars*. Bachelor Thesis. RWTH Aachen University. 2012 (cit. on p. 101).

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993 (cit. on p. 56).

Appendices

APPENDIX A

An Introduction to Order Theory

In this appendix I briefly summarize some basic order theoretic background, which is needed to follow parts of my thesis. I adapted the definitions from [NNH99; Nol15]. We are interested in ordered sets $(D, \sqsubseteq)$ that exhibit some additional structure:

Definition A.1 — **Least upper bound.** Let $S \subseteq D$. $u \in D$ is called the least upper bound (LUB) of S if

1. For all $d \in S$, $d \sqsubseteq u$, i.e., u is an upper bound of S.
2. For all $v \in D$, if $d \sqsubseteq v$ for all $d \in S$, then also $u \sqsubseteq v$, i.e., all other upper bounds of S are greater than u.

We write $d_1 \sqcup d_2$ for the LUB of two elements and $\bigsqcup S$ for the LUB of the set S.

Definition A.2 — **Complete lattice.** $(D, \sqsubseteq)$ is called a *complete lattice* if all subsets $S \subseteq D$ have both a least upper bound and a greatest lower bound.

It is a well-known result of order theory that it is sufficient to show only one of the two conditions [NNH99, Lemma A.2].

Theorem A.3 The following statements are equivalent.

1. $(D, \sqsubseteq)$ is a complete lattice.
2. Every subset of D has a least upper bound.
3. Every subset of D has a greatest lower bound.

Sets ordered by the subset relation form a complete lattice.

Lemma A.4 — **Powerset lattices.** Let A be an arbitrary set. Then $(2^A, \subseteq)$ is a complete lattice.

Proof. Let $S \subseteq D$. For all $d \in S, d \subseteq \bigcup S$, so $\bigcup S \in 2^A$ is an upper bound of S. Let T be an upper bound of S. Then in particular, for all $d \in S$, $d \subseteq T$. Hence, $\bigcup S \subseteq T$ and $\bigcup S$ is the least upper bound of S. ■

Definition A.5 — **Chain.** Let $(D, \sqsubseteq)$ be a complete lattice. A sequence $\langle d_1, d_2, \ldots \rangle$ of elements $d_i \in D$ is called a *chain* if $d_i \sqsubseteq d_{i+1} \forall i$.

Definition A.6 — **Ascending chain condition (ACC).** Let $(D, \sqsubseteq)$ be a complete lattice. $(D, \sqsubseteq)$ satisfies the *ascending chain condition* (ACC) if all chains in D become stationary, i.e., if for all chains $\langle d_1,d_2,\ldots\rangle$ there exists an i such that for all $j \geq i$, $d_j = d_{j+1}$.

Definition A.7 — **Height of a lattice.** Let $(D, \sqsubseteq)$ be a complete lattice. If there exists a maximal sequence $\langle d_1,\ldots,d_n\rangle$ in D such that $d_i \sqsubset d_{i+1} \forall i$, $(D, \sqsubseteq)$ has *height* $n-1$. Otherwise, its height is ∞. We say that $(D, \sqsubseteq)$ has finite height if its height is not ∞.

Definition A.8 — **Monotonic and distributive functions.** Let $(L, \sqsubseteq_L)$ and $(M, \sqsubseteq_M)$ be complete lattices and $f : L \to M$. f is called *monotonic* if for all $d_1,d_2 \in L$, $d_1 \sqsubseteq_L d_2 \implies f(d_1) \sqsubseteq_M f(d_2)$. If $f(d_1 \sqcup d_2) = f(d_1) \sqcup f(d_2)$, f is called *distributive*.

All distributive functions are monotonic, but not the other way around.

Monotonicity is an interesting property, because it guarantees the existence of effectively computable fixed-points.

Theorem A.9 — **Knaster-Tarski fixed-point theorem [Nol15].** Let $(D, \sqsubseteq)$ be a complete lattice and $\varphi : D \to D$ a monotonic function on D. Then φ has a least-fixed point $\text{fix}(\varphi)$, which is given by

$$\text{fix}(\varphi) = \bigsqcup\{\varphi^k(\bot) \mid k \in \mathbb{N}\}$$

For the proof see, for example, [NNH99].

