TreeGen: a monotonically impure functional language

by

Alistair Finn Hackett

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters in Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We present TreeGen, an impure functional language designed to express, consume, and validate JSON-like documents, as well as generate text files. The language aims to provide a more reliable and flexible way to create customised Interface Definition Languages, since the current state of the art is implemented via monolithic, ad-hoc codebases, which cannot easily be modified.

TreeGen's principal contribution, aside from being tailored to the domain of manipulating documents and generating text, is the concept of monotonic mutability: despite being an impure scripting language, its execution remains deterministic under arbitrary reordering of operations, making it robust to many common classes of programmer error possible in languages that allow unchecked mutability. We prove this by basing TreeGen's unordered constraint-based formal semantics on a partially-ordered model of TreeGen's heap, then showing that the execution of any TreeGen expression's constraint set is deterministic under chaotic iteration.

We also give notes on our experience implementing the language. These notes include a model for execution tracing and error reporting, necessary data structures to practically implement the formal semantics, related performance issues, and comments on potential mitigations of those performance issues.

Acknowledgements

I would like to thank Ondřej Lhoták for giving patient criticism and suggestions on the many previous drafts of this thesis. While I am reasonably comfortable as an implementer and software architect, I am relatively new to developing and writing up formal correctness arguments, especially at this scale. This has taught me a lot, both things that I can do now that I could not before, and skills that I now know I want to work on going forward.

I would like to thank Jo Atlee for her insightful advice on how to navigate some of the administrative aspects of this project, including the suggestion to go write the Mitacs Accelerate proposal that started it all. Without that push, this thesis as it is now would not exist.

I would like to thank PDFTron Systems Inc. for partially funding this project as part of a Mitacs Accelerate grant, as well as for playing along with my opportunistic reply to a recruitment e-mail and providing this project's motivating design problem.

I would like to thank my friends and family for listening to my ramblings as I work through things.

Dedication

To Kira, Kaddy and Martin.

Table of Contents

List of Figures

List of Listings

Chapter 1

Introduction

TreeGen is a monotonically impure functional programming language. It is designed as part of a wider project aiming to improve the state of the art in domain-specific code generation utilities. This thesis focuses mainly on novel aspects of the language itself, principally its monotonic treatment of impurity, but for context we begin by discussing the problem whose solution the language's semantics are designed to facilitate.

The context for this task comes from PDFTron Systems, a Vancouver-based company offering a suite of software libraries for processing documents, which was looking for better ways to generate the code required to define bindings between their core algorithms and the various language and platform bindings they offer to clients.

These bindings take the form of code and configuration files, in a variety of languages and formats, that provide access to a core library of code (in this case, C bindings to a C++ library) from any number of other programming languages across various platforms. Examples of other platforms, relative to C-language bindings, are bindings to scripting languages like Python, Ruby, Perl, JVM bindings, .Net bindings, or JavaScript bindings to a version of the C/C++ code that has been cross-compiled to either asm.js or WebAssembly for execution in a web browser. All of this diversity means that the only thing it is reasonable to assume of the bindings we want to generate is that they are represented by a group of text files in any number of possible languages. This representation covers binding mechanisms for essentially all mainstream programming languages and platforms, with the generation of code in non-textual languages [3, 25] being left as a future exercise.

While the output can vary drastically as discussed above, there is one thing that stays the same: the underlying software for which the bindings are to be generated. This allows the different platform bindings to be generated from some central source of information that

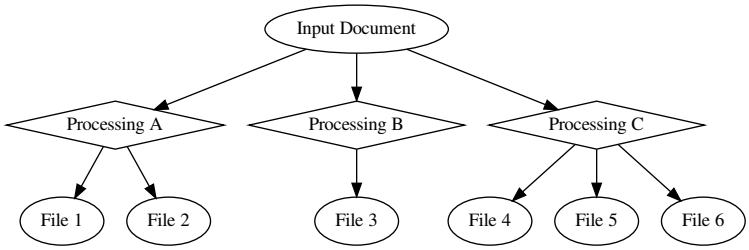


Figure 1.1: Abstract architecture of a code generator

abstractly describes how to interact with the underlying software. While its exact structure and contents may vary situationally – different kinds of software can have substantially different underlying assumptions – we call this data source the *input document*.

As explored in more detail in chapter 2, none of the existing tools for bindings generation are flexible enough to be ideal for the problem at hand – their input document formats are generally tailored to a particular domain and cannot be easily adapted. The well-established all-round option, SWIG [1], is tied to a specific model of C/C++ interfaces and data types, whereas related systems based on Interface Definition Languages (IDLs) [10, 17, 18] are tied to their own particular models. Furthermore, tools designed to more abstractly target text-file generation in general tend to be targeted at authoring human-readable content like blogs and other web content that benefit from heavily logic-restricted templating [21, 22, 37], with more powerful solutions relying directly on ad-hoc code in mainstream imperative programming languages [11, 14, 20], or specialised to operate on XML and XML-like data [23, 33].

As a result, it is currently impractical to directly express complex code generation logic via anything other than ad-hoc programs or plug-in libraries written in a general-purpose programming language. This situation leaves us with an architecture like that pictured in fig. 1.1, with an input document held separate from one or more monolithic processing steps, such as those described above.

Conceptually, the idea of having a separate input document and processor with a well-defined relationship is good, and it can work in large groups where a team can be dedicated to maintaining and evolving such a system. The problem is that this separation leads to friction, leading to some changes being trivial additions to the input document, while other changes that require adding new semantics to the input document become full feature requests to a piece of separately maintained utility code written in a general-purpose

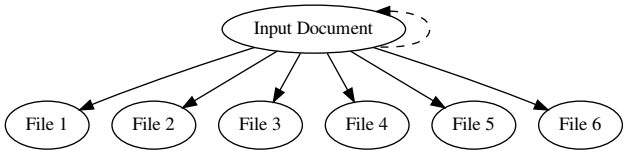


Figure 1.2: Abstract architecture of TreeGen as a code generator

programming language.

TreeGen's design philosophy is to generalise and extend the IDL-based design, while working around the inherent friction described above by allowing all aspects of a custom IDL to be expressed via a single specialised programming language. This requires that TreeGen be genuinely just as capable of expressing input documents and text templates as it is capable of expressing complex underlying logic, leading to a design looking like that in fig. 1.2, where the input document defines both static information and any logic necessary to generate the output files. This means that the input document becomes fully programmable, allowing it to contain both static information as with an IDL, as well as any specialised logic needed to guide output generation or add custom semantics to the input document, freeing the underlying TreeGen implementation from relying on anything more than the general model we have defined: an input document and some domain-specific logic that dictates the generation of a group of text files.

This design is unlike both existing domain-specific solutions that rely on implementation-level extensions for complex generation semantics, and just using a general-purpose programming language that may have little or no special support for document-oriented processes like parsing, validation, or ensuring deterministic output.

To satisfy these claims, TreeGen has the following key features:

- The ability to specify, give schemas for, and validate arbitrary documents with an underlying JSON-like structure in-line with other TreeGen code.
- Support for arbitrary cross-references within and between documents, irrespective of the typical restrictions found in general-purpose languages like definition or execution order.
- Runtime-enforced deterministic execution regardless of the order in which instructions are written or happen to execute. This is despite TreeGen not being a "pure"

language – the language supports mutation and side-effects in the tradition of other scripting languages, but with an ordering requirement that ensures programmers cannot write programs with timing-dependent bugs or unstable outputs.

- As stated in the assumptions, support for generating arbitrary text files. Per the previous point, this feature is also designed to either give deterministic output or raise an error.

1.1 Hello World, an illustrative example

```
1  {
2      type greet = { name: predef/string }
3
4      to_greet: predef/list/of (greet) = [
5          greet { name = "Jeff" },
6          greet { name = "Sophia" },
7          greet { name = "Ondrej" },
8      ]
9
10     effect predef/list/foreach {
11         list = to_greet
12         fn = \\ g =>
13             predef/file/generate {
14                 path = predef/path/relative [ "greet_${ g/name }.txt" ]
15                 contents = ''
16                     Hello, ${ g/name }!
17                 ''
18             }
19     }
20 }
```

Listing 1.1: A "Hello, World!" of sorts written in TreeGen

Listing 1.1 gives a "Hello, World!" of sorts in TreeGen, showcasing many of the language's key features. While most of this listing can be understood after reading chapter 3, it is useful as an introduction to give a superficial explanation of what each of the three main parts (line 2, lines 4-8, lines 10-20) do alongside notes on which design principle or principles are involved:

- Line 2 is a type declaration, expressing a simple schema. The type is named `greet`, and on the right-hand side of the `=` it uses a pattern notation to indicate that an

instance of `greet` is an object `{ ... }` (similar to those in JSON), with one field `name` that has the built-in string type `predef/string`.

- Lines 4-8 bind the name `to_greet` to the list expression delimited by `[ ... ]`. In addition to expressing the binding, we also include a type assertion that `to_greet` be bound to a value that satisfies the type `predef/list/of (greet)`, which intuitively means exactly how it reads: `to_greet` should be a list of instances of the `greet` we defined on line 2.

  Note that while it is in the standard library (bound to the `predef` object), `list/of` and similar "higher-order types" can be user-defined, acting as re-useable schema elements. The precise definition for `list/of` is discussed at the end of chapter 3.

  Each element of the list given on lines 5-7 is correspondingly an instance of `greet`, showcasing TreeGen's claim to naturally describe document-like constructs. While it is correct to say these are function calls to `greet`, due to TreeGen's determinism guarantees, it is equally natural to view `greet` as similar to an XML tag whose semantics are described on line 2.

- Lines 10-20 give an example of generating text files, in this case just simple files containing "Hello, <name>!". The `effect` is just an explicit way to say we have no use for the overall expression's output as we are evaluating it for side-effects – in this instance it could just as easily be replaced by a wildcard assignment `_ =`, which reads "assign to nothing".

  Moving into the expression, the next layer is `predef/list/foreach { list = ... fn = ... }`, which is a call to the standard library's list "foreach" function, which as in other list functions of the same name calls the parameter `fn` on each element of the paremeter `list`. The function `fn` in question is defined via TreeGen's lambda function syntax and uses the built-in file generation function `predef/file/generate` alongside template strings to generate different file names and contents for each `greet` in `to_greet`. The `"${ ... }"` indicate a location where a value should be spliced into the surrounding string, with double single quotes `'' .. ''` indicating an indentation-aware multiline string similar to those used by the Nix programming language [7]. `predef/path/relative` is a utility from TreeGen's standard path manipulation utilities that transforms a list of path segments into an instance of `predef/path/relative`, TreeGen's standard representation of relative filesystem paths.

  For generating more than one line of text, a library is available that allows declarative specification of key code-generation concerns like composable indentation management and fresh name generation.

Note also that if a more complex program were to inadvertently try to output different file contents under the same name, a runtime error would be raised preventing any such code from silently corrupting the output as it normally would in a scripting language without these safeguards.

Aside from particular line groups, note that TreeGen's deterministic evaluation guarantee allows any of these three elements to be placed in any order without affecting the overall result. That in and of itself is achievable in many other languages, but TreeGen supports the same reordering guarantees even in much more complex situations such as "concurrent"[1] function calls.

1.2 Thesis outline

This thesis is laid out as follows:

- Chapter 2 provides a survey of comparable code generation tools and some related programming languages, giving further context for the claims in this introduction.
- Chapter 3 informally introduces the majority of the TreeGen language, providing an in-depth look into what the language is capable of and giving context for chapters 4 and 5.
- Chapter 4 provides formal specifications for structures underlying the language, then introduces formal semantics for a subset of the language called *core TreeGen*. This subset itself is translated into unordered sets of constraints, whose semantics are shown to execute deterministically via chaotic iteration.
- Chapter 5 recovers the higher-level semantics missing from core TreeGen. It provides a grammar and key lexical edge cases for the full language, then reduces the full language to core TreeGen by specifying a series of syntax-driven rewrite operations. These rewrites double as precise definitions of the semantics of higher-level TreeGen features such as patterns and types.
- Chapter 6 provides insight into what is needed to implement a TreeGen interpreter in practice. This covers important data structure-level adaptations to the formal semantics, benchmarking results that highlight performance concerns with the current

[1] Concurrent in the sense that they have no defined sequencing, not to imply that we present any multithreading features.

prototype design, and notes on how to provide execution traces that help programmers debug failed program executions.

- Chapter 7 summarises what has been presented and makes note of some future work possibilities, including potential solutions to performance issues measured in chapter 6.

Chapter 2

Motivation and Related Work

This chapter gives more context to the claims in the introduction by examining a representative set of tools for generating API binding code, as well as some related document templating systems. We also situate TreeGen's treatment of mutability relative to both other scripting languages and pure functional programming.

2.1 API binding generators

Existing work on the topic of API binding generation can be broadly categorised as mostly-automatic interface generators [1], different kinds of Interface Definition Language (IDL) [10, 17, 18], common runtimes with interoperability functions among languages that operate within that runtime [13], expressive language-specific solutions [39], and more generic tools that use term manipulation [32]. We will discuss each of these in turn, noting advantages and disadvantages of these approaches.

2.1.1 Automatic or semi-automatic tools

The most prominent automatic interface generator is SWIG [1], a tool that scans C/C++ headers (along with some special directives that enable helper features like auto-instantiating C++ templates or defining Python docstrings) and generates bindings to a variety of other languages. SWIG is well-used in practice [24, 28, 35, 36] due to its generality and the low initial effort required. It is however inflexible, leading to at least two of the referenced

bindings, OpenCV and wxPython, being later rewritten to use custom code generators instead [27, 29].

Aside from other more specialised binding generators like ctypesgen for Python [6], research has been conducted into more effective automated translations. Ravitch et al [31] succeed in using static analysis to detect some C function calling idioms and translating them into Python equivalents, but fundamentally such systems can only deal with conventions for which a static analysis has been designed.

More generally, automatic tools like these successfully abstract away the glue code for calling functions across languages, but struggle to transfer many language or API-specific idioms due to the language or API-specific assumptions involved. This means that developers either have to write unidiomatically around foreign API calls, or they have to write more glue code to bridge semantics gaps between the two languages.

2.1.2 Existing interface definition languages

IDLs are mostly used within specific frameworks for specifying RPC APIs or interprocess communication, as in Apache Thrift [10] or Android IDL [17] respectively. There are examples of frameworks that specifically target heterogeneous programming like Docker Djinni [18], but these use restrictive type systems and have fixed semantics for how interactions may occur. Djinni for example is limited to expressing common value and container types, records, and interfaces with callable methods. This works well for their use case of linking Java, C++ and Objective-C, but might quickly become unidiomatic to use if one introduced custom data structures not recognised by the IDL's type system or tried to add support for a programming language for which the interface semantics were not a good match. The popular GObject framework [30] for language-agnostic object-oriented programming has similar issues: it works well for languages that are like itself object-oriented, but generates relatively unwieldy bindings in a pure functional language like Haskell [15].

2.1.3 Common language runtimes

Common runtimes as a means for interoperability are an interesting approach, especially in how they can lead to improvements in performance. The Truffle VM [13] for example mixes a low-level interoperability layer with cross-language compiler optimisations in order to run heterogeneous code very efficiently. While this can help generate very efficient code, this both requires a particular runtime environment and does not really deal with how to idiomatically interoperate among the hosted languages. Truffle's generic access framework

consists of a small set of generic low-level messages that allow accessing fields and calling methods, leaving mismatches among unique language features or idioms to either the user or some other tool.

2.1.4 Language-specific tools

There exist a wide range of expressive language-specific foreign interface tools. The ocaml-ctypes library [39] for example shows how generic programming can lead to very expressive declarative ways of generating interfaces between OCaml code and C. Alternatively, the Jeannie language [16] experiments with nested embeddings of C and Java, which implicitly generates glue code based on the Java Native Interface that handles interactions between the two languages. In general though, work in this area is unlikely to generalise past the language(s) to which it is specialised.

2.1.5 Declarative domain-specific languages

FIG [32] is a term-rewriting engine that progressively rewrites C definitions into foreign function interface definitions. While the authors evaluate it using the Moby language, it is intended to be a general system capable of outputting definitions in any programming language. FIG operates using a default automatic translation that can be overridden by custom term rewriting rules, leading to effectively arbitrary transformations as shown against the OpenGL library. While FIG does show that it is viable and useful for application-specific binding generation, it misses a lot of features that would be essential in a development context: support for outputting bindings to multiple languages with vastly different paradigms, ways of organising and re-using rules across bindings, and most importantly the system be sufficiently agnostic that a user can write a script that adds support for a new language or API concept.

2.2 Templating engines

In the introduction, we referenced some general-purpose templating engines. Here we give a detailed comparison between their design and TreeGen's.

2.2.1 Plain text templating

Plain, minimalist templating tools like Liquid [21], Handlebars [22] or Mustache [37] are a popular way to generate text files, especially in web-publishing contexts. These tools are used extensively. For example, Liquid is the underlying templating engine used by Jekyll, Github Pages' default static-site generator.

The default behaviour of one of these templating engines is to just output the input file, with specially escaped tags allowing the programmer to dictate repetition and text substitution. These tags are quite limited however, with any specialised code needing to be implemented separately as an add-on module ("helper" for Handlebars, "lambda" for Mustache, and "custom objects, tags, and filters" for Liquid) to the templating engine. While this model is clearly satisfactory for generating templated web content, it falls short on code generation tasks due to its inability to express complex logic without offloading the task to a special plugin.

As an alternative to the minimalist languages described above, more expressive languages exist that follow the same metaphor of interleaving imperative control-flow and text generation. The most well-known example is PHP [14], which operates on the same basic principle of outputting all text outside special tags `<?php ... ?>`. The difference compared to the minimalist templating languages described above is that there is a complete, mature imperative programming language available to perform relatively complex logic. This same metaphor can be attributed to similar offerings such as ASP.Net [20], its predecessor ASP, and the Java-based JSP [11]. The main risk behind using this kind of tool is that the underlying imperative-language semantics, while intuitive to users of mainstream programming languages, are prone to the usual issues possible in an unrestricted imperative program: the flow of information may not be visually obvious, and invariants relying on side-effects may be accidentally broken during maintenance or refactoring. In comparison, TreeGen allows programmers to still use side-effects when useful while preventing the kind of inconsistency to which imperative programs are prone under reordering.

2.2.2 XML querying and transformation

XQuery [33] and XSLT [23] are mature W3C-standardised languages specialised in dealing with the querying and transformation of XML documents. Due to TreeGen's document-oriented nature these languages have provided a starting point for TreeGen's semantics, with TreeGen retaining some elements of XQuery's syntax such as "`/`" being used for projection operations.

Both XQuery and XSLT are based on the XPath [8] expression language, a language based around "paths" through an XML document. An XPath expression describes a stream of 0 or more XML "nodes", with operators available to navigate attributes and nested tags, filter the query result via arbitrary conditions (including cross-referential lookups), concatenate two node streams, as well as special operations to do things like access sibling nodes in the XML tree or filter, filter every second node, and so forth. On top of using XPath to perform queries, both XQuery and XSLT add the ability to insert the yielded nodes into XML-based templates, splicing queried nodes into a single output XML document. This approach is quite powerful, allowing for expressive data querying and reporting as well as what becomes effectively a declarative XML-specific version of the PHP-like languages discussed in the previous section. The eXist-db database [9] is an example of this approach, acting as a complete database and web-service engine that can serve interactive websites whose logic is coded entirely in XQuery.

While TreeGen shares some ideas and syntax with these languages, there are also some important design differences that distinguish TreeGen, the most obvious being that TreeGen needs to output general-purpose text and not XML. While XQuery is capable of outputting plain text in principle (a limited port of the already-discussed Mustache to XQuery exists, for instance), the process is unwieldy due to the language's specialisation to XML and XML-compatible formats like (X)HTML. This problem on its own could be addressed by providing a variant of XQuery featuring easier to use text-handling primitives, but more fundamental issues exist that are discussed below.

Separation of input document from processing

Part of TreeGen's design objective is to allow free mixing of the input document and code. This objective is quite different from the objectives of XQuery and XSLT, which aim to express complex queries, reports and transformations based on a fundamentally separate input document. There is some work on adapting XML-based query languages to do something similar to TreeGen's ability to take part in generating the input document, but they do not align well with TreeGen's motivation.

The XQuery Update Facility [26] provides granular operations for updating parts of an XML document via node-level splices, though the primitive nature of these operations would make any pervasive use of them quite unwieldy compared to TreeGen's ability to directly insert function calls into the input document.

XPathLog [38] provides a Prolog-like logic language capable of expressing declarative updates to the input document by allowing the use of XPath in the logical "head" position.

This allows an XPath expression to not only query paths through the input document but to also assert that paths in the input document should exist. Unfortunately this solution, while very elegant for concisely expressing changes to an external XML document, would likely become as unwieldy as the XQuery Update Facility if used to the same effect as TreeGen's direct function calls.

Output uniqueness

A key design principle of any query language is that we expect any number of results. Essentially all query languages are designed such that query sub-expressions yield a stream of results, with compound expressions combining these results. In fact, even general-purpose declarative languages from the Prolog family do this in their own way, with a program being formulated as a query that may have any number of results. Accidentally writing a query that gives no results is a common mistake in such a language, and since there is no way for the implementation to know whether this was intentional without consulting the programmer, all query-centric languages must propagate an empty result all the way to the user without error. In historical Prolog implementations, this requirement would lead to the frustratingly unhelpful output "no".

TreeGen on the other hand is designed to describe one unique set of text files, with this assumption leading to each sub-expression being expected to describe exactly one value. This is why TreeGen does not say "no", and instead can give a precise error message at the first indication that no result is available. This is also why TreeGen has deviated significantly from XQuery, since most of the features for implicit and explicit iteration become unnecessary once we know each expression can only yield a single result.

Conversely, any query in a query language will generally result in one report that is defined to be the result of the combined query expression. This is not the case for TreeGen, where the same program may yield any number of output files required to express the generated codebase. There is no easily defined correspondence between TreeGen's expressions and the intended result, so instead of relying on expression structure, the generated files occur purely as side-effects.

2.3 Scripting languages and mutability

As TreeGen is a scripting language, it makes sense to compare it to existing general-purpose scripting languages in terms of the design tradeoffs made.

Scripting languages are a good option for rapid application development, offering a lot of flexibility to the developer alongside a typically short path from writing code to running it. This is true of many languages, be it the currently popular Python or the well-established LISPs. It is for these advantages that TreeGen is also fundamentally a dynamic language – when generating code there is no need for high performance or ahead-of-time typechecking as the entire execution likely has no user-facing results without prior review by a developer.

The tradeoff made with general-purpose scripting languages, however, is that this flexibility allows the programmer to write all sorts of programs, including programs that behave in ways they may not have intended. One particular "problem feature" common to scripting languages (including LISPs) is mutable data. It is very powerful and efficient when used correctly, but without careful discipline it can lead to confounding bugs and erratic program behaviour.

```
1 lst = [0, 1, 2]
2
3 def put_5(lst):
4     lst[1] = 5
5
6 put_5(lst)
```

Listing 2.1: A Python program that uses mutation

Consider the Python program in listing 2.1. While small and contrived, it aims to illustrate a programming pattern that can be problematic at scale.

Python's lists are mutable and passed by reference, so we can write a function called `put_5` that takes a list `lst` and sets one of its elements to the number `5`. While this is a perfectly legitimate Python program and the concept is easy to understand as long as the programmer remembers what `put_5` does, for large programs it can become difficult to keep track of. This can result in the following issues:

- When does the programmer expect `put_5` to be called? How far can a maintainer move line 6 without causing bugs?
- Will code that erroneously accesses `lst[1]` before it is set give a clear error, or will it misbehave in a way that is hard to diagnose?
- Does any code rely on the starting value of `lst[1]`? This might not even be intentional, since as long as nothing is disturbed, code with such a hidden reliance can still pass test cases.

These aforementioned issues are not specific issues with Python, but rather the common hazards of using mutability when programming. While they can of course be mitigated by strict testing alongside documentation of code assumptions and invariants, it can be more effective to restrict the program such that the issues given above have straightforward answers.

2.3.1 Immutability

```
lst = [0, 1, 2]

def put_5(lst):
    # this concatenates the two-element list [lst[0], 5]
    # with a list slice taking all elements of
    # lst starting with lst[2], thus replacing lst[1] with 5
    return [lst[0], 5] + lst[2:]

lst2 = put_5(lst)
```

Listing 2.2: A rewrite of listing 2.1 in an immutable style

Listing 2.2 describes a common way to deal with mutability: to forbid it. Instead of mutating `lst`, the new `put_5` constructs a new list with all the same elements as `lst` but with the second element replaced by the number `5`.

This helps answer the problems stemming from mutability like so:

- It does not matter when `put_5` is called, as long as `lst2` is lexically available to the code that needs it.
- Which "version" of the list any given piece of code is supposed to access is defined by which name is given, `lst` or `lst2`.
- If any code relies on `lst` then it will be clear for the same reason as above.

Immutability is a stylistic choice in Python, but other pure functional languages such as Haskell or Standard ML enforce this at a semantic level. This approach is gaining traction over time since it is an easy way to force programs to behave more predictably. For instance, popular libraries are available for the ubiquitous web programming language JavaScript that facilitate immutable programming [19], and the language has had some built-in runtime support for immutability via object freezing since the ECMAScript 5 standard [4].

Full immutability is not perfect, however, as it can require complex monad-based constructs to achieve things that would be conceptually straightforward, if dangerous and hard to maintain, with mutability. Most pure functional languages have opt-out mechanisms that allow programmers to directly manipulate mutable memory cells, though how commonly these are used varies.

Haskell for instance rarely makes direct use of these devices, since many common uses of mutability can be replaced with lazy evaluation that Haskell supports by default. In this particular aspect TreeGen and Haskell have something in common: both of them deal with cyclic data structures via lazy evaluation, achieving similar levels of expressivity in that regard. TreeGen does not however share Haskell's functional purity, allowing for restricted but more robust uses of mutable data as discussed in the next subsection.

2.3.2 Monotonic mutability

The other way to deal with the principal hazards of mutable data is to use TreeGen's restricted form of mutability: monotonic mutability.

This allows us to rewrite listing 2.1 into the TreeGen program in listing 2.3, where we replace the placeholder value `1` with TreeGen's unknown literal `???`. This assumes we were not relying on the initial value `1` anywhere, making it clear that the second element of `lst` does not have a valid initial value – the alternative that we were somehow relying on a distinct, known original value is invalid under monotonic mutability.

The rest of the program directly corresponds to the Python version, aside from some adaptations to how mutating assignments and list operations are written in TreeGen. `effect` is used to visually distinguish between local operations and side-effects.

```
lst = [0, ???, 2]

put_5 = \\ lst =>
    // TreeGen does not have a built-in list indexing syntax,
    // so we use the list's head and tail fields
    // (like Scheme/ML linked lists)
    { effect lst/tail/head = 5 }

effect put_5 (lst)
```

Listing 2.3: A rewrite of listing 2.1 in TreeGen

This rewrite resolves each of the mutability-related issues posed by the original Python code without avoiding mutation:

- Accessing `lst/tail/head` "early" will be resolved by TreeGen's lazy evaluation, due to which operations on unknown values will wait for the initial unknown value to be updated to a useable "known" value.

  Put another way, due to TreeGen's determinism requirement, a maintainer can move line 9 wherever they like without worrying about disturbing any side-effects, as long as `lst` remains accessible.

- Any access to an unknown value leads to deferred evaluation, forcing ahead-of-time accesses to `lst/tail/head` to wait until it holds the value `5`.

- Unlike full mutability (and as hinted above), monotonic mutability allows only mutations from unknown to known (this is granular to individual values, so mutating an unknown field of a known object will work), so no code can ever rely on `lst/tail/head` having a known value other than `5`. Once `lst/tail/head` holds the known value `5`, it is an error for a different piece of code to change it to any other value.

Chapter 3

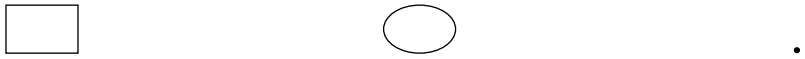
Language tutorial

The TreeGen programming language aims to provide a platform that combines the reliable, declarative aspects of non-programmable markup languages like XML and JSON with an expressive programming language tailored to IDL specification and implementation tasks.

These tasks include:

- Checking that the "input document", also defined in TreeGen, conforms to any relevant schemas and integrity conditions.
- Providing advanced cross-referencing capabilities to the "input document", such as "this function returns the type specified by that other interface definition defined elsewhere in the input document".
- Performing arbitrary logic, calculations and string manipulation in order to compute the contents of generated files.

This principle drives many of the language features presented here, with design decisions centered around providing an impure functional language that "feels" as much like writing fully declarative JSON-like markup as possible, JSON-like syntax being chosen as it is simpler to build on than XML. This "feeling" can be defined as maximal order-independence, both lexically and semantically. A TreeGen program's evaluation should be robust to definitions being moved around, and to implementation details of a function being changed to imply a different order of operations. A maintainer should not need a deep understanding of how a particular part of the code works, but rather should be able to rearrange the code they are working on as needed, like in XML or JSON, concerning



(a) Root value node (b) Value node (c) Call node

Figure 3.1: An illustration of the three kinds of node a document may contain

themselves only with obvious, lexically visible interfaces presented, freely assuming that any underlying computations are robust to the impact changes in how exactly their inputs are specified might have on the necessary order of operations. Equally importantly, a maintainer should be able, if needed, to customise and extend any code generation logic without switching languages, preserving the majority of their declarative, markup-like assumptions in the process, while allowing a coding style reasonably similar to that afforded by lightweight scripting languages such as Python or Javascript.

These requirements are approximated using a combination lazy evaluation, the restricted, monotonic form of null values motivated by section 2.3, and the specialised language constructs presented here. While the actual generation of text files is crucial to the motivating problem, almost the entire TreeGen language can be discussed without reference to it. Once the rest of the language is introduced, a fleshed-out, practical example including text file generation will be given, demonstrating how the features presented are expected come together in practice.

3.1 Documents

A TreeGen document is a collection of *nodes*. These nodes are usually *values*, which will be illustrated in diagrams via a circle or oval. The other two kinds of nodes are the *root value*, which represents the root of the document, and *call nodes* that are involved in notating the results of function calls. In a valid document, there is exactly one root value, and it is illustrated as a box. Aside from being specially marked as the root, the root value acts as any other value. Call nodes are a specialised auxiliary node that is illustrated as a point. They are not essential for a basic understanding of TreeGen, so they will be ignored until the introduction of functions. Figure 3.1 shows a root node, a value, and a call node arranged left to right.

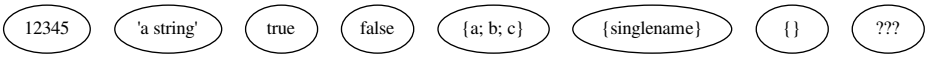


Figure 3.2: An illustration of values with the three essential kinds of shape

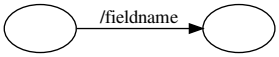


Figure 3.3: An illustration of the field relation between two values, shapes omitted

3.1.1 Essential shapes

Each value, including the root value, shall have a label that is called a *shape*. This shape indicates what is locally known about that value. The most essential kinds of shape are *primitives*, *objects* and *unknowns*. Primitive shapes can be integers like `12345`, strings like `'a string'`, the booleans `true` and `false`, and type tags `type 'name'`. Object shapes contain a list of names like `{a; b; c}`, `{singlename}`, or the empty list `{}`. There is one unknown shape written `???`, which as the name suggests, means that locally nothing is known about that value. Figure 3.2 illustrates value nodes that have each shape example discussed in this paragraph. The remaining two kinds of shape are *functions* and *type tags*, but they will be discussed once the first three have been properly demonstrated.

3.1.2 Essential relations

Aside from intrinsic shapes, values can have relations. The most common relation is the *field relation*, which is illustrated by a connecting arrow between two value nodes that has a label starting with a `/`. The field relation points from one value to another value, showing that the second value is a field of the first. For instance, fig. 3.3 illustrates two values where the value on the left has the value on the right as a field named `fieldname`. The shapes of the values are omitted.

Other relations are the *calls relation* and the *closure relation*, both of which are peculiar to functions. These will be introduced later.

When mixing shapes and the field relation, we get our first invariant: only values with

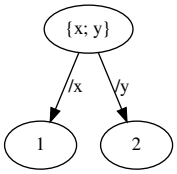


Figure 3.4: An illustration of a simple object with two fields

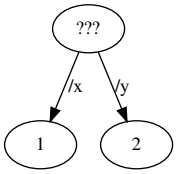


Figure 3.5: An illustration of a simple unknown with the same two fields as the object in fig. 3.4

unknown or object shapes can have fields. Primitive-shaped values cannot have fields, as they are considered leaf values in the document graph. Unknown values can have any field relations, and object values must have exactly one field relation corresponding to each name listed in their shape. Also, the same value cannot have more than than one field with the same name. This guarantees that paths through a document will not be ambiguous.

Figure 3.4 shows a basic example of field relations: we have an object-shaped value with two fields `x` and `y`, each pointing to values with primitive shapes `1` and `2` respectively. Figure 3.5 shows that it is also valid for the value having fields `x` and `y` to be unknown. This second example shows that unknown-shaped values can take part in any relation, since they may be later updated to have a shape for which those relationships are valid.

Lastly, it is important to note that the document graph is not acyclic: fig. 3.6 shows a similar object to that in fig. 3.4, but with the field named `y` changed to a self-reference to demonstrate a cyclic graph.

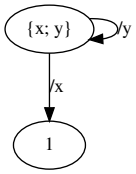


Figure 3.6: A modification of the illustration in fig. 3.4 that demonstrates the possibility of a cyclic graph

3.1.3 Implicit value deduplication

While the model described so far is very close to the complete model and could fully describe an alternate version of TreeGen, there is an additional feature to discuss: *implicit value deduplication*. In a similar spirit to the hash-consing optimisation from other functional programming languages, TreeGen has a set of rules for implicitly deduplicating values based on their shape and *local topology*, that is, any other graph nodes immediately related to the given value.

The deduplication rules vary depending on a value's shape. The rules for the essential shapes already discussed are as follows:

- For primitive-shaped values, there can be only one value with a given primitive shape. This means one could colloquially refer to "the value `1`" unambiguously, since there can only ever be one value with the primitive shape `1` in a given document.
- For object-shaped values, there can only be one value with a given object shape and set of immediate field relations. For example, fig. 3.7 shows an impossible situation: two object values with the same shapes and local field relations. In any situation that seems like it might produce such a result, the actual result will be the one shown in fig. 3.4 instead.
- For unknown-shaped values, there are no constraints. It is perfectly valid for a situation like that in fig. 3.8 to exist. Since we do not know anything about the unknown values involved, future manipulations might end up distinguishing these two values (one could gain an extra field that the other lacks), so it does not make sense to try to deduplicate an unknown.