Corollary A.10 — **Effectively computable fixed-points.** If $(D, \sqsubseteq)$ has the ACC, then there exists a $k \in \mathbb{N}$ such that $\text{fix}(\varphi) = \varphi^k(\bot)$.

For the theory of abstract interpretation (cf. Sections 3.2 and 4.2) we also need to relate pairs of lattices via translation functions that are structure-preserving in a certain sense:

Definition A.11 — **(Monotone) Galois connection.** Let $(L, \sqsubseteq_L)$ and $(M, \sqsubseteq_M)$ be complete lattices. Further, let $\alpha : L \to M$ and $\gamma : M \to L$ be monotonic. (L,α,γ,M) is called a *Galois connection* iff

1. $\forall \ell \in L.\ell \sqsubseteq_L \gamma(\alpha(\ell))$
2. $\forall m \in M.\alpha(\gamma(m)) \sqsubseteq_M m$

α is called the lower adjoint of γ, γ is called the upper adjoint of α.

In the context of abstract interpretation, L is called the *concrete domain*, and M the *abstract domain*. Consequently, we refer to α as the *abstraction function* and to γ as the *concretization function*. The intuition behind Galois connections is as follows.

1. No concrete value is lost through abstraction and subsequent concretization.
2. We do not get a coarser abstraction (i.e., greater abstract value) through concretization and subsequent abstraction.

These are exactly the properties that we need for a theory of sound approximation.

For a more thorough introduction, which is also tailored towards applications in static analysis, I suggest a look at “Principles of Program Analysis” [NNH99]; the original paper on abstract interpretation [CC77] is also quite accessible and certainly worth reading.

APPENDIX B

A Primer on (Intraprocedural) Data-Flow Analysis

Chapter 5 may be hard to follow without background in data-flow analysis. If that is the case for you, you may want to read this appendix as preparation for Section 5.2. In the following, I introduce all the concepts used in Section 5.2 in the simpler intraprocedural setting. This appendix is based on [Nol15; NNH99; KS92].

Let $(D, \sqsubseteq)$ be a complete lattice. Let $\sqcup$ denote the LUB operator, $\bot$ the least element and $\top$ the top element of the lattice.

Definition B.1 — **(Distributive) data-flow system.** A *data-flow system* over D is a triple $S = (G,\Phi,\iota)$, where

- $G = (N,E,s,e)$ is a CFG (cf. Section 5.1)
- $\Phi = \{\varphi_n : D \to D \mid n \in N\}$ is a set of monotonic *transfer functions*
- $\iota : D$ is the extremal value, i.e., the analysis information for s

If all individual φ_n are distributive, we call S distributive.

A data-flow system gives rise to a set of data-flow equations. Let $n_1, \ldots, n_k$ be an (arbitrary) ordering of N. Let $\alpha_n \in D$ denote the analysis information at node n. We call a vector $(\alpha_{n_1}, \alpha_{n_k})$ a *solution* of a data-flow system if it satisfies the following equation system.

$$\alpha_{n_i} = \begin{cases} \iota, & n_i = s^* \\ \bigsqcup\{\varphi_m(\alpha_m) \mid (m,n_i) \in F^*\}, & \text{otherwise} \end{cases}$$

Additionally, Φ can (by abuse of notation) be viewed as a monotone operator

$$\Phi : D^k \to D^k, (\alpha_1, \ldots, \alpha_k) \mapsto (\hat{\alpha_1}, \ldots, \hat{\alpha_k})$$

where

$$\hat{\alpha_i} := \begin{cases} \iota, & n_i = s^* \\ \bigsqcup\{\varphi_m(\alpha_j) \mid (m_j,n_i) \in F^*\}, & \text{otherwise} \end{cases}$$

It is easy to see that $(\alpha_{n_1}, \alpha_{n_k})$ is a solution of the equation system if and only if it is a fixed point of Φ [Nol15].