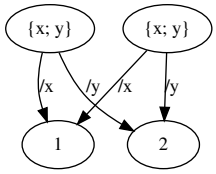


Figure 3.7: An illustration of the impossible case where two objects with the same field set and local topology exist

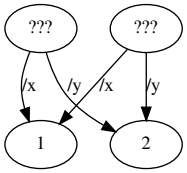


Figure 3.8: An illustration of two unknowns with the exact same field relations as each other

Rationale

The motivation behind deduplication is to semantically guarantee that for different syntactic constructions in the TreeGen language, the same graph node is produced. For instance, writing the same object literal `{ x = 1; y = 2 }` twice should give the same graph node, not just an identical value. While TreeGen does not use reference equality explicitly, this concept does exist and has an effect on the execution of function calls. In particular, when functions are introduced these deduplication semantics will have a very interesting effect on the expressivity of recursive code operating on cyclic structures.

For some practical purposes it might be quite reasonable to provide an alternate, simplified version of TreeGen not including these deduplication rules. While the resulting language would lack some of the regularity stemming from deduplication, it would share many of the desirable features of TreeGen and could potentially work around some performance issues described in chapter 6.

3.2 The merge operation

A TreeGen program may constrain two values to be the same. Aside from basic computations that generate values and other relations, there exists the *merge operation* which is used both explicitly and implicitly to force two values to be combined, affecting both the individual values and any relationships involving either value. This may also indirectly invoke the uniqueness invariant.

Here is a list of the rules involved when merging two values:

- Combining the same value with itself has no effect.
- Unknown shapes can be replaced by other shapes. Shapes that are not the unknown shape can be colloquially called *known shapes*. If both shapes are unknown then the resulting value still has an unknown shape, but if only one shape is unknown then the combined value will take on the shape that is known.
- If both shapes are object shapes with the same field names, then the values are combined successfully and retain the same shape.
- Notionally if two values have the same primitive shape then they may successfully combine as well, but due to the uniqueness rules it is not possible to have two distinct values with the same primitive shape.

- If two values have successfully combined at the shape level, their fields are combined as well. If two field relations with name `f` exist originating at each of the two values being combined, a recursive merge operation is applied to the pair of values that the field relations point to, ensuring that the resulting combined value has a single field relation with name `f` pointing to the recursively combined field value.
- It is an error to attempt a merge operation that leads to the merging of values with *incompatible shapes*, where incompatible shapes means that there does not exist a rule to merge the two shapes. An example of this is attempting to merge two values, one with primitive shape `1` and the other with primitive shape `2`. Similarly, merging a value with object shape `{a; b}` and a value with object `{b; c}`, or a value with primitive shape `3` is not valid. If this happens, program evaluation will stop immediately and an error will be raised.

There are some additional shape-specific rules for shapes that have not yet been introduced. These will be discussed later.

3.3 Essential syntax and semantics by example

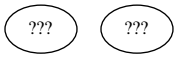
Now that we have introduced the essential components of a document in the abstract alongside TreeGen's unique merge operation, we need to show how these relate to programs you can write. These programs will be shown alongside graphical representations of their results and explanations of what they mean.

3.3.1 Unknown syntax

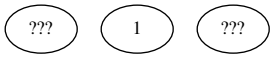
A simple expression that underlies a lot of what is interesting about TreeGen is the unknown expression `???`. The expression evaluates to a fresh unknown-shaped value, allowing other compound expressions to contain "holes". The operations that unknown-shaped values support will be described in more detail in later sections.

3.3.2 Object syntax

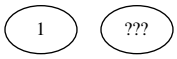
Since all TreeGen documents must start with a document at their root, it makes sense to start with the object syntax that all TreeGen programs must therefore use. Just the expression `4` on its own, for instance, is not a valid document.



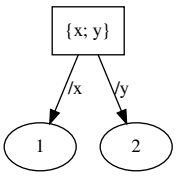
(a) Before evaluating either assignment directive, after initial setup stage of the object expression



(b) After only evaluating the right-hand side of the first assignment directive



(c) After evaluating the entire first assignment directive



(d) After evaluation has completed

Figure 3.9: Graphs showing snapshots of different stages of the evaluation of listing 3.1

Note that the unusual scoping mechanics presented here are intended to make TreeGen definitions robust to lexical reordering by default, allowing programmers to painlessly move code around in the majority of cases. This can however cause some counter-intuitive edge cases, which require unique workarounds. Coming from many popular programming languages that feature top-to-bottom lexical scoping, these issues may require a little getting used to.

```
{
    x = 1
    y = 2
}
```

Listing 3.1: A basic TreeGen program

Listing 3.1 shows a simple example of object syntax: the entire expression is enclosed in curly braces, and each field name of the object is assigned via an *assignment directive* to the result of some expression, in this case the primitive expressions `1` and `2`.

Of particular interest is the subtlety that order between the directives `x = 1` and `y = 2` is not important. While for this simple example it is easy to imagine that "it just works like JSON", which has similarly order-independent key-value maps, the details for how an expression evaluates in TreeGen are a little different. First all the field names are collected

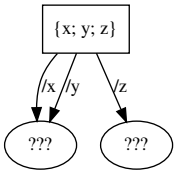


Figure 3.10: The complete document resulting from listing 3.2

and assigned to fresh unknown-shaped values as pictured in fig. 3.9a[1] (assume each of the unknown values is bound to one of the field names in program scope), then each assignment is evaluated in turn. For each assignment, first the right-hand side is evaluated, then the merge operation is used to combine the result of the right-hand side with the existing value bound to the field name. Figure 3.9b shows the document after just the expression `1` is evaluated, whereas fig. 3.9c shows the document after the entire assignment `x = 1` is evaluated and the value with shape `1` is merged with the till then unknown field value. The same applies to the second assignment, at which point the object-shaped value itself is constructed resulting in the final document shown in fig. 3.9d.

```
{
    x = y
    y = y
    z = z
}
```

Listing 3.2: A program illustrating cross-referencing in object syntax

This particular mechanism, as well as the fact that sibling fields can arbitrarily cross-reference each other by virtue of all field names being in-scope as unknowns from the start of the object expression, can be demonstrated by writing code like that in listing 3.2. Like before, all field names start as unknown and in scope. What's different is that since the field values are just references between fields, the field values keep their initial unknown shapes. Instead, the merge operation merely affects how many distinct unknowns there are. The assignment `x = y` merges the unknown bound to field `x` with the unknown bound to field `y`, leaving those two field relations pointing to the same unknown value. The unknown value

[1]Notice that this figure alongside figs. 3.9b and 3.9c does not have a root value. This is because these are snapshots of the partial document under construction. The root value only needs to be considered after the expression has finished defining it, not before.

bound to `z` is simply merged with itself, so it remains distinct. The resulting document is shown in fig. 3.10.

```
1 {
2     x = ???
3     y = ???
4     z = ???
5     effect x = y
6 }
```

Listing 3.3: A program equivalent to that in listing 3.2, demonstrating the effect directive

While careful manipulation of assignment directives can be used to force merge operations, there is a much more convenient and much clearer syntax that should be used to write this in general: the *effect directive*. Unlike assignment directives, effect directives allow both the left and right-hand sides of an `=` sign to be expressions, allowing the programmer to request the merging of the results of arbitrary expressions. Listing 3.3 combines this syntax with the expression `???`, which generates a fresh unknown value, to give an equivalent program to that in listing 3.2. While the exact evaluation path is a little different and involves each `???` expression generating a fresh unknown that is immediately merged into the unknown bound to the corresponding field name, the merge operation ensures that the resulting document is still the one shown in fig. 3.10.

Projection expression

Aside from the directives used for object syntax, the other key piece of syntax associated with objects is the *projection expression*. The expression means field access, taking an arbitrary expression on the left hand side and a field name on the right hand side.

Listing 3.4 shows some simple examples of projection expressions. Line 5 demonstrates the common case of projecting the field x from the object bound to the name `object`, whereas line 6 demonstrates the less common case of projecting from a compound expression `{ y = 2 }`. Incidentally, listing 3.4 also demonstrates the common practice of nesting the object expression in order to produce trees of data.

```
1 {
2     object = {
3         x = 1
4     }
5     getx = object/x
6     gety = { y = 2 }/y
```

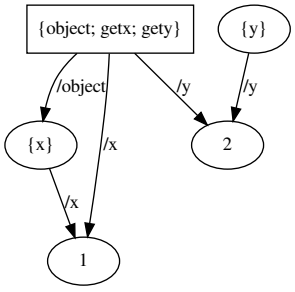


Figure 3.11: Document derived from listing 3.4

```
7 }
```

Listing 3.4: An example of the projection expression

Figure 3.11 illustrates the resulting document, showing the resulting cross-references. For explicitness, the figure also shows something that is usually omitted: the dangling object with shape `{y}`. Normally one only cares about values reachable from the document root, but it is important to realise that in normal TreeGen evaluation dangling objects will exist, coming from the various temporary expressions whose results are for some reason not fully incorporated into the final document.

Scoping of nested object expressions

Listing 3.5 shows how scoping and name resolution work in the context of nested object expressions. Since `name2` is referenced but not defined in, the sub-object acts as a regular compound expression and can reference `name2` in its enclosing scope. On the other hand, if a name like `name1` is bound in both the outer and inner object, the inner-most binding takes priority. This demonstrates a relatively easy programmer mistake, whereby an intended reference to `name1` defined at line 2 in the enclosing scope becomes a reference to the unknown field value bound to the same name on line 5. The problem is illustrated in fig. 3.11, showing how the nested `name1` is mis-bound to a fresh unknown value due to the scoping error in listing 3.5.

```
1 {
2     name1 = "a string"
3     name2 = 26
```

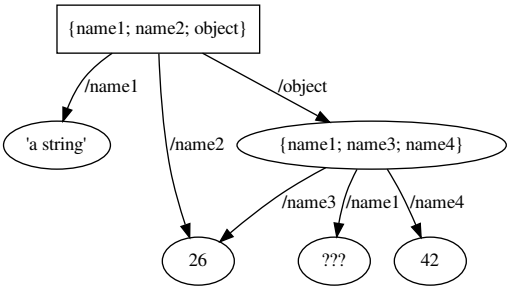


Figure 3.12: Document derived from listing 3.5

```
    object = {
        name1 = name1 // scope shadowing!
        name3 = name2
        name4 = 42
    }
}
```

Listing 3.5: A more natural basic document with objects and cross-references

Fixing the mistake in listing 3.5 is a good opportunity to talk about the two alternative ways to bind names in TreeGen, since a simple fix using only what we know is actually quite inconvenient.

Listing 3.6 shows a simplistic attempt to work around the scope shadowing by re-binding the outer `name1` to another name `noshadow`. While this works around the original problem, it introduces another issue that is illustrated in fig. 3.13: the root object now has a new field `noshadow` that was not previously present. This might be tolerable under some circumstances, but TreeGen has better ways to manipulate scoping.

```
{
    name1 = "a string"
    name2 = 26
    noshadow = name1
    object = {
        name1 = noshadow
        name3 = name2
        name4 = 42
    }
```

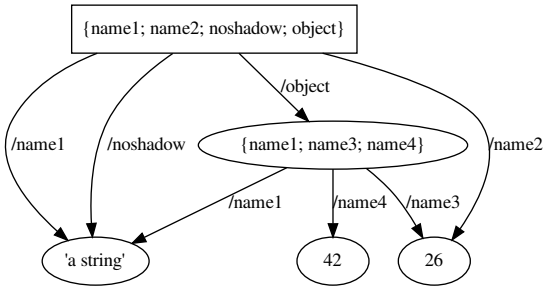


Figure 3.13: Document derived from listing 3.6

```
10 }
```

Listing 3.6: An unsatisfactory way to address the scope shadowing in listing 3.5

Let directives and dollar names

TreeGen provides two facilities that can be used to manipulate scoping: *let directives* and *dollar names*.

Let directives are an extension to let bindings, which TreeGen also supports. As one can locally bind a name `x` for use in an expression `e`, as in `let x = ??? in e`, let directives are a directive that adapts the concept of let bindings to being included in object expressions, somewhat like a let binding where the body is the enclosing object expression.

Dollar names on the other hand are a special kind of name that starts with a dollar sign, like `$x`. They operate almost identically to other names except for their effect on scoping: they are scoped non-recursively, only being referenceable in what is notionally the corresponding definition's "body" (as opposed to the bound value). This goes counter to the intuition behind many common programming languages where recursive definitions must be explicitly enabled via constructs like `letrec`, demand-computed definitions or mutable pointers, but it is important to keep in mind that for TreeGen's goal of by-default cross-reference capability, this intuition must be inverted.

These devices have orthogonal effects but related practical use cases, which we will introduce in more detail as part of the quest to improve listing 3.6.

```
1 {
```

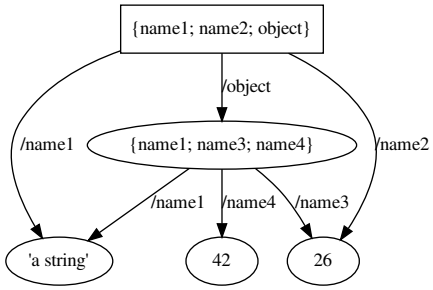


Figure 3.14: Document derived from listings 3.7 and 3.8

```
    name1 = "a string"
    name2 = 26
    let noshadow = name1
    object = {
        name1 = noshadow
        name3 = name2
        name4 = 42
    }
}
```

Listing 3.7: A better way to address the scope shadowing in listing 3.5

Listing 3.7 shows how to use a let directive to write a correct version of listing 3.6 resulting in the intended document without any extra fields, as shown in fig. 3.14. The key difference between a let directive and an assignment directive is that the let directive does not include the bound name in the resulting object. The let directive is generally useful for binding temporary variables in more involved object expressions while still benefitting from the ability to access the bound name out of order as with an assignment to the same name[2].

The version with let bindings still has its problems however: a new dummy name needs to be introduced, which means the programmer has to come up with some new throwaway alias for something they have already named. Not only does this introduce a redundant name, but the programmer will have to context-sensitively switch between these "inside"

[2]Remember that since out of order name references are performed with a mix of unknown values and merges, any lexically valid cross-reference follows the same rules for order independence regardless of whether object fields or free-standing values are involved.

and "outside" names, which from experience is not an easy task when working with larger programs.

```
{
    name1 = "a string"
    name2 = 26
    object = {
        $name1 = name1
        name3 = name2
        name4 = 42
    }
}
```

Listing 3.8: The most idiomatic way to address the scope shadowing in listing 3.5

Really this is not the best use case for let directives. They are for when you specifically want a let binding in-line with other directives, which misses the fact that `noshadow` was added as a workaround to begin with.

Listing 3.8 shows a different more idiomatic solution that does not introduce any substantially different names. This uses the other tool TreeGen makes available for manipulating scope: dollar names. Dollar names are valid where regular names are, but their definitions have different scoping rules: while a normal name assignment can be accessed from everywhere in an object expression, dollar name assignments (and let directives using dollar names) only allow the dollar name to be accessed in subsequent directives. Importantly, the dollar sign is not included in the resulting field name, leading to listing 3.8 evaluating to the same document as listing 3.7, shown in fig. 3.14. Either syntax has an equivalent effect here, despite the underlying mechanisms being quite different. This allows the programmer to prevent recursive references when they are not wanted, both allowing the simple one character workaround shown in listing 3.8 and allowing the more sequentially-minded code shown in listing 3.9.

```
{
    let $x = 1
    a = $x
    let $x = $x + 1
    b = $x
}
```

Listing 3.9: Sequentially-minded code using dollar names to allow rebinding `$x`

Listing 3.9 shows that because dollar names are only in scope in the directives that follow them, defining the dollar name `$x` multiple times with let directives (to avoid redefining a field named `x`) allows the programmer to give code an imperative flavor by repeatedly

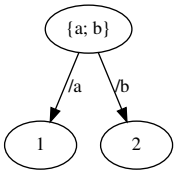


Figure 3.15: Document derived from listing 3.9

rebinding the same name at the expense of not allowing direct out of order cross-references. Figure 3.15 shows that this results in fields `a` and `b` having different values despite both binding `$x`.

Object merging and uniqueness

The rules for the merge operation include a case where objects with the same sets of fields can be merged.

```
1  {
2      a = {
3          x = ???
4      }
5      b = {
6          x = 1
7      }
8      c = a/x
9      effect a = b
10 }
```

Listing 3.10: Demonstration of object merging

If unknowns are involved as in listing 3.10, this can be used to "fill a hole" in one object by using another object. In this case, we merge the objects `a` and `b` in order to propagate information about `b/x` to `a/x` and, transitively, `c`. Figure 3.16 shows this effect, with the values bound to `a` and `b` being forced to be the same object by the directive on line 9.

```
1  {
2      foo = {
3          a = 1
4          b = 2
5      }
```

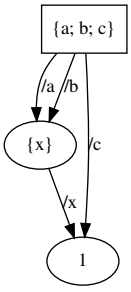


Figure 3.16: Document derived from listing 3.10

```
    bar = {
        b = 2
        a = 1
    }
    ping = {
        a = 2
        b = 2
    }
}
```

Listing 3.11: Demonstration of object uniqueness

Like object merging, object uniqueness is worth demonstrating via practical examples. As shown in fig. 3.17, evaluating listing 3.11 gives (aside from the root object) only two distinct objects. Since both `foo` and `bar` have the exact same fields and values, those names refer to the same singular object.

```
{
    foo = {
        a = ???
        b = 2
    }
    bar = {
        b = 2
        a = ???
    }
    ping = {
        a = ???
        b = 3
    }
```

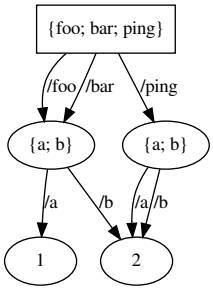


Figure 3.17: Document derived from listing 3.11

```
14
15      effect foo/a = bar/a
16      effect foo/a = ping/a
17  }
```

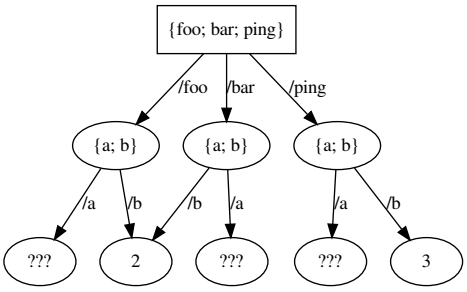
Listing 3.12: Interaction of object uniqueness with unknowns

The deduplication of objects over time can be shown by progressively introducing lines 15 and 16 of listing 3.12. Each step of this is shown in figs. 3.18a to 3.18c.

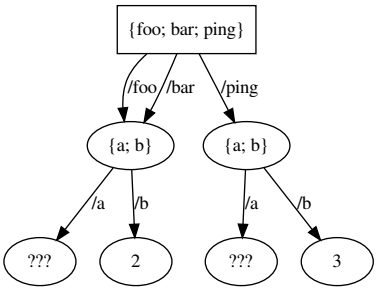
- In fig. 3.18a all the unknowns start off distinct, making each object also distinct as a consequence.
- In fig. 3.18b, `foo/a` and `bar/a` are declared the same value, making `foo` and `bar` also the same value as they no longer have any fields to differentiate them.
- In fig. 3.18c, `foo/a`, `bar/a` and `ping/a` are all transitively the same value, but `ping` remains distinct as its `b` field is still not the same as the other values' `b` fields.

3.3.3 Primitive expressions and operations

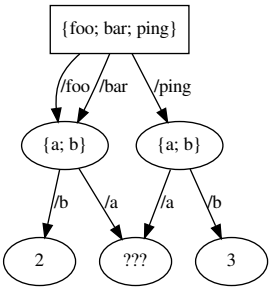
TreeGen supports traditional primitive-shaped values similar to those supported by other scripting languages: strings, integers and booleans. Type tags are also primitive shapes, but their operations are better discussed alongside type directives in section 3.5.



(a) Graph excluding lines 15 and 16



(b) Graph excluding line 16 only



(c) Graph derived from the entire listing

Figure 3.18: Graphs derived from listing 3.12 excluding lines 15 and 16, excluding line 16 only, or derived from the entire listing

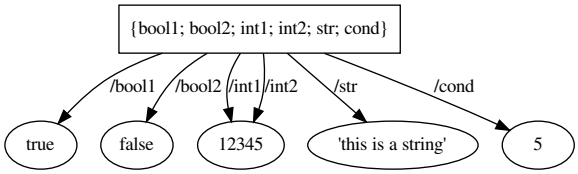


Figure 3.19: Document derived from listing 3.13

Primitive values support a standard set of operations mostly inherited from C-style scripting languages. Booleans are subject to the core logic operations like `&&`, `||` and `!` which correspond to "and", "or" and "not" respectively. Additionally, booleans can be used in conditional if-then-else expressions. Integers follow typical arithmetic, supporting the traditional `+`, `-`, `*` and `div` for plus, minus, times and integer-divide respectively. Strings support `++` for concatenation.

This list of primitive operations covers all those that have dedicated syntax, but it's important to understand the TreeGen's architecture makes it easy to add other primitive operations via the standard library and interpreter intrinsics. For instance, `predef/string/substring`, where `predef` is a built-in name that references the root of the standard library, is one such operation.

A demonstration of expressions commonly used in examples is given by listing 3.13, with the resulting document being illustrated by fig. 3.19.

```
{
    bool1 = true
    bool2 = false
    int1 = 12345
    int2 = 12344 + 1
    str = "this is a string"
    cond = if bool1 then 5 else 6
}
```

Listing 3.13: An example of primitive values

Looking more closely at the resulting figure, there is one detail that might be a little surprising coming from most general-purpose programming languages: the application of the deduplication rule. There is only one value with shape `12345`, despite the value being produced by two different expressions.

3.3.4 Operations on unknown values

In principle unknown values allow all types of operations. This does not mean all operations will work – it means that all operations might work. With the exception of field projection, all operations on unknowns launch a *deferred evaluation* that waits for the operation's inputs to be known as a side-effect of some merge operation. If the unknowns eventually merge with appropriate inputs to the operation, then the result is valid. If no further evaluation is possible, but deferred evaluations are still waiting once all other possible execution has completed, then that is an error.

Projecting from an unknown value

As mentioned above, projecting from an unknown value is the only operation that does not involve deferred evaluation. Listing 3.14 shows a simple example of projecting from an unknown value: if the value does not have a field of the required name already, then a fresh unknown value is created to be that field. The resulting document is illustrated by fig. 3.20a. Then, if the unknown is later merged into an object, that field can be merged into the object's field, retroactively making the projected value the same value one would get from applying the projection operation directly to the eventual object value. Adding the extra line `effect a = { x = 1 }` to listing 3.14 demonstrates this eventuality, with the corresponding document shown in fig. 3.20b.

```
{
    a = ???
    b = a/x
}
```

Listing 3.14: Example of projecting from an unknown value

Transitive side-effects of merging unknowns

One last thing to highlight about unknowns and fields is that merging two unknowns will merge their fields as well. Listing 3.15 shows this, with figs. 3.21a and 3.21b.

- In fig. 3.21a, `a` and `b` are distinct, meaning most other values are also unknown: the only thing that is derivable is that `b/x` is bound to `1`.
- In fig. 3.21b however, we know that `a` and `b` are the same value. Transitively, this means that `a/x` and `b/x` must be the same value, which in turn means `c` must be that same value, that same value being `1`.

(a) Document derived from only the listing

(b) Document with the additional effect directive

Figure 3.20: Documents derived from listing 3.14 with and without the additional effect directive `effect a = { x = 1 }`

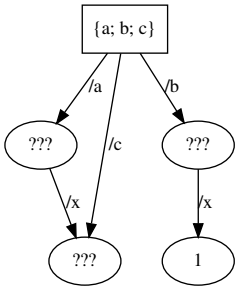
```
{
    a = ???
    effect a/x = ???
    b = ???
    effect b/x = 1
    c = a/x

    effect a = b
}
```

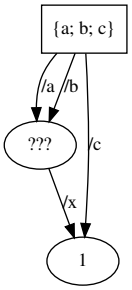
Listing 3.15: Demonstration of merging two unknowns that have fields

Primitive operations on unknowns

Using deferred evalution, unknowns can also be used with primitive operations. When a primitive operation on an unknown value like `a + 1` in listing 3.16 is interpreted, it initially evaluates to a fresh unknown value and posts a deferred evaluation that waits for the unknown to be merged with a known value. This allows primitive operations to be written in the same order-independent style as previous constructs, using unknowns and merging to deal with the resulting partial information. Because this system is agnostic of what exactly the deferred operations do, this concept can be used to extend the idea of out of order evaluation to essentially arbitrary computations as long as the underlying data can be transferred via the merge operation.



(a) Document excluding line 8



(b) Document including line 8

Figure 3.21: Documents derived from listing 3.15 excluding and including line 8

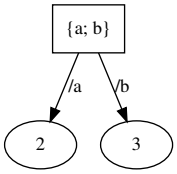


Figure 3.22: Document derived from listing 3.16 with the addition of `effect a = 2`

```
1 {
2     a = ???
3     b = a + 1
4 }
```

Listing 3.16: Demonstrating an invalid primitive addition on unknown values

As mentioned before, if a deferred operation is never able to complete because its inputs never merge with a known value, then that is an error – in a valid program all deferred evaluations should complete eventually. For example, listing 3.16 is not a valid program. Since there is no way for `a` to become known, the operation on line 3 will never complete, resulting in an error. Adding a directive `effect a = 2` however would enable `a` to become known, resulting in fig. 3.22.

```
1 {
2     c = c + c
```

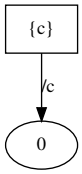


Figure 3.23: Document derived from listing 3.17

```
3     effect c = 0
4 }
```

Listing 3.17: Demonstrating how TreeGen allows potentially circular arithmetic statements

Listing 3.17 shows that TreeGen's semantics can resolve some relatively odd statements. `c = c + c` would not make much sense as a starting definition in most languages, but as long as we externally force `c` to have the only possible valid value as is done on line 3, the deferred computation will evaluate `0 + 0` to `0` and then successfully merge `0` with the value of `c`, which is already `0`. The overall result of this example is shown in fig. 3.23.

Note that if line 3 did not merge `c` with `0`, but rather something different, like `effect c = 42`, then line 2 would attempt to merge `42 + 42` with `42`, causing a runtime error. How such errors are reported is discussed in chapter 6.

3.4 Function values

Function values work in a similar way to functions in other functional languages, and are TreeGen's fundamental source of programmability. Functions are the last major shape available in Treegen, and introduce all of the remaining kinds of relation that can exist in a document.

A function shape is derived from a function expression, which is also the syntax used to produce a function value. In TreeGen, a function expression names a single argument, mapping it to an expression involving that argument. For example, a simple function expression can be written `\\ x => x + 1`. `\\` is a syntactic artifact that starts a function expression, `x` is the name of the argument, and `x + 1` is a function body. A function shape is a whitespace-insensitive copy of the same syntax, written `x => x + 1` which is the same expression except that the `\\` is missing.

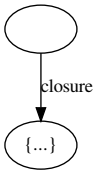


Figure 3.24: An illustration of the closure relation

Function values have a uniqueness rule with two criteria:

- Values with different function shapes are definitely different, but values with the same function shape could be the same.
- If two values have the same function shape, uniqueness depends on the free variables captured by the function's closure. The captured variables are modeled as fields of an object, so uniqueness of the function value depends on the uniqueness of that related object. Figure 3.24 shows how the *closure relation* is illustrated in general: there is an arrow labeled "closure" from the function-shaped value to the object holding the closed-over variables as fields. Specific instances of this configuration will follow.

Knowing the function uniqueness rule, the merge criterion becomes almost trivial: two function values may be merged if they have the same shape and the objects holding their respective closed-over values can be merged.

For example, the function shaped `x => x + 1` captures no free variables, so there can only be one function value with that shape. A function shaped `x => x + y` on the other hand is tied to the uniqueness of values captured via the name `y`. For every different value `y`, there is a unique function of shape `x => x + y`. Both of these examples are given in listing 3.18, with the resulting values illustrated in fig. 3.25.

```
{
    fn1 = \\ x => x + 1
    y = ???
    fn2 = \\ x => x + y
}
```

Listing 3.18: Some simple examples demonstrating function values

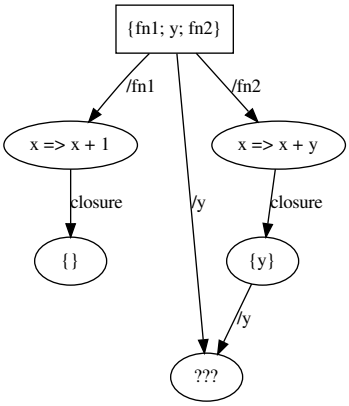


Figure 3.25: Document derived from listing 3.18

3.4.1 Function deduplication via closed-over values

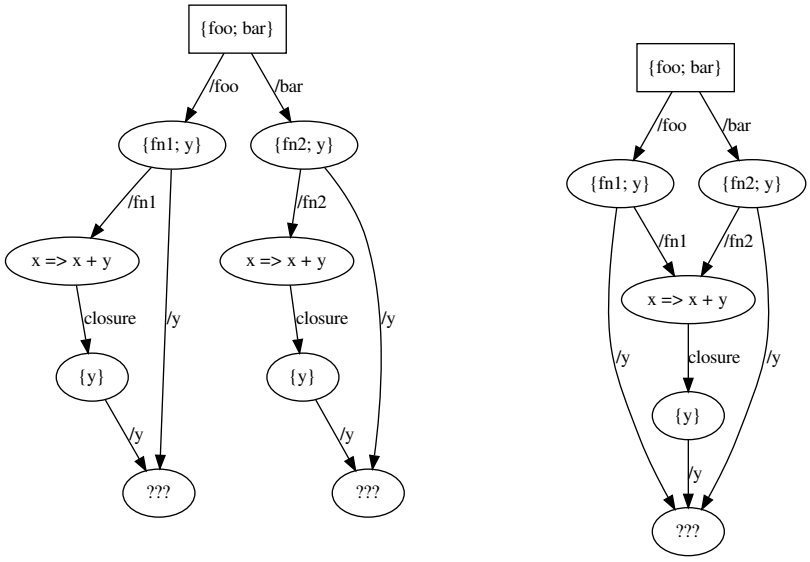
A more interesting side-effect of function uniqueness depending on closed-over values will be discussed here. Since we have defined functions to be unique via the pair of function definition and function closure, the same function definition can produce as many or few values as there are distinct closed-over values.

```
{
    foo = {
        y = ???
        fn1 = \\ x => x + y
    }
    bar = {
        y = ???
        fn2 = \\ x => x + y
    }

    effect foo/y = bar/y
}
```

Listing 3.19: Demonstration of function values being affected by their closures

Listing 3.19 demonstrates how, as long as they are syntactically identical and therefore able to unify at all, unifying values closed over by function values can cause the functions themselves to unify. In fig. 3.26a, `fn1` and `fn2` are initially distinct because `foo/y` and `bar/y`



(a) Document excluding line 11 (b) Document including line 11

Figure 3.26: Two documents derived from listing 3.19, including or excluding line 11

are distinct. When those two values are declared to be the same, this distinction disappears and fig. 3.26b shows the resulting document containing only one function value.

3.4.2 Calling a function

Function calls themselves are modeled in a subtle way that involves a deferred evaluation and an auxiliary ternary relation, which has its own uniqueness rule. The ternary relation is the *calls relation*, which records all function calls that have started evaluating. It holds references to the argument value, the function value, and the result value. It is illustrated in fig. 3.27 as a point representing an individual instance of the calls relation connected by arrows to the relation's three components.

When a function call expression is evaluated, first the call's relation is inspected. If there exists a record of a call to the specified function with the specified argument having started (and possibly completed), then the stored reference to the result value is simply returned. In this scenario, where there was already a record of an equivalent function

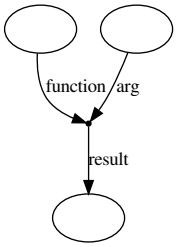


Figure 3.27: An illustration of the call relation, drawn as a point

call, no call really took place and the operation succeeds immediately, returning the return value held by the existing instance of the call relation.

If no record of that function call starting exists, then one is created with a fresh unknown result value and a deferred evaluation is posted waiting for the function value to be known. It does not matter at all whether the argument is known at any stage – it is simply passed along as-is with any requirements on it coming from its eventual use in the function body. The function value however has to eventually become known due to the deferred evaluation waiting on it. When this happens, possibly immediately if the function value is already known, the function body will be executed with the argument value bound to the argument name, with the resulting value being merged into the stored result value. It is at that point, once the body's result value has merged with the recorded function call result, that the function call can be considered to have completed successfully.

There are several subtleties to this process which will be discussed further on, but first the simple case. A function call has the syntax `fn (arg)`, where `fn` and `arg` describe arbitrary sub-expressions. A simple example of this is listing 3.20, with the resulting document illustrated by fig. 3.28.

```
{
    fn = \\ x => x + 1
    result = fn (1)
}
```

Listing 3.20: An example of a single function call

As a slightly more advanced concept, we can introduce the call relation's uniqueness rule: for every pair of function value and argument, there must be exactly one result value.

This can be demonstrated by listing 3.21, which relies on the uniqueness of unknowns.

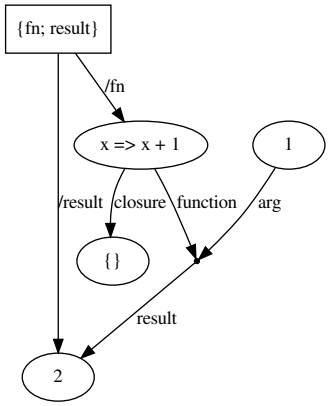


Figure 3.28: Document derived from listing 3.20

```
1 {
2     fn = \\ x => ???
3     a = ???
4     b = ???
5     result1 = fn (a)
6     result2 = fn (b)
7     effect a = b
8 }
```

Listing 3.21: Demonstration of function call uniqueness

Since `a` and `b` are distinct when the two function call expressions are evaluated (evaluation can be read as following directives top to bottom), one would expect that the function body is evaluated twice, producing two fresh unknown values that are bound to `result1` and `result2` respectively. That is not the end result however, as the uniqueness invariant for the calls relation forces the two return values to become one as a consequence of the merge operation on line 7. Figure 3.29 shows the resulting document containing only one instance of the calls relation, making `result1` and `result2` refer to the same value.