We also observe that if $(D, \sqsubseteq)$ is a complete lattice, so is $(D^k, \sqsubseteq^k)$, where $\sqsubseteq^k$ is defined point-wise. In addition, if $(D, \sqsubseteq)$ is of finite height, then so is $(D^k, \sqsubseteq^k)$, and the fixed point is effectively computable (cf. Theorem A.9). This gives us a way to solve a data-flow system: Fixed-point iteration from $\bot^k$, yielding the least solution of the system $\text{fix}(S)$.

Observation B.2 For a data-flow system over a lattice of finite height, there is an $n \in \mathbb{N}$ such that $\Phi^n(\bot^k)$ is a solution. Such a data-flow system can thus be effectively solved by fixed point iteration of Φ. The solution obtained in this way is the *least fixed point* of Φ.

R For historic reasons, this solution is usually called the *maximal fixed point* (MFP) solution, although it is in fact the least fixed point of the data-flow system [NNH99]. For consistency with the literature, I will also refer to it as the MFP solution, even though we are exclusively interested in least fixed points in this thesis.

What can we say about the quality of the fixed-point solution? Let us briefly characterize the most precise solution of a data-flow system.

Ideally, a data-flow analysis would analyze all paths through the program and merge (i.e., join) the results. Let us formalize this idea. First of all, we define the sets of paths through a CFG.

Definition B.3 — **Paths.** A (finite) path through a CFG is a (finite) sequence of nodes $\langle x_1, \dots, x_j\rangle$ such that all pairs of consecutive nodes are connected by an edge, i.e., $(x_i, x_{i+1}) \in F \; \forall i \in \{1, \dots, j-1\}$.

The set of all finite paths is defined as

$$\mathbf{P} := \{\langle x_1, \dots, x_j\rangle \in N^* \mid \forall i \in \{1, \dots, j-1\}.(x_i, x_{i+1}) \in F\}$$

We further define

$$\mathbf{P}_n^m := \{\pi \in \mathbf{P} \mid \pi = \langle n, x_2, \dots, x_{j-1}, m\rangle\}$$

to be the set of all paths from n to m.

We define the combined transfer function for paths in a straightforward way.

Definition B.4 — **Path transfer function.** Let π be path. The path transfer function for π given the set of transfer functions Φ is

$$\varphi_\pi := \begin{cases} id, & \text{if } \pi = \epsilon \\ \varphi_{x_j} \circ \varphi_{\langle x_1, \dots, x_{j-1}\rangle}, & \text{if } \pi = \langle x_1 \dots, x_j\rangle \end{cases}$$

Now we can make precise the *meet over all paths* (MOP) solution of the data-flow system.

$$\text{mop}(S)(n) := \bigsqcup \{\varphi_\pi(\iota) \mid \pi \in \mathbf{P}_s^n\}$$

The MOP solution for a node n is the least upper bound over all paths leading from the start node s to n. Intuitively, this is the most precise solution that any data-flow analysis could possibly compute, as it takes into account all paths that lead to a node but nothing else.

R Again, a more accurate term would be the *join over all paths* solution as we take least upper bounds (joins), but for historic reasons we speak of the MOP solution [NNH99].

Now that we have two possible solutions for the data-flow system, the obvious question is how the two compare. First, we observe the following.

Lemma B.5 The MFP solution $\mathrm{fix}(S)$ safely approximates the MOP solution $\mathrm{mop}(\Phi)$, i.e., $\forall n \in N.\, \mathrm{mop}(S)(n) \sqsubseteq \mathrm{fix}(S)(n)$.

This intuitively makes sense, as we implicitly consider more paths through the CFG than necessary when we perform the fixed point iteration. A formal proof can be found in [NNH99, Lemma 2.32] Thus, in general, the fixed point construction is a source of additional over-approximation. That is to say, the fixed point iteration may yields results that are less precise than the analysis of all paths through the program, the MOP solution. This is not surprising, as the latter problem is undecidable for some analyses [NNH99, Lemma 2.31]. It is, however, well known that the two solutions coincide if all data-flow equations are *distributive*.