3.4.3 Non-function values can be called

While only functions have a body to evaluate and a closure under which to evaluate that body, the function call-related relations in a document can apply to values that are not

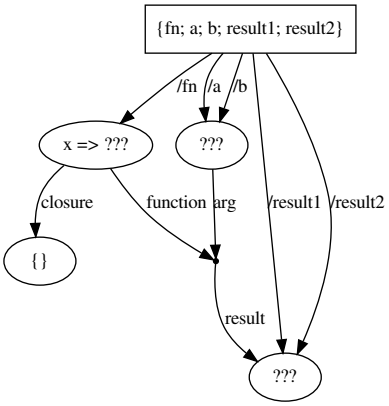


Figure 3.29: Document derived from listing 3.21

functions. As previously mentioned, one can call an unknown as long as the deferred operation can eventually resolve. Also, as a usability feature it is allowed to call objects that have a field named `apply` that is either a function or another object that itself has an `apply` field. `apply` fields may be recursively dereferenced ad infinitum.

As an example of the first case, consider that both listings 3.22 and 3.23 represent identical documents.

```
{
    fn = ???
    a = fn (1)
    effect fn = \\ x => x + 1
}
```

Listing 3.22: Transiently calling an unknown that resolves to a function

```
{
    fn = \\ x => x + 1
    a = fn (1)
}
```

Listing 3.23: Semantically the same document as listing 3.22, but with `fn` defined directly

To demonstrate the second case, listing 3.24 evaluates to the document shown in fig. 3.30. Notice that the calls relation only involves the function and not the object, even though technically the object has been "called".

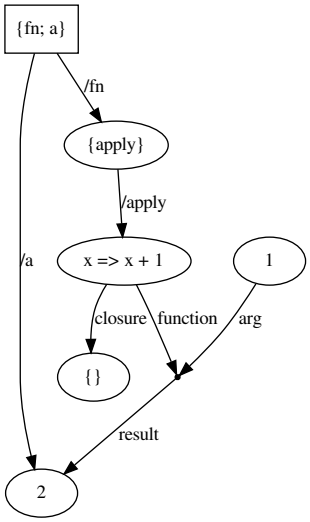


Figure 3.30: Document derived from listing 3.24

```
{
    fn = {
        apply = \\ x => x + 1
    }
    a = fn (1)
}
```

Listing 3.24: Demonstration of an object with a field called “apply” being callable

3.4.4 Passing an object expression to a function

Since practical code often involves calling a function with an object expression as an argument, there is a syntactic shortcut for that: you can omit function call parentheses when calling a function with an immediately defined object. Listing 3.25 shows this special syntax: the expression assigned to `a` shows the general but inconvenient way, whereas the expression assigned to `b` shows the fully equivalent but easier to type version.

It is intentional that when using this syntactic sugar TreeGen function applications begin to look a little like the tags from markup languages like XML. This parallel will become clearer in the context of type tags.

```
{
    fn = \\ x => x/bar

    a = fn ({
        bar = 1
    })
    b = fn {
        bar = 2
    }
}
```

Listing 3.25: A demonstration of the syntactic sugar for calling a function with an object expression

3.4.5 Recursion over graph loops can terminate

Due to the enforced uniqueness of function calls, it is possible generate a finite, cyclic result by recursing over an infinite loop in the document.

Listing 3.26 gives an example of this, whereby `foo` models an infinite list of alternating values `1` and `2`. Figure 3.31 shows that TreeGen's uniqueness guarantees `bar = fn (foo)` not only terminates but produces a correctly mapped cyclic list of infinitely alternating values `2` and `3`.

```
{
    // foo refers to a cyclic linked list-like structure: [1, 2, 1, ...]
    foo = {
        x = 1
        y = {
            x = 2
            y = foo
        }
    }
    fn = \\ f => {
        x = f/x + 1
        y = fn (f/y)
    }
    bar = fn (foo)
}
```

Listing 3.26: Demonstration of a terminating recursion over an infinite loop

This works because the result of a function call in progress is unknown, allowing recursions over loops to reference existing incomplete return values. Note that because the

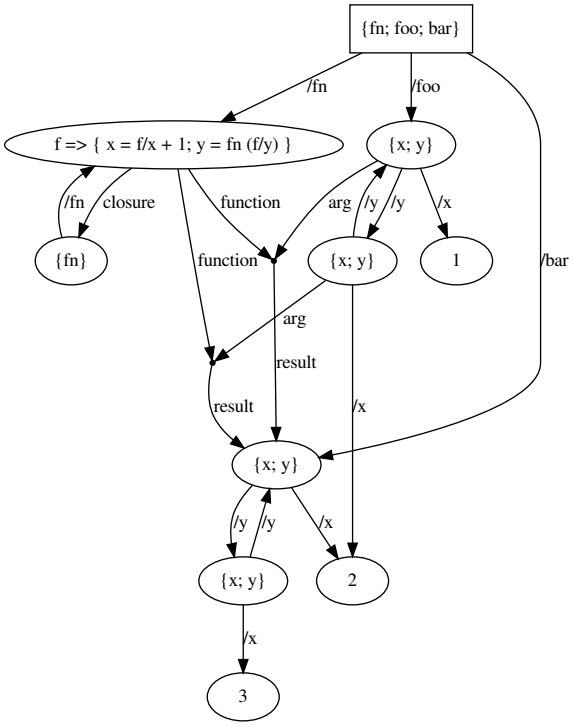


Figure 3.31: Document derived from listing 3.26

return value will be incomplete, this can only work if the result is usable in an incomplete state. While in this case it is fine to bind an unknown value to an object's field, a similar arrangement of primitive values will not do. Since, unlike object operations, primitive operations only resolve once their results are fully known, making a primitive operation rely on its own result will just lead to deadlock[3].

3.5 Type tags and type directives

TreeGen is a dynamically typed language, but for convenience it has a tagging system that allows it to emulate the discriminated unions and more advanced control flow constructs typically used by statically-typed languages like Haskell, ML, and Scala.

To say a value "is tagged with a type" means that the value is object-shaped with a field called `%type`. That field refers to the type tag itself, a kind of primitive-shaped value. This value allows runtime type checks to be explicitly performed by comparing type tags for equality. Furthermore, because, like other primitives, type tag shapes must be equal for two type tag values to merge successfully, the merge operation's rules make it an automatic error to attempt to merge two otherwise identical objects with different type tags, affording a reasonable degree of intrinsic separation among objects with different type tags.

Type tags themselves are not directly user visible, so we introduce them via the syntax the end-user would actually use. There are two directives for this, with an introductory example of each shown in listings 3.27 and 3.28 respectively.

```
{
    type tag1 = _
}
```

Listing 3.27: An introductory example of the type directive

```
{
    singleton tag2
}
```

Listing 3.28: An introductory example of the singleton directive

The more common form is the *type directive* as introduced by listing 3.27. The easiest way to explain this new syntax is via the observation that it is mostly syntactic sugar on top of some built-in functions. Listing 3.29 shows what an expansion of this type directive example might look like.

[3]This refers to the primitive operation starving itself of one or more of its own dependencies.

```
{
    tag1 = {
        let tag = type("tag1")
        apply = \\ v =>
            predef/object/apply_tag {
                src = v
                $tag = tag
            }
        unapply = \\ v =>
            if predef/object/has_field {
                obj = v
                name = "%type"
            } then {
                val = v
                check = v/%type == tag
            } else {
                val = v
                check = false
            }
    }
}
```

Listing 3.29: A desugaring of listing 3.27

`type tag1 = _` defines a new type tag alongside functions `tag1/apply` and `tag1/unapply`, which respectively allow `tag1` to be called like a function in order to create a tagged version of a given object `v`, and provide an operation that checks if an object `v` has that type tag. The extra `= _` part means any object that is tagged must satisfy the pattern `_`. Since patterns have not been discussed yet, the wildcard `_` is used to mean "accept everything", allowing the current explanation to avoid requiring an existing understanding of pattern semantics.

Even without patterns, this listing introduces several constructs:

- `type(...)` is a construct that takes a string value and makes it into a type tag value of the same name. The extra boilerplate and confusion likely to arise from using it directly means it is currently user-inaccessible. Rather, it represents the only part of type semantics that cannot be defined in terms of TreeGen itself and must therefore be included in core TreeGen.

- `predef/object/apply_tag` is a built-in function that takes the object `src` and creates a new object with all the same fields as `src`, but with the extra field `%type = tag`. This operation is a proxy for a more general form called `predef/object/updated`, but that

requires list syntax that has yet to be introduced so this simplified operation is given to explain the meaning while avoiding unnecessary complication.

- `predef/object/has_field` is a built-in predicate that checks if `obj` has a field corresponding to the string `name`. This allows `tag1/unapply` to gracefully give a negative answer when given either an object that has no type tag or a value that is not an object at all.

Aside from the underlying operations, it is also worth highlighting that the `unapply` function's peculiar return value is part of the *pattern protocol*. `check` is the boolean result one might expect, while `val` is only useful once one knows how patterns operate.

Less common but equally important is the *singleton directive*. In some cases one wants a type tag to be associated with one single value, for which we have the singleton directive as shown in listing 3.28.

Unlike type directives which must be revisited once patterns are explained, singleton directives are much simpler: all that is needed is the syntax `singleton tag2`, where `tag2` could be any valid TreeGen name.

```
{
    tag2 = {
        let tag = type("tag2")
        %type = tag
        unapply = \\ v => { val = v; check = v == tag2 }
    }
}
```

Listing 3.30: A desugaring of listing 3.28

Listing 3.30 shows how this might be desugared. Unlike the type directive that has a function for tagging objects, there is no `apply` function. Instead, the object `tag2` will be the only instance available and comes pre-tagged with the field `%type = tag`. Because there is only a single instance, the `unapply` function just checks for equality with that instance.

3.6 Pattern semantics

At a high level, patterns are a more expressive syntax for writing decisions and inspecting values. Patterns exist throughout the TreeGen language wherever names can be bound. The meanings of patterns, both general meanings and details particular to certain usages, shall be introduced in this section.

It is thanks to the simplest case for patterns that they could be glossed over up to this point. The simplest case for a pattern is a name, which requires nothing and binds a value to a single name. This is how all the assignment directives and let bindings so far have worked, and is a special case of *name patterns*. Aside from replacing the name with a *wildcard* `_`, allowing nothing at all to be bound, name patterns can be extended in one of two ways: *type requirements* and *sub-patterns*. Type requirements relate, among some other cases to be discussed shortly, to the `unapply` from type directives.

Listing 3.31 shows a type requirement being used as part of an assignment directive to assert that a singleton is its own instance. A wildcard is used to avoid generating any fresh name bindings.

```
{
    singleton s
    _: s = s // s must be an instance of s
}
```

Listing 3.31: A demonstration of a simple type requirement on a singleton

Listing 3.32 gives a desugaring of that requirement using an *assert directive*. Assert directives take a sub-expression and require that the value produced by that sub-expression's value eventually has the shape `true`. They are not common in idiomatic TreeGen code because most assertions are much more convenient to write as patterns. Aside from the new directive, we see the first example of how a type's `unapply` might be called.

```
{
    singleton s
    let anon1 = s
    assert (s/unapply (anon1))/check
}
```

Listing 3.32: A desugaring of the assignment directive in listing 3.31

Another kind of pattern that can be used is the *object pattern*. Object patterns look like objects, allowing the programmer to write the layout of the object value they expect in terms of name bindings and further sub-patterns rather than having to write out all the necessary conditionals and/or directives themselves. As a basic example, consider listing 3.33. It binds multiple fields of the object on the right-hand side of the assignment to multiple values, which can sometimes be a useful shortcut for writing code similar to the construct's desugaring into temporary variables and projections as given in listing 3.34.

```
{
    // imagine the right-hand side is a complex, opaque expression
    { a; b } = { a = 1; b = 2 }
```

```
}
```

Listing 3.33: An example of destructuring assignment

```
{
    let anon1 = { a = 1; b = 2 }
    a = anon1/a
    b = anon1/b
}
```

Listing 3.34: A desugaring of the pattern in listing 3.33

Notice how in this case the field name being accessed and the name it is bound to are the same. While it is often natural to just assume the two names are the same, sometimes the names need to be different. A *rebinding* can be used to change this if needed, as demonstrated in listing 3.35. Instead of just taking the name `c` from the object being accessed, the syntax `c->a` binds that value of field `c` to variable `a`, read as "rebind from `c` to `a`".

```
{
    { c->a; b } = { c = 1; b = 2}
}
```

Listing 3.35: An example of destructuring assignment with rebinding

Coming back to sub-patterns, listing 3.36 extends listing 3.33 to allow not only the binding of fields of the right-hand side to names but also the entire right-hand side as well. This is written `name := subpattern`, allowing a name pattern to operate on the entire value while also passing that value along to the sub-pattern for, in this case, field-specific patterns.

```
{
    obj := { a; b } = { a = 1; b = 2 }
}
```

Listing 3.36: A version of listing 3.33 that also binds the whole object

Sub-patterns can also be used inside object patterns, with a pattern like `{ x->_ := { a }; b}` being able to extract the same `a` and `b` bindings from a nested object `{ x = { a = 1 }; b = 2 }`. Notice that in this case the field `x` is rebound to the wildcard `_` in order to specifically avoid binding the value of that field to any name.

Aside from sub-patterns, object patterns can also express *default values*. Evaluating as if `predef/object/has_field` has been called, a default value checks if an object has the given field. If the field is present then the value is retrieved as normal. Otherwise, the default

expression is evaluated and the result is bound to that name instead. Listing 3.37 shows how this is written, with the semantics binding both `x` and `y` to the value `2`.

```
{
    { x ?= 2 } = {}
    { y ?= 3 } = { y = 2 }
}
```

Listing 3.37: A demonstration of default values in object patterns

For both name patterns and each part of an object pattern, all of the features presented so far have to be used together in a specific order. That order can be shown by writing a pattern that uses every feature at once: `{ name->rebind ?= default: typecheck := subpattern }`. When read left to right, the operations can be considered to happen "in that order". Note that this means the result of the default value will be subject to both the typecheck and the subpattern, which can sometimes matter if for instance the default value would fail the typecheck.

The last kind of pattern is called a *refinement pattern*. It can be included either as a suffix to an existing pattern or inside an object pattern, and allows checking arbitrary conditions involving the names bound by the pattern. It is written as `pattern if condition`, and an example of its usage is given in listing 3.38.

```
{
    // only succeeds if x has value 2
    { x if x == 2 } = { x = 2 }
}
```

Listing 3.38: A basic example of how to write a refinement pattern

The other way to write listing 3.38 would have been to put the condition outside of the block pattern instead, as in `{ x } if x == 2 = { x = 2 }`.

3.6.1 Let directives

When patterns are introduced, the concept of a let directive actually disappears. For maximum flexibility in the kinds of name binding available to the programmer, there is no real distinction between let directives and assignment directives. In an assignment directive, `let` is allowed as a prefix to any name in the pattern, indicating it should be bound without becoming a field of the object being defined. This allows code like that in listing 3.39, where the `a` binding becomes a field of the object being constructed and the `b` binding does not due to the `let` prefix.

```
{
    { a; let b } = { a = 1; b = 2 }
    c = b + 1
}
```

Listing 3.39: An example of selectively binding one name in a pattern as a field and another as just a local name using the `let` prefix

The let prefix also interacts with rebindings, where it becomes a prefix of the rebound name as in the pattern `{ a; c->let b }`.

`let` can also be used as a distributive prefix over an entire object pattern. Listing 3.40 compares having to individually prefix each part of an object pattern with the distributive option.

```
{
    foo = { a = 1; b = 2; c = 3 }

    // these two lines both bind a, b, and c as non-field locals
    // both lines are equivalent
    { let a; let b; let c } = foo
    let { a; b; c } = foo
}
```

Listing 3.40: An example illustrating the situational advantage of using the distributive version of the `let` prefix

3.6.2 Patterns in type directives

The pattern in a type directive is a way to specify the properties required of an object tagged with the type directive's type tag. These patterns are quite useful to look for when reading TreeGen code as they effectively make up a gradual, checked schema for parts of a TreeGen document.

For instance, a record type `rec` with fields `x` and `y`, that should both be integer values can be written as in listing 3.41. Notice `predef/int` is used as a type. There is a built-in `predef/int/unapply` that waits until the value it is given is known and then checks if it is a primitive integer.

```
{
    type rec = {
        x: predef/int
        y: predef/int
```

```
    }
    r1 = rec { x = 1; y = 2 }
}
```

Listing 3.41: An example defining a simple record type directive

Now that we have established patterns, type directives actually have an additional, optional piece of syntax. A type directive can have an *as-clause*, allowing the author of a type directive to include extra behaviour aside from the pattern. This is useful for implementing things like the filling of optional values, as the pattern itself cannot add fields to the object being checked. Default values in a pattern only affect what values are bound to a given name, not the underlying object(s). Listing 3.42 shows this syntax in action.

```
{
    type a_bad_idea = {x} as x

    result = a_bad_idea {
        // the as-clause will cause this inner
        // object to be tagged and returned
        tag_me = { foo = 1 }
    }
    // result = { %type = type(a_bad_idea); foo = 1 }
}
```

Listing 3.42: A simple, contrived example of the optional as-clause on a type directive

The idea is that instead of adding the type tag to the matched object, the expression on the right-hand side of `as` is evaluated with the pattern's bound names in scope and the result of that expression is used to generate the tagged object. This allows a the author of a type to alter the result of a type application if they need. Listing 3.42, however, is a contrived, simplified example that is not likely to be useful in practice. It may in fact be confusing, since to a user of `a_bad_idea` might normally expect the result of applying the type to "look like" the argument they provided.