Theorem B.6 — **Intraprocedural coincidence theorem [KU77].** If S is distributive, the MFP and MOP solutions coincide: $\forall n \in N.\, \mathrm{fix}(S)(n) \sqsubseteq \mathrm{mop}(S)(n)$

In Section 5.2, we developed an interprocedural framework with the same property. (Which we were even able to use for parallel programs, because we could analyze them as if they were sequential interprocedural programs thanks to our deterministic semantics.) This gave us a way to effectively perform an interprocedural analysis with the full precision of the MOP solution.

APPENDIX C

Towards an Implementation

Introductory remark

During the time that I worked on this thesis, my focus shifted: Initially, I concentrated largely on an implementation of Jansen and Noll's paper on interprocedural analysis [JN14], with an aspiration to lift the analysis to parallel programs afterwards. During this process, I became increasingly interested in writing a coherent introduction to and developing a formal justification for hypergraph-based program analysis.

On the one hand, this formalization has diverged from the semantics that I implemented, especially regarding parallel programs. On the other hand, the formalization ended up taking so much time and space (you are reading page 117, after all) that I was not able to get the implementation to the degree of maturity that would warrant a dedicated chapter in the main part of this thesis. I instead decided to end this thesis on a somewhat lighter note[1] with an informal appendix that introduces my prototypical implementation.

High-level overview

My implementation is written entirely in `Scala`.[2] I started from scratch, i.e., without reusing parts of the (Java) Juggrnaut implementation to have the freedom to design my own hypergraph data structures and to be able to work completely in Scala. It would, of course, be possible to integrate the two implementations, as Java and Scala are both JVM languages.

The implementation follows the same approach as the thesis: I implemented both the concrete and the abstract semantics ($\Rightarrow$ and $\xRightarrow{A}$ in the notation of Chapter 3), and then built the interprocedural analysis as a rather thin wrapper around the semantics, which basically just applies the rules of the semantics to the postconditions of the procedure contracts, as described in Section 5.3.

I developed both a simple command line interface (CLI) and a browser-based graphical user interface (GUI), which I will discuss some more in a dedicated paragraph later.

[1]No definition to be found here, for a change!

[2]More specifically, Scala 2.11, using Scala's own combinator parsing and XML libraries for I/O, scalaz for additional functional programming support, spray for building the REST/HTTP-layer, ScalaTest for unit tests, and Typesafe's Scala logging library. There are also a few snippets of JavaScript that make the web interface a little more convenient to use, but these are negligible.

The implementation of the sequential semantics and analysis is reasonably complete[1], whereas the implementation of the parallel semantics is not yet finished. Overall, the implementation currently amounts to ca. 6500 lines of code (not counting comments, blank lines, and unit tests). These are distributed roughly equally among the following three areas.

- Data structures (hypergraphs, HRGs, abstract syntax trees, control-flow graphs) plus core algorithms (abstraction, concretization, canonical heap computation)
- Symbolic execution (i.e., implementation of the semantics) and interprocedural analysis
- CLI and GUI, including (combinator) parsing of the input files and dot export

Data structures

I mostly used a functional programming style, and hence all core data structures (hypergraphs, grammars, control-flow graphs, etc.) are immutable. To this end, my implementation makes heavy use of Scala's built-in collection framework, especially of (immutable) sets and (immutable) sequences. A hypergraph in my implementation, for example, essentially consists of an immutable set of nodes, an immutable set of edges, and an immutable sequence of external nodes. This is not a problem from a performance point of view, because Scala's collection are persistent: They are implemented using structural sharing, so that prepending to an immutable list is, for example, an $\mathcal{O}(1)$ operation.

Nodes have a unique identifier to simplify substitution of subgraphs (in grammar rule application or upon procedure return; I briefly touched upon this when discussing the cost of symbolic execution on page 96): We never have to check for duplicate nodes or rename nodes, apart from the assignment of unique identifiers to the nodes in production rules whenever they are applied.[2] An edge is nothing more but a sequence of nodes, a label, and a type to distinguish between variables, selectors, and nonterminals.[3]

Whenever we need to compare hypergraphs for equality, we compute the graphs' canonical representatives roughly as described on page 95.

Contracts are represented as sets of precondition–postcondition pairs, where each such pair consists of a hypergraph and a set of hypergraphs, i.e.,

```
type Contract = Set[(HyperGraph, Set[HyperGraph])]
```

(This is a bit of an oversimplification, but it boils down to this.) Here I use the fact that a partial function can be regarded as a set of pairs. Because I defined graph equality as equality of canonical representatives, Scala automatically only keeps one graph per isomorphism class in the postconditions. I use the same equality check to detect whether precondition–postcondition pairs have to be merged, because their preconditions are isomorphic.

All in all, the core data structures are therefore quite close to the mathematical representation chosen in the main part of the thesis. The other data structures (such as syntax trees and control flow graphs) are all straightforward and hence not discussed here.

Algorithms

The IPA implementation is not exactly straightforward, but it does not contain any surprises that would merit a detailed discussion. A brief overview of the more interesting algorithms follows.

[1]There are still a few rough edges at the time of writing.
[2]Assigned through one of the very few side-effecting commands that are not concerned with I/O.
[3]Realized as so-called case classes.

- On-demand concretization simply applies all applicable rules that remove at least one violation point. I implicitly assume that this is a language-preserving operation, but do not enforce anywhere in the code that the given HRG is actually locally concretizable. Rule application consists in removing the nonterminal edge, adding (uniquely named copy of) the right-hand side to the graph, and connecting the attached nodes of the nonterminal edge to the external nodes of the right-hand side in the correct order.
- In the abstraction process, we have to match the right-hand side of the production rules against subgraphs of the heap. To this end, many (very small) instances of the subgraph isomorphism problem are solved. Because the right-hand sides of the rules are very small and the heaps have enough structure to quickly rule out most node combinations, this has so far not been a performance bottle neck.
- Implementation of the inference rules is mostly straightforward. The reachable fragment computation in the call semantics is a simple breadth-first search. The return semantics are not difficult either, because the external nodes of the caller heap are not abstracted by the callee, and we can hence just "glue" the returned fragment to the caller heap at the external nodes.
- Canonical representatives are computed a little differently than suggested in the main part of the thesis: I perform a breadth-first-search, with the twist that I use a queue of edges rather than nodes. I initialize this queue with the set of all variable edges, ordered by their respective label. Whenever a new node is visited, it is assigned the next unused positive integer and its outgoing edges are added to the queue, ordered by their label. This yields a unique representative of the graph's isomorphism class. (Under our usual assumption that the entire graph is reachable from the variables.)