Listing 3.43 is a more practical use of the as-clause, using the feature to make the application of `rec_opt` give its result's field `y` the default value `12`, if that field is missing from its input.

```
{
    type rec_opt = $rec := {
        x: predef/int
        y ?= 12: predef/int
    } as predef/object/updated {
        src = $rec
```

```
7           updated = [["y", y]]
8       }
9       r1 = rec_opt { x = 1; y = 2 }
10      r2 = rec_opt { x = 1 } // r2/y = 12
11  }
```

Listing 3.43: An practical example of the optional as-clause on a type directive

The operation `predef/object/updated` is used to create a new object-shaped value based on `$rec`, where the field with name equal to `"y"` is replaced by the value bound to the name `y`. By pattern semantics, `y` will be bound to either that field's actual value, should the field be present, or the default value `12`. The combined result is an object with all the fields present on the input object `$rec` except `y`, with that field bound to the defaulted value for `y` extracted by the type's pattern.

3.6.3 Partial functions with patterns

While it is impossible to notice if one never uses pattern- or match-based constructs, all functions in TreeGen have the built-in ability to behave like partial functions. Instead of just a simple return value, functions can be explained to return a pair of values `{ check = ...; val = ...}` where `check` is a boolean indicating whether the function accepted its argument and `val` is the return value if `check == true` or an unspecified value if `check == false`. This is hidden from the programmer and has no effect on programs that do not use patterns for control flow, but understanding what effect `check` can have on control flow can lead to very concise, robust code.

For instance, the function expression `\\ x if (2 div x) == 0 => x+1` gives a function value that only accepts even numbers (numbers that integer-divide 2 to 0) and returns their next odd number. The pattern syntax is the same pattern syntax as introduced before, binding `x` to be used in the body, should the pattern be successful. If the pattern fails then the body is not executed, with the return value becoming `{ check = false; val = ??? }`.

From the caller's perspective one might wonder why the `check` is returned rather than just asserted internally, since for the examples given so far that would be conceptually simpler. In a regular function call, there is not much difference – the caller has to additionally assert that `check == true`, but there is no substantial gain. The advantages come from the fact that functions can be used as an ad-hoc replacement for type directives when needed. If a function value (or an object with only an `apply` field) rather than an object with an `unapply` field is given in a pattern's type requirement, the resulting function application's

`check` is made part of the pattern's condition. This means that if there is an incidental type-like property that should not be a type tag – maybe the object being checked already has a different type tag – a function with a pattern that checks that property can be used as a type requirement.

This can be demonstrated using the example from listing 3.41: listing 3.44 shows a version of that example that uses functions instead of a type directive.

```
{
    rec_fn = \\ v := { x: predef/int; y: predef/inf } => v
    r1 = rec_fn { x = 1; y = 2 }
    r2: rec_fn = r1
}
```

Listing 3.44: A version of listing 3.41 using a function instead of a type directive

A more interesting revision of listing 3.41 might use functions for a form of metaprogramming: the function `meta_fn` represents a parameterised "type", a higher-order function that defines all records that have fields `x` and `y` of the same type. An example of this is given in listing 3.45, a listing that factors out the structural features of the `rec` definition. This doubles as a demonstration that type directives and partial functions are very much complementary mechanisms.

```
{
    meta_rec = \\ t => \\ v := { x: t; y: t } => v
    type rec = _: meta_rec (predef/int)
    r1 = rec { x = 1; y = 2 }
}
```

Listing 3.45: An example of using functions-as-types for metaprogramming

3.6.4 Match expressions

Match expressions, as they usually are in other languages, are a good way to lay out sets of cases. In TreeGen, match expressions will try each case from top to bottom in sequence, moving onto the next case if one of the cases fails. It is an error to run out of cases. Match cases are written `case pattern => body`, where if `pattern` succeeds then `body` is executed with the names bound by `pattern` in scope. Listing 3.46 gives a simple example of a match expression that distinguishes between two different singletons using type requirements.

```
{
    singleton a
    singleton b
```

```
    x = a
    y = x match {
        case _: a => 1
        case _: b => 2
    }
}
```

Listing 3.46: A simple example of a match expression

In addition to regular cases, match expressions also have a special *delegate case* syntax. This is in order to take advantage of partial functions: written `case * = partial_fn`, a delegate case "delegates" a portion of matching to that partial function. The function is called with the match expression's scrutinee, and if the function's `check` becomes true then the entire match expression yields the function's return value. If the function's `check` is false, then the next case is tried as normal. Listing 3.47 gives a simple example of the delegate syntax, resulting in `y` being bound to the value `12` due to the success of `fn`'s pattern.

```
{
    singleton a
    singleton b
    fn = \\ v: b => 12
    x = b

    y = x match {
        case _: a => 1
        case * = fn
    }
}
```

Listing 3.47: A simple example of a delegate case in a match expression

The other interaction that match expressions have with partial functions is the *match function* expression. It is very common to express a function as a set of cases, so it makes sense to allow a special syntax that fuses a function expression with a match expression. A match function starts with the `\\` of a regular function expression, but instead of having a pattern it has the keyword `match` which starts a match expression that uses the implicit function argument as a scrutinee. This interacts with partial function semantics by making the function's `check` true if one of the cases in the match expression succeeds and false if all the branches fail. This is different from just having a match expression nested inside a function expression, since in that situation all of the cases failing would just raise an error rather than making the function's `check` false.

An idiomatic use of this feature is shown in listing 3.48: use a match function to emulate a type union, giving a name to a group of individual types. `a_or_b` is a match function with two cases that each delegate to `a` and `b`'s `unapply` functions. That means that `a_or_b` will accept its argument if one of the two `unapply` functions accepts the same argument. If the argument is neither `a` nor `b` then `a_or_b` rejects.

```
{
    singleton a
    singleton b
    a_or_b = \\ match {
        case * = a/unapply
        case * = b/unapply
    }

    _: a_or_b = a
    _: a_or_b = b
}
```

Listing 3.48: An example of using a match function to express a type union

3.7 List definitions and syntax

Once patterns and type tags are introduced it becomes possible to discuss TreeGen's approach to lists. While TreeGen offers several pieces of syntactic sugar to help the programmer write concise code, lists are entirely expressible using plain TreeGen.

TreeGen's lists are linked lists, the same kind as found in languages like ML. They are an ADT with two cases: the *cons* case with a head and a tail, and the *nil* case which represents the empty list. We can write out TreeGen's list definitions in listing 3.49.

```
{
    list = {
        unapply = \\ match {
            case l: nil => l
            case l: cons := { tail: list } => l
        }
        singleton nil
        type cons = { head; tail }
    }
}
```

Listing 3.49: TreeGen's list definitions

These definitions are a subset of the standard library relating to lists – they are accessible as `predef/list`.

While the data definitions should be relatively self-explanatory, the `unapply` definition bears a little explanation. The definition allows patterns to express the requirement that a value be "a list", since using `list` itself as a type means the pattern will look up `list/unapply` and perform the checks listed there. The match function itself has two cases: first the base case which checks if we have reached `nil`, the end of the list. The recursive case is an example of a recursive pattern, checking first that the current value is a cons cell, then recursively checking that the tail matches the same `list` type we are defining. Notice that for this simple case we do not constrain the list elements, though it is quite possible to write a different checker that does so.

3.7.1 Syntactic sugar

Without syntactic sugar a list can be written, albeit inconveniently, as in listing 3.50.

```
{
    a_list = predef/list/cons {
        head = 1
        tail = predef/list/cons {
            head = 2
            tail = predef/list/nil
        }
    }
}
```

Listing 3.50: How to define a list without syntactic sugar

To make this multi-line operation a bit easier to write, TreeGen provides some syntactic sugar. The same list can be written most simply as `[1,2]`. When needing to prepend an element to a list there is also the `::` version common to ML-like languages, which allows the same list to be written very slightly less concisely as `1 :: 2 :: []`. All of these syntactic sugars expand to exactly the same code as listing 3.50.

As well as syntactic sugar for creating lists, it is equally useful to have syntactic sugar for matching lists in patterns. `list/unapply` in listing 3.49 is already a little convoluted due to having to explicitly destruct the list node using object syntax, but as shown in listing 3.51 trying to match a multi-element list is quite tedious.

```
{
    _: predef/list/cons {
```

```
        head->a
        tail: predef/list/cons := {
            head->b
            tail: predef/list/cons := {
                head->c
                tail: predef/list/nil
            }
        }
    } = [1,2,3]
}
```

Listing 3.51: How to pattern match a list without syntactic sugar

A more compact way to write the same code is via the syntactic sugar `[a,b,c] = [1,2,3]`. This syntax allows `a`, `b` and `c` to be completely arbitrary sub-patterns, while in assignment directives allowing the `let` prefix to distribute. For instance, both `let [a,b,c] = [1,2,3]` and `[a,let b,c] = [1,2,3]` are also valid patterns, with the first making none of the bound names part of the enclosing object and the second making only `a` and `c` part of the enclosing object.

Similarly to the expression-level sugar, for matching only the head of a list rather than an entire list, the `::` can be used. As such, another way to write the directive in listing 3.51 is `a :: b :: c :: [] = [1,2,3]`. The same rules apply regarding `let` placement and sub-patterns.

3.7.2 How to write a generic list-of type

```
{
    list = {
        of = \\ tpe =>
            \\ match {
                case [] => []
                case lst := (_: tpe) :: _: list/of (tpe) => lst
            }
    }
    _: list/of (predef/int) = [1,2,3]
}
```

Listing 3.52: The definition of `predef/list/of`

As a little exercise to put some of these definitions in context, listing 3.52 gives the definition of `predef/list/of`, a higher-order function that, given a type-like value `tpe` (function

or object with `unapply` field) will produce a partial function that can be used to check if a value is a list whose elements satisfy `tpe`.

There is a usage example on line 9, showing how to specify that a value should be a list of integer values.

3.8 A practical code generation example

Having introduced all the important intuitions behind the TreeGen language, it is now reasonable to discuss a detailed practical application. The application is a simplification of a common design pattern for which TreeGen is suited. The simplification is such that, while clearly many important concerns are skipped over in order to cut down in listing size, most important features of a real, practical application will be shown at least once.

As one might expect of a practical application, some standard library operations will be used in addition to the minimal set of operations introduced so far. Many of these operations are fully user-definable using normal TreeGen code, but are so common that they are kept in the auto-loaded standard library `predef`. Some special functions, however, are an interface to intrinsic operations built into the interpreter. The two intrinsics relevant here are a pair of idempotent operations relating to files:

- `predef/file/generate { path; contents }` is a function which upserts that a file exists at a particular `path`, with a particular `contents`. If this file does not exist, then it is generated. If it does exist, then the operation asserts that the required contents matches the actual contents, preventing any accidental conflicts whereby TreeGen code might provide multiple conflicting contents for the same file.

- `predef/import (path)` is a function that "imports" the TreeGen code stored in the file at the given `path`. This operation evaluates the stored TreeGen code, returning the resulting root object as the import function's result. Each unique path can only be imported once, with repeated imports evaluating to the same object. This restriction allows safe mutual imports, with recursive imports of the same file yielding the same under-construction root value rather than re-evaluating the code.

The design presented is laid out as follows, split into multiple files that import each other. Each listing corresponds to one of these files:

- `schema.tgen`: a schema detailing the structure of the input document using type definitions and partial functions.

- `definitions.tgen`: an input document, listing the necessary information for code generation in terms of the schema and any other necessary utilities. In this case, the example lists a small collection of abstract class and method information.
- `main.tgen`: a main file that imports the schema and input document, performs any necessary preprocessing, and provides the preprocessed input document to one or more code generation routines. This example will link to just one code generation routine for brevity, but in principle any number of generators could be run on the same preprocessed definitions, given a schema containing sufficient information for each generator to operate.
- `java_gen.tgen`: a basic, incomplete example of generating Java code, aiming to showcase the `formatting.tgen` text formatting library. Using this example, it should be possible to infer how one might perform arbitrarily sophisticated code generation tasks in the same mostly-declarative style.

3.8.1 Simple schema for classes and methods

The schema presented in listing 3.53 is a straightforward application of ideas from section 3.6.2. Many simplifications are made to keep the overall example compact, but the important ideas are there: one can model class definitions with names and a collection of methods; and type specifications have their own structure. Notice in particular that the type-like partial function `type_spec`, which is used to define type specifications in method and method argument specifications, allows cross-references to other `class` specifications to be used as valid type specifications in addition to the statically defined `int` type specification, ensuring that any such cross-references are valid by construction.

```
{
    types = {
        unapply = \\ match {
            case * = int/unapply
        }
        singleton int
    }

    type_spec = \\ match {
        case * = types/unapply
        case * = class/unapply
    }

    type class = {
```

```
        name: predef/string
        methods: predef/list/of (method)
    }
    type method = {
        name: predef/string
        arguments: predef/list/of (argument)
        return_type: type_spec
    }
    type argument = $argument := {
        name: predef/string
        _type: type_spec // the name "type" is a keyword
    }
}
```

Listing 3.53: The `schema.tgen` file, describing a simple, practical schema for specifying classes and methods

There are of course significant limitations to this schema, such as the assumption that the definitions using it operate in a flat namespace: there is no allowance for different class specifications being part of different namespaces, with the underlying assumption being that all specifications exist in one single common namespace.

If this schema were used in practice and a maintainer needed to add the namespace feature after the fact, however, TreeGen has features to deal with that kind of change. A type's pattern can contain default values, which, as in listing 3.43, would allow a maintainer to add a model for namespaces to class specifications without immediately needing to rewrite all of the specification code (`definitions.tgen`, which may be large). More sophisticated contextual logic could also be applied by varying the post-processing that takes place in `main.tgen`.

Another practically relevant addition that could similarly be introduced by adding optional parts to the type patterns would be documentation, both as a single block of text added to a method or class specification, and as specific, structured annotations attached to particular method argument specifications. This documentation could then be used by the code generators, allowing them to generate properly formatted documentation that matches the conventions and requirements peculiar to different programming languages.

3.8.2 Some sample definitions

Listing 3.54 describes the file `definitions.tgen`, an example of the previously-defined schema being used to specify some arbitrary interfaces. The listing describes two class

specifications named "Fooinator" and "Barinator", with the former having two methods `give_bar` and `something_else` specified, and the latter having a single method `give_foo` specified. Aside from showing what larger-scale usage of a schema looks like, there are a few important details that bear highlighting.

```
1  {
2      apply = \\ { meta; schema } => [
3          schema/class {
4              name = "Fooinator"
5              methods = [
6                  schema/method {
7                      name = "give_bar"
8                      arguments = []
9                      return_type = meta/Barinator
10                 },
11                 schema/method {
12                     name = "something_else"
13                     arguments = [
14                         schema/argument {
15                             name = "x"
16                             _type = schema/types/int
17                         },
18                     ]
19                     return_type = schema/types/int
20                 },
21             ]
22         },
23         schema/class {
24             name = "Barinator"
25             methods = [
26                 schema/method {
27                     name = "give_foo"
28                     arguments = []
29                     return_type = meta/Fooinator
30                 },
31             ]
32         },
33     ]
34 }
```

Listing 3.54: Some example definitions using the schema defined by listing 3.53

Instead of directly importing `schema.tgen`, this listing is callable (the `apply` field defined on line 2 makes the root object callable, as in listing 3.24), taking both the schema's root object and another object called `meta` as parameters. The rationale for this is that the

definitions should not concern themselves with exactly where the definitions they rely on come from. Unlike the other files discussed here, which would usually be stored in a well-known centralised location, these definitions might be stored across a source tree, next to the files to which they specify an interface. This would make it an error-prone, inconvenient process to ensure that any import paths remain correct as files are moved around over time, so in this kind of situation it can be useful to use a dependency injection-like style like this one instead.

Another interesting detail is what `meta` does: provided by the caller, in this case `main.tgen`, `meta` represents a holistic view of the namespace being defined. While in this example there is one file containing all definitions, it should not be hard to imagine there being multiple files, each with their own definitions. As a variation, `definitions.tgen` could just as well use imports to aggregate other definition files, acting as an index for the available `.tgen` files containing definitions. In either case, `meta` would have fields representing cross-references to any other interface, including those in different files. `meta`'s construction will be discussed in more detail alongside `main.tgen`.

3.8.3 Main file and out-of-order cross-references

Listing 3.55 describes the top-level control flow of a code generation task in TreeGen. Many parts are self-explanatory, being simple cases of importing one of the other files and sometimes calling one function, whereas lines 9-16 showcase the power afforded by TreeGen's support for out-of-order cross-referencing and computation.

There is only one code generation pass for Java code in this example, but it would be logically trivial to add broader language support by appending further imports and effect directives like those on lines 19-21.

```
1  {
2      schema = predef/import {
3          path = predef/path/relative ["schema.tgen"] }
4      definitions_fn = predef/import {
5          path = predef/path/relative ["definitions.tgen"] }
6
7      // cyclically generate definitions using meta and meta
8      // using definitions
9      definitions = definitions_fn {
10         $schema = schema
11         meta = predef/object/from_fields (predef/list/map {
12             list = definitions
13             fn = \\ class =>
```

```
                [class/name, class]
        })
    }

    // call into java_gen.tgen
    java_gen = predef/import {
        path = predef/path/relative ["java_gen.tgen"] }
    effect java_gen (definitions)
}
```

Listing 3.55: The `main.tgen` file, representing the top-level control flow and post-processing of a code generation task

Notice that `meta` is generated from the definitions themselves, using the operation `predef/object/from_fields`. The operation takes a list of name-value pairs and synthesizes an object value with those name-value pairs as fields. Using `predef/list/map`, the standard list mapping function, `meta` becomes an object with all of the defined class names as fields and all the defined class records as values. This is why `definitions.tgen` can use `meta/Fooinator` to refer to the class "Fooinator" as a type.

This whole operation takes full advantage of TreeGen's support for out of order references, allowing `meta` to be computed using the full functionality of the TreeGen programming language, despite the reference cycles inherent to what the code is doing. This flexibility also means that the processing given in `main.tgen` could be extended arbitrarily to match any additional design requirements, such as by constructing a version of `meta` that supports namespacing via nested objects, or by attaching additional metadata to the definitions, as required by the code generation task.

3.8.4 Generating Java code

The longest listing is listing 3.56, showcasing the kind of mostly-declarative text formatting library that can be built in TreeGen. Java is chosen as the output language because it is a well-known language with explicit types, requiring the generation of fully typed method definitions. The `formatting.tgen` library is omitted due to space concerns, but it is fully written in pure TreeGen and not that complex, taking up aroung 140 lines of code. The previously described `predef/file/generate { path; contents }` is used to output file contents.

The formatting library operates by mapping from nested lists of either strings or formatting information to one single, formatted string. For instance, `["a", "b"]` would be

rendered as `"ab"`, as would `[["a"], ["b"], []]`. The special formatting markers are `fmt/line_break`, which marks a new line (bound to `nl` in this listing, as a shorthand), and `fmt/indent { fmt }`, which specifies that the nested content should be indented. Additionally, a helper `fmt/separating { list; sep }` exists that operates similarly to a string join from other languages, placing instances of `sep` between each element of `list`.

Other more specialised markers exist, such as one for locally changing the indentation type, and a helper for declaratively generating fresh globally fresh names (useful for binding unique temporary variables), but the markers given above are sufficient to understand this example.

```
{
    schema = predef/import {
        path = predef/path/relative ["schema.tgen"] }
    fmt = predef/import {
        path = predef/path/relative ["formatting.tgen"] }

    // convenience shorthand
    let nl = fmt/line_break

    // how to render an abstract type to a Java type
    let render_type = \\ match {
        case _: schema/types/int => "int"
        case tpe: schema/types/class =>
            tpe/name // this would normally include namespace handling
    }

    // entry point
    apply = \\ definitions =>
        predef/list/foreach {
            list = definitions
            fn = \\ definition =>
                predef/file/generate {
                    path = predef/path/relative [
                        "out", "org", "example",
                        "${ definition/name }.java"]
                    contents = fmt/render {
                        default_indent = "    " // 4 spaces
                        $fmt = [
                            "// AUTO-GENERATED, DO NOT EDIT", nl,
                            "package org.example", nl,
                            nl,
                            "public class ", definition/name, " {",
                            fmt/indent {
                                $fmt = predef/list/map {
```

```
35                                      list = definition/methods
36                                      fn = \\ method => [
37                                          nl,
38                                          "public ",
39                                          render_type (method/return_type),
40                                          " ", method/name, "(",
41                                          fmt/separating {
42                                              sep = ", "
43                                              list = predef/list/map {
44                                                  list = method/arguments
45                                                  fn = \\ arg => [
46                                                      render_type (
47                                                          arg/_type),
48                                                      " ", arg/name
49                                                  ]
50                                              }
51                                          }, ") {",
52                                          fmt/indent {
53                                              $fmt = [nl, "// TODO"]
54                                          },
55                                          nl, "}"
56                                      ]
57                                  }
58                              },
59                              nl, "}",
60                              nl,
61                          ]
62                      }
63                  }
64          }
65  }
```

Listing 3.56: The `java_gen.tgen` file, showcasing how advanced text formatting can be achieved in TreeGen

Listing 3.57 illustrates what a file generated by `java_gen.tgen` would look like, complete with proper type cross-references (which require no qualification in Java as long as two classes are in the same package). Clearly this is not a complete set of API bindings, but what is there illustrates a system that can be extended to support arbitrarily complex output requirements.

Notice also that, as a generator grows, it is easy to factor its functionality out into definitions, as has happened to `render_type` on line 11. Other, more general, functionality could even be made into a library function – this is what happened to `fmt/separating`, for instance.

```
// AUTO-GENERATED, DO NOT EDIT
package org.example

public class Fooinator {
    public Barinator give_bar() {
        // TODO
    }
    public int something_else(int x) {
        // TODO
    }
}
```

Listing 3.57: The contents of the file `out/org/example/Fooinator.java`, as would be generated by `java_gen.tgen`

Counter-based fresh variable names

While it did not fit with the example given here, the formatting library's fresh variable generation feature is an interesting practical application of explicit unknown values.

In addition to handling whitespace, the formatting library allows for code that looks like this: `let i = ??? in [ fmt/unique_int { value = i }, "tmp_${ predef/int/to_string (i ) }" ]`. This pattern leverages unknown value semantics to allow the formatting engine to fill `i` with a value taken from an internal counter, ensuring that, were this code to appear twice, it would produce `tmp_1` the first time and `tmp_2` the second.

In other programming languages, this would either be an unnecessarily risky and hard to validate use of a mutable memory cell, or it would require significant changes to control flow in order for the formatting code to pass the counter value back to the code generator. In TreeGen, not only could this feature be added to the formatting library with minimal code changes, but runtime checks are in place by construction that ensure sound use of the feature, making it valid and reasonably safe for completely arbitrary formatting code to use the counter value, as long as it only affects formatting that appears "later" in the list(s) than the corresponding `fmt/unique_int` marker.

Of course, when doing this, some care needs to be taken to avoid a depencency cycle, such as making sure to not require a full typecheck of the formatting structure before using the formatting process to fill in `i`'s value, on which typechecking depends. This is because the would-be string based on `i` must be known before it can be typechecked, so if the code that would make `i` known requires this check before it can run, there is a circularity that TreeGen cannot navigate unassisted. Luckily, this can be easily worked around with a

small change to the formatting library, shifting the input typechecking from being a strict precondition to being an assertion that can finish at any time, including mid-way through the formatting process, once all the requested counter values are known.

Chapter 4

Formal semantics

Giving a formal model for TreeGen requires two general parts: first a formal model of the space containing the document values and topology, then the formal grammar, semantics and properties of core TreeGen.

4.1 Shapes and identifiers

Values, ignoring relationships between values for a moment, are formally given a shape and an identifier.

Each value has a unique identifier, allowing cross-references to be made using that identifier. Identifiers have no internal structure. They are just that, opaque identifiers that can be compared for equality. For convenience, identifiers can be notated as increasing natural numbers.

The shape is very similar to what is discussed in chapter 3, but with some simplifications that make formal discussion easier. In chapter 3, it was easier to give a set of relatively ad-hoc uniqueness rules for shapes, since the simplified formalism given here is difficult to represent and reason about graphically. Instead of the ad-hoc uniqueness rules, uniqueness is simplified as follows: for any shape that is not `???` (formalised as $\bot$ to reflect the parallel to lattice theory), there can only be one value (node in the graph) with that shape.

The simplification of shapes means we have to encode the uniqueness rules for each shape by embedding any information that dictates uniqueness into the shape itself.

- For primitive shapes this requires no change – the rule says nothing special about local topology so non-$\perp$ shapes in general having to be unique is a direct match to the primitive uniqueness rule.
- For object shapes, there are requirements on local topology: for every object shape that has the same set of field names, no two values can have the same set of field values. To express this in terms of overall shape uniqueness, we can add the identifiers for the field values to the object shapes. This would require that any mapping of field names to field values in an object shape be unique, which is equivalent to the original requirement.
- For function shapes, for every function shape with the same syntactic definition no two values can have the same closure object. This can be expressed via the same logic as object uniqueness by including the closure object's identifier in the function shape.
- Call and field uniqueness will not be encoded into shapes – they will be taken care of as we introduce the overall structure of an environment in the next section.

The formal set of shapes including all of the details described above can be described using this context-free grammar:

⟨*shape*⟩ ::= $\perp$
| '`prim`' ⟨*primitive*⟩
| '`object`' (⟨*name*⟩ $\mapsto$ ⟨*identifier*⟩)*
| '`function`' ⟨*function-expression*⟩ '`;`' ⟨*identifier*⟩

As mentioned before, the $\perp$ shape corresponds to the informal `???` shape.

⟨*primitive*⟩ ::= '`true`' | '`false`'
| ⟨*int*⟩
| ⟨*string*⟩
| '`type`' ⟨*string*⟩

Primitive value shapes all start with `prim`, and consist of all the kinds of primitive values available to the language. As shown below, this consists of booleans, integers, strings and type tags. These all match the informal descriptions exactly.

Object and function shapes are prefixed with "object" and "function" respectively, one encoded as the previously discussed mapping from field names to identifiers and the other encoded as a semicolon-separated pair of function expression (corresponding to the formal grammar for function expressions) and the closure object's identifier.

4.2 Environments

An *environment* in TreeGen gives a formal, structured view of the graphical notation used in chapter 3. An environment represents a snapshot of the entire graph structure of a valid document as it is built up. Environments are like a structured version of heap memory from imperative languages. The difference is that only monotonic mutability is allowed, making typical "set variable, act on variable value, change variable to be inconsistent with action" mutability bugs impossible. Environments do not represent error states – if an error occurs the resulting environment is undefined. Environments can be viewed as global state when imagining program evaluation, but are presented formally here as a 5-tuple that is threaded throughout the semantics.

Aside from their intrinsic structure, environments are comparable, both by $N \cong N'$ (*congruence*) as a more flexible form of equality and with $N \sqsubseteq N'$ (*sequencing*), a relation between environments that defines valid sequences of environments. $N \sqsubseteq N'$ forms a non-strict partial order when $N \cong N'$ is used as an equality relation. Both of these relations will be discussed once the environment structure has been fully introduced.

The 5-tuple that makes up an environment is $N = (S, F, C, C_f, R)$. Each part contributes a different feature and set of invariants to the environment N. A particular quirk to watch out for is that some of each part's structural invariants will be transiently violated by the merge operation. These violations are called *uniqueness violations* and apply to properties defined by defs. 4.2.2 to 4.2.5.

4.2.1 Shape relation

$S \subseteq \mathbb{P}(\mathit{identifiers} \times \mathit{shapes})$, meaning that S is a set of pairs of identifiers and shapes. Often, but not always, this set of pairs represents two functions described by defs. 4.2.1 and 4.2.2. S^1 gives a mapping from an identifier to its shape, while S^{-1} defines the previously mentioned shape uniqueness condition: all shapes except $\bot$ must correspond to exactly one identifier.

$$S^1 : \mathit{identifiers} \to \mathit{shapes} \tag{4.2.1}$$

$$S^{-1} : (\mathit{shapes} \setminus \{\bot\}) \to \mathit{identifiers} \tag{4.2.2}$$

The interpretation in def. 4.2.1 must always be valid, whereas the interpretation in def. 4.2.2 may be transiently invalid during the execution of a merge operation.

4.2.2 Field relation

$F \subseteq \mathbb{P}(\mathit{identifiers} \times \mathit{names} \times \mathit{identifiers})$ describes the field relation, as a set of triples relating value identifiers, field names and field value identifiers. Outside of merge operations, F can be interpreted as in def. 4.2.3, mapping pairs of value identifier and field name to unique field value identifiers.

$$F^1 : \mathit{identifiers} \times \mathit{names} \to \mathit{identifiers} \tag{4.2.3}$$

Like the shape invariant, the interpretation in def. 4.2.3 may also be transiently invalid during a merge operation.

4.2.3 Calls relation

$C \subseteq \mathbb{P}(\mathit{identifiers} \times \mathit{identifiers} \times \mathit{identifiers})$ is a set of triples of identifiers relating function argument identifiers, function identifiers and function call result identifiers respectively. This corresponds to the calls relation. Outside of merge operations, C can be interpreted as in def. 4.2.4, a function relating each pair of function argument identifier and function value identifier to a unique result value identifier. This is the formal representation of the call uniqueness rule.

$$C^1 : \mathit{identifiers} \times \mathit{identifiers} \to \mathit{identifiers} \tag{4.2.4}$$

As with defs. 4.2.2 and 4.2.3, def. 4.2.4 may also be transiently invalid during a merge operation.

4.2.4 Call fingerprints

$C_f \subseteq \mathit{identifiers} \times \mathit{identifiers} \times (1 \ldots k \in \mathbb{N} \to \mathit{identifiers})$ represents the concept of *call fingerprinting*. The relation mirrors C, mapping pairs of function argument identifier i_a and function identifier i_f to what can be called a call fingerprint: an arbitrary-arity tuple of identifiers listing the identifier of each fresh value created by executing the function body associated with the function call (i_a, i_f). It is almost irrelevant in practice, but very important formally.

```
{
    fn = \\ x => let y = ??? in x
    a = ???
    b = ???
    foo = fn (a)
    bar = fn (b)
    effect a = b
}
```

Listing 4.1: An example of a program where call fingerprinting matters

Consider listing 4.1: logically, we know that when the program has completed, `fn` should have only one instance of the calls relation, but evaluating the listing top to bottom, `fn (a)` and `fn (b)` both execute to completion before the directive `effect a = b` merges both the arguments `a` and `b` as well as the results of `fn (a)` and `fn (b)`.

This leaves us with a question: given that `???` creates a fresh, distinct value and `y = ???` is not referenced by `fn`'s return value, are there one or two redundant unknown values added to the environment, bound to `y` then never referenced again? Without C_f the answer depends on execution order.

C_f resolves this as follows: since the value(s) bound to `y` are created when the body of the function `fn` is executed, those values must be part of this fingerprint we have defined. If we require the merge operation to merge fingerprints when it merges function calls, then merging the two calls relation instances would also merge the respective `y = ???`. With this, it becomes possible to prove that in any scenario similar to that in listing 4.1, the final environment will contain exactly one result of `y = ???`, regardless of execution order.

Note that for the same reasons as with C, outside of merge operations, C_f can be interpreted as the function defined by def. 4.2.5.

$$C_{f^1} : \mathit{identifiers} \times \mathit{identifiers} \to (1 \dots k \in \mathbb{N} \to \mathit{identifiers}) \tag{4.2.5}$$

4.2.5 Rewrite history

$R : \mathit{identifiers} \nrightarrow \mathit{identifiers}$ is a partial function mapping some arbitrary set of identifiers to identifiers that can be indirectly considered "in" the environment, following the indirection $R^* : \mathit{identifiers} \to \mathit{identifiers}$ defined by def. 4.2.6. R maps from any identifier to either an identifier in $\mathit{domain}(S^1)$ or $\mathit{domain}(R)$ where each pair represents a single substitution, while R^* maps directly from any identifier to a final substituted identifier that must be in $\mathit{domain}(S^1)$.

(a) Environment N (b) Environment N'

Figure 4.1: An illustration of two environments N and N' that cannot structurally be ordered relative to one another

$$R^*(i) = \text{if } i \in \mathit{domain}(R) \text{ then } R^*(R(i)) \text{ else } i \tag{4.2.6}$$

The rewrite history is not directly meaningful to the environment's structure, but it plays an important role in defining valid environment sequences. Aside from R, environments have no notion of "direction" – there are several cases where structurally one environment N can be followed by a different environment N', but N' could just as well be followed by N. Consider N and N' as pictured respectively in figs. 4.1a and 4.1b. One could imagine a merge operation that maps N to N', but one could also imagine an operation that goes from N' to N by adding a fresh $\perp$-shaped value. R solves this by requiring that a record be kept of the merge operation going from N to N'. Once we know that record is in R, we can no longer claim that N can be reached from N' since any fresh value involved in such a step must not be in $\mathit{domain}(R)$ by the classical definition of "freshness".

4.2.6 Environment accessor notation

In order to make the very common operation of accessing parts of an environment easier to read and write, some convenience subscript functions will be used. Given an environment N, N_S, N_F, N_C, N_{C_f}, and N_R can be used to reference each of N's elements.

Particular interpretations of a given part will be accessed via the same subscript notation. For example, $N_{S^{-1}}$ refers to the interpretation of element S as defined in def. 4.2.2, a function from all shapes except $\perp$ to that shape's unique identifier.

4.2.7 Shape invariants

So far the overall structure of an environment has been given without discussing the relationship between the shape associated with an identifier and the relationships in which an

identifier is allowed to take part. For instance, as discussed in chapter 3, a primitive-shaped value should not have fields. These rules are *shape invariants* and will be presented here as rules applying without loss of generalisation to some arbitrary environment N.

$\perp$ shapes

The $\perp$ shape supports all operations, so it has no further invariants. All structurally possible configurations involving $\perp$-shaped values are valid.

Primitive shapes

Primitive shapes are not allowed to have fields, nor are they allowed to be called as functions. The corresponding invariant is def. 4.2.7, which asserts that in a valid environment no identifiers associated with a primitive shape may appear on the left-hand side of the N_F relation or in the function position of the N_F relation.

$$\forall (i, \text{prim } p) \in N_S,\ \nexists (i, n, i_f) \in N_F \wedge \nexists (i_a, i, i_r) \in N_C \tag{4.2.7}$$

Object shapes

Object-shaped values have multiple properties which are best discussed one at a time.