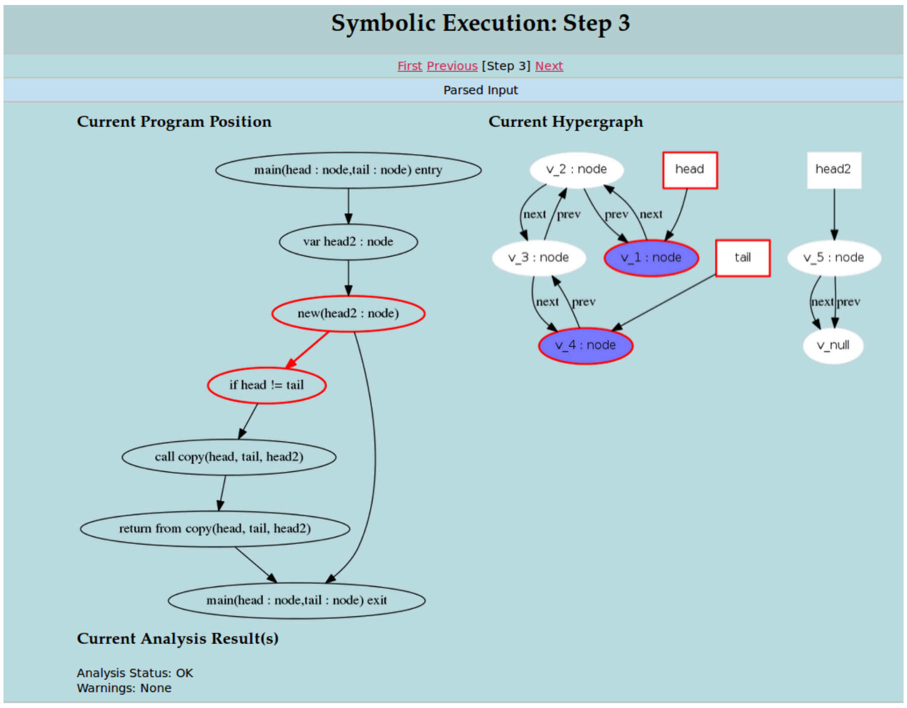
The user interfaces

As mentioned earlier, I implemented a CLI and a browser-based GUI. The latter consists of a simple interactive webservice, which I also implemented in Scala, and which directly uses the core algorithms and data structures. The webservice can be run on localhost using an embedded HTTP server; no installation is necessary.

Both interfaces receive their input as text files: The user places the PL programs and type declaration, the heap abstraction grammars, as well as the preconditions for the main procedure in files, then creates a configuration file which defines both a suitable set of input files and a few parameters that control the execution of the program.

The CLI performs a fixed task specified in the configuration file, whereas the web GUI lets the user choose whether she wants to explore the semantics (symbolic execution) or wants to perform an interprocedural analysis. In both cases, there are interactive step-by-step modes and "full execution" modes, which run or analyze the program until termination and report the result. In the interactive mode, the GUI displays control flow graphs, HRGs, heaps, and contracts as they are generated. Graph layouting is performed by graphviz, based on a straightforward export from my hypergraph and CFG data structures to the dot format.

I would like to conclude this thesis with two screen shots from the web GUI: Figs. C.1 and C.2 show the interactive mode "in action". The former shows an intermediate step of the symbolic execution on concrete graphs, i.e., the execution of $\Rightarrow$. The latter shows a single precondition–postcondition pair as generated during the interprocedural analysis.


Symbolic Execution: Step 3

First Previous [Step 3] Next

Parsed Input

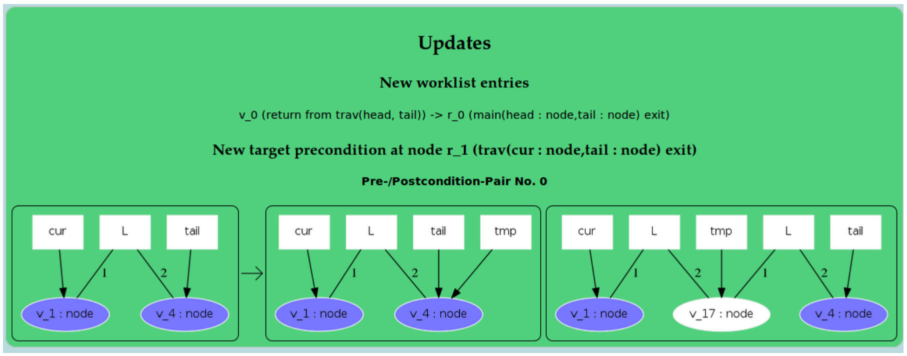
Current Program Position

Current Hypergraph

Current Analysis Result(s)

Analysis Status: OK
Warnings: None

Figure C.1.: An intermediate step in the symbolic executing using the concrete semantics $\Rightarrow$ as displayed in the browser-based GUI. The left-hand side shows the current control-flow graph. Each node of the CFG is labeled with the command that it represents. The next edge in the control-flow graph is highlighted. On the right-hand side, we see the current heap. Each ellipsis represents a node in the heap graph. It is labeled with a unique identifier and the data type of the node. External nodes are displayed on a dark background. Following the convention of the thesis, variable edges are displayed as rectangles, whereas pointer edges are represented by ordinary edges.


Updates

New worklist entries

v_0 (return from trav(head, tail)) -> r_0 (main(head : node,tail : node) exit)

New target precondition at node r_1 (trav(cur : node,tail : node) exit)

Pre-/Postcondition-Pair No. 0

Figure C.2.: An intermediate step in the execution of the worklist algorithm that computes the fixed point of the IPA DFS, as displayed in the browser-based GUI. We see that a new precondition–postcondition pair has been added to the contract at node r_1. It corresponds to the execution of the command $tmp := curr.next$ in a simple list traversal program.

Index