First, for some identifier i the fields in the object shape must correspond exactly to the fields i is given by N_F. This is formalised by def. 4.2.8 which requires that for all identifiers i with an object shape there be a biconditional relation between the existence of a field in O and the corresponding record in N_F.

$$\forall (i, \text{object } O) \in N_S,\ (n, i_f) \in O \iff \exists (i, n, i_f) \in N_F \tag{4.2.8}$$

The second property has to do with calls: formally an object-shaped value may not be called, as shown in def. 4.2.9.

$$\forall (i, \text{object } O) \in N_S,\ \nexists (i_a, i, i_r) \in N_C \tag{4.2.9}$$

Note that there is an exception in practice for object-shaped values that have a field called "apply". This can be emulated via the syntactic rewriting given in chapter 5 so we do not consider it here.

Function shapes

Function-shaped values can be called in any way but cannot have fields. Of course, technically, some function calls are not valid, but this is impossible to determine without looking at the evaluation semantics, so it is not modeled at the level of valid environment. Definition 4.2.10 shows this requirement, mirroring the non-callable part of the invariant for primitive values.

$$\forall(i, \text{function } f; i_c) \in N_S,\ \nexists(i, n, i_f) \in N_F \tag{4.2.10}$$

In a similar spirit to not detecting valid function calls, notice that this invariant does not concern itself with i_c pointing to an appropriate closure object. Informally, i_c must eventually be object-shaped and have a set of fields that exactly matches the free variables in the expression f, but this is more relevant to evaluation semantics so it is not formally stated here.

4.3 Environment congruence

Environments encode a lot of information, but specifically which identifier is which is never important. What is important is, regardless of identifiers, that we have the same set of shaped values, the same relationships between them, and that we have an equivalent history of substitutions. Aside from the substitution history this is the same as the informal graphical notation from chapter 3 – it doesn't matter which identifier is which, just that there are the functionally the same values with the same shapes and topology.

We could in fact express a more permissive equivalence relation where only values, shapes and relationships matter without considering substitution history, but formally it is most useful to introduce the relation $N \cong N'$, as defined in eq. (4.3.1).

$$\begin{aligned} &N \cong N' = \\ &\quad \exists R_s : \mathit{domain}(N_S) \cup \mathit{domain}(N_R) \to \mathit{domain}(N'_S) \cup \mathit{domain}(N'_R), \\ &\qquad i \neq i' \implies R_s(i) \neq R_s(i') \wedge [i \to R_s(i)]N = N' \end{aligned} \tag{4.3.1}$$

We say that two environments are *congruent* if there exists a bijective substitution function R_s that, when used to perform the substitution $[i \to R(i)]N$ mapping all identifiers

in N i to an equivalent identifier in N' $R(i)$, gives an environment syntactically equal to the a corresponding environment N'

This implements the intuition that "identifiers don't matter", or approximately alpha-renaming, since if two environments are structurally identical but different by some set of identifiers, then a set of substitutions R_s must exist that makes them exactly equal. For environments that actually are syntactically equal, R happens to be the identity function.

4.4 Environment sequencing

In order to reason about how environments can evolve during the execution of a TreeGen program, we need to understand what sequences of environments should be possible. This section defines the relation over environments $N \sqsubseteq N'$ which can be used to answer this question. The relation $N \sqsubseteq N'$ will be shown to be a non-strict partial ordering over environments when using the $N \cong N'$ relation as equality condition.

$$\frac{\begin{array}{c}\exists R_s : \mathit{domain}(N_S) \cup \mathit{domain}(N_R) \to \mathit{domain}(N'_{S^1}) \cup \mathit{domain}(N'_R) \\ \forall (i, i'), i \neq i' \implies R_s(i) \neq R_s(i') \qquad [i \to R_s(i)]N_R \subseteq N'_R \\ \forall i \in \mathit{domain}(N_{S^1}),\ (N, N'); N'_{R^*} \circ R_s \vdash i \sqsubseteq_i N'_{R^*}(R_s(i))\end{array}}{N \sqsubseteq N'} \text{ (ENV-LEQ)}$$

$N \sqsubseteq N'$ itself is defined by one single rule requiring two properties, given some function over identifiers R_s.

Similar to the definition of congruence, there must exist some function R_s that translates each identifier in $\mathit{domain}(N_S) \cup \mathit{domain}(N_R)$ to a corresponding identifier in $\mathit{domain}(N'_S) \cup \mathit{domain}(N'_R)$. Unlike in congruence however, R_s need only be injective. Additional identifiers may be present in $\mathit{domain}(N'_{S^1})$ or $\mathit{domain}(N'_R)$ that have no equivalent at all in $\mathit{domain}(N_{S^1})$ or $\mathit{domain}(N_R)$. This allows N' to freely gain fresh identifiers, while making $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ a universal minimum environment.

The first property is that under the substitutions R_s, N_R must effectively be a subset of N'_R – the set of substitutions may only grow from N to N'.

The second property is the most involved, requiring that for each identifier in $i \in \mathit{domain}(N_{S^1})$, that is, each non-substituted identifier in N, there must be a corresponding greater or equal identifier $N'_{R^*}(R_s(i))$ that is greater than or equal to it according to subordinate relation $(N, N'); R \vdash i \sqsubseteq_i N'_{R^*}(R_s(i))$. Aside from the specific definition of the subordinate *identifier sequencing* relation, the usage of N'_{R^*} composed with R_s has

an important meaning. In addition to the injective substitution of identifiers R_s, we also follow the substitutions in N'_R which need not be injective: a substitution in N'_R can map multiple identifiers in N to the same corresponding identifier in N', at the cost of needing to be explicitly listed in N'_R. This would be how, for example, merge operations work: two originally distinct values with different identifiers (i, i') would appear in N'_R as a mapping from i to i', forcing both i and i' in N to correspond to the single identifier i' in N'.

$(N, N'); R \vdash i \sqsubseteq_i i'$ defines the sequencing of pairs of identifiers i and i' and will be defined shortly, but first it's important how the identifiers of shapes being compared are related.

4.4.1 Shape sequencing

For each identifier, the local property that sequencing relies on is *shape sequencing*. The relation $R \vdash s \sqsubseteq_s s'$ relates the pair of shapes s and s', defining the valid shape sequences. Since shapes also contain identifiers, R is referenced to ensure that the contained identifiers are consistent with the overall mapping between identifiers.

The first rule is that the $\bot$ shape can be followed by any shape.

$$R \vdash \bot \sqsubseteq_s s \qquad \text{(SHAPE-LEQ-BOT)}$$

Equally simply, primitive shapes can only be followed by themselves: once a value has a primitive shape, it cannot be changed.

$$R \vdash \text{prim } p \sqsubseteq_s \text{prim } p \qquad \text{(SHAPE-LEQ-PRIM)}$$

Object shapes can be followed by any object shape that has the same field names and a related set of field value identifiers. Further requirements on the field values, like the shapes of the field values forming a valid sequence, are enforced by the fact that all identifiers must independently follow these rules via a consistent R.

$$\frac{\mathit{domain}(O) = \mathit{domain}(O') \qquad \forall (n, i_f) \in O, R(i_f) = O'(n)}{R \vdash \text{object } O \sqsubseteq_s \text{object } O'} \text{(SHAPE-LEQ-OBJ)}$$

Function shape rules are a mix between the primitive shape rule and the object shape rule: the function ASTs f must be equal, and the closure objects i_c and i'_c must be related by R.

$$\frac{R(i_c) = i'_c}{R \vdash \text{function } f; i_c \sqsubseteq_s \text{function } f; i'_c} \qquad \text{(SHAPE-LEQ-FN)}$$

4.4.2 Identifier sequencing

Now that individual shape sequencing has been defined, what remains is identifier sequencing. As elsewhere, the main intuition being that i' has to have an "equivalent or bigger" topology when compared to i.

$$\frac{\begin{array}{c} R \vdash N_{S^1}(i) \sqsubseteq_s N'_{S^1}(i') \\ \forall (i, n, i_f) \in N_F,\ (R(i), n, R(i_f)) \in N'_F \\ \forall (i_a, i, i_r) \in N_C,\ (R(i_a), R(i), R(i_r)) \in N'_C \\ \forall (i_a, i, I_f) \in N_{C_f},\ (R(i_a), R(i), (k \in \mathit{domain}(I_f), R(I_f(k))) \in N'_{C_f} \end{array}}{(N, N'); R \vdash i \sqsubseteq_i i'} \quad \text{(I-LEQ)}$$

Each point on this list matches one line of the (I-LEQ) rule:

- The shapes s and s' of i and i' must form a valid sequence according to $R \vdash s \sqsubseteq_s s'$, as previously defined.
- i' must have at least the same set of fields as i, and all of the field value identifiers must be related by R. i' may have additional fields which are not explicitly constrained – $\{(n, i_f) \mid (i', n, i_f) \in N'_F\}$ may grow or stay the same, but it may never shrink.
- i' must have at least as many instances of the calls relation as i that feature it in the function position where both the argument and result identifiers are related by R. i' may be called in additional ways that are, like additional fields, not explicitly constrained. Likewise, the set of calls relations involving i' may grow but never shrink.
- In the same way as C, all call fingerprints in C_f must have an equivalent in the subsequent environment, given substitution by R.

4.4.3 Environment sequencing defines a partial order

Having given all of the component rules that make up environment sequencing, we can show that it defines a partial order when equality is defined as environment congruence.

A relation forms a partial order if we can prove reflexivity, antisymmetry and transitivity.

Reflexivity

$$N \cong N' \implies N \sqsubseteq N' \tag{4.4.1}$$

To show reflexivity we have to show that for any pair of congruent environments N and N', $N \sqsubseteq N'$. We know from the congruence definition that both environments have to be structurally identical, so we can safely assume that all identifiers in N have an equivalent in N' with the same shape and relations. We also know that every substitution in N_R has an equivalent in N'_R, and that no additional substitutions can be in N'_R.

Without loss of generality, we select a function R_s such that $N \cong N'$ holds, as well as an identifier $i \in \mathit{domain}(R_s)$. While the conditions for R_s under congruence and sequencing are different, those for congruence are strictly stronger than those for sequencing (R_s being bijective as opposed to just being injective), so it is safe to move an R_s from congruence to sequencing. Since we know that no additional substitutions can be in N'_R and every substitution in N_R has an equivalent in N'_R, trivially $[i \to R_s(i)]N_R \subseteq N'_R$ must hold, because $[i \to R_s(i)]N_R = N'_R$. We aim to show that $(N, N'); R \vdash i \sqsubseteq_i R(i)$, where effectively $R = R_s$ since N'_R cannot contain any substitutions affecting identifiers in $\mathit{domain}(N'_{S^1})$.

First we think locally and aim to show that $R \vdash N_S(i) \sqsubseteq_s N'_S(R(i))$. Consider each possible kind of shape to which $N_S(i)$ can refer:

- $\bot$ shapes form a valid sequence thanks to (SHAPE-LEQ-BOT). $\bot$ can be followed by any shape, including $N'_S(R(i)) = \bot$.
- Primitive shapes form a valid sequence thanks to (SHAPE-LEQ-PRIM). The rule requires primitive shapes following each other to be exactly equal, and since primitive shapes do not contain identifiers we know that under any substitutions $N_S(i) = N'_S(R(i))$ must be true.
- Object shape sequencing follows the rule (SHAPE-LEQ-OBJ), requiring that the set of field names in each shape be exactly equal and that all fields correspond to each other through R.

  Field names do not contain identifiers so under any set of substitutions we know that $\mathit{domain}(O) = \mathit{domain}(O')$ must hold.

  For the second condition, correspondence by R of field value identifiers, we can use the fact that the rule $N \cong N'$ selects an R_s that makes $N = [i \to R(i)]N'$. This means that object $O = [i \to R(i)](\text{object } O')$ must hold for the overall environments to be syntactically equal under substitution by R, implying that R must contain a

substitution mapping each identifier in O to an identifier in O', satisfying the second condition.

- By (SHAPE-LEQ-FN), function shapes form a valid sequence if they share the same syntax tree and their closure object identifiers are related by R.

  For the first condition, since the syntax trees do not contain identifiers $N \cong N'$ requires them to be syntactically equal.

  For the second condition an equivalent argument to that for object shape identifiers can be used. Since the shapes have to be syntactically equal under substitution by R it means that R must contain a mapping from i_c to i'_c, satisfying the condition $R(i_c) = i'_c$.

As for the topological rules implied by $(N, N'); R \vdash i \sqsubseteq_i R(i)$, we can use a similar reasoning as for the identifiers embedded inside shapes. Since from congruence N and N' have to be syntactically equal aside from a bijective mapping $R_s = R$ over identifiers, we can reason that for the fields, calls and fingerprint conditions there must exist records in N'_F, N'_C and N'_{C_f} that under substitution of identifiers by R correspond directly to each record in N_F, N_C and N_{C_f}. This fact should be sufficient to claim that any of the specific subsets of F, C and C_f related by each individual instance of $(N, N'); R \vdash i \subseteq_i R(i)$ must be syntactically equal under substitution by R.

Antisymmetry

$$N \sqsubseteq N' \wedge N' \sqsubseteq N \implies N \cong N' \tag{4.4.2}$$

To show antisymmetry, we can begin by showing that given the two injective functions R_s and R'_s required by $N \sqsubseteq N'$ and $N' \sqsubseteq N$ respectively, both functions must be bijective. From the definition of (ENV-LEQ) we know that $R_s : \mathit{domain}(N_{S^1}) \cup \mathit{domain}(N_R) \to \mathit{domain}(N'_{S^1}) \cup \mathit{domain}(N'_R)$ and $R'_s : \mathit{domain}(N_{S^1}) \cup \mathit{domain}(N_R) \to \mathit{domain}(N'_{S^1}) \cup \mathit{domain}(N'_R)$, which means that $\mathit{domain}(R_s) = \mathit{range}(R'_s)$ and $\mathit{range}(R_s) = \mathit{domain}(R'_s)$. Since both R_s and R'_s must be injective and we independently know that their domains and ranges must be equal, then by the properties of injectivity R_s and R'_s must also be bijective.

Now that we know R_s and R'_s must be bijective, either mapping also satisfies the conditions for the congruence relation. We choose R_s and consider R'_s to be its inverse.

From the second constraint of (ENV-LEQ) on N_R and N'_R we can show that $[i \to R_s(i)]N_R = N'_R$. We know that $[i \to R_s(i)]N_R \subseteq N'_R$ and $[i \to R'_s(i)]N'_R \subseteq N_R$. Having

chosen R'_s as the inverse of R_s, we can rearrange the second term to $[i \to R_s(i)]N'_R \subseteq N_R$. Then, from antisymmetry of the subset relation, we know that $[i \to R_s(i)]N_R = N'_R$.

We must now use this to show that from the third constraint in (ENV-LEQ) all the other environment components are also related by $[i \to R_s(i)]N = N'$. To do this we first notice that since $[i \to R_s(i)]N_R = N'_R$, N_{R^*} can have no effect on the R in shape or identifier sequencing, since the only substitutions that can have an effect must be in N'_R and not N_R. Then without loss of generality we choose $i \in \mathit{domain}(N_{S^1})$ and $i' \in \mathit{domain}(N_{S^1})$ such that $R_s(i) = i'$, and consider both $(N, N'); R \vdash i \sqsubseteq_i i'$ and $(N', N); R^{-1} \vdash i' \sqsubseteq_i i$. Note that R^{-1} is derived from R'_s being inverse of R_s and $[i \to R_s(i)]N_R = N'_R$.

For the first case of identifier sequencing we consider without loss of generality the cases for shape sequencing of s and s'. Given $R \vdash s \sqsubseteq_s s' \wedge R^{-1} \vdash s' \sqsubseteq s$, we show that $[i \to R(i)]s = s'$ must hold:

- If $s = \bot$, then the relevant shape sequencing rule is (SHAPE-LEQ-BOT). Swapping s and s', the only rule that accepts $\bot$ on the right-hand side is also (SHAPE-LEQ-BOT). This means that $s' = \bot$, so in this case $[i \to R(i)]s = s'$ must be true.
- If $s = \text{prim } p$, then by definition of (SHAPE-LEQ-PRIM), the only valid s' is prim p. In this case as well, $[i \to R(i)]s = s'$ must be true.
- If $s = \text{object } O$, then by (SHAPE-LEQ-OBJ), the only valid $s' = \text{object } O'$. By definition, the domains of O and O' must be equal, and their ranges must correspond via R, which are the two conditions needed to satisfy $[i \to R(i)]s = s'$ in this case.
- If $s = \text{function } f; i_c$, then by (SHAPE-LEQ-FN), the only valid $s' = \text{function } f; i'_c$. By definition $R(i_c) = i'_c$, which again is the condition required to satisfy $[i \to R(i)]s = s'$ in this case.

Since we have shown that $R \vdash s \sqsubseteq_s s' \wedge R^{-1} \vdash s' \sqsubseteq s \implies [i \to R(i)]s = s'$, and from the first condition of (I-LEQ), we know $R \vdash N_{S^1}(i) \sqsubseteq_s N'_{S^1}(i') \wedge R^{-1} \vdash N'_{S^1}(i') \sqsubseteq_s N_{S^1}(i)$, then we know that $[i \to R(i)]N_{S^1}(i) = N'_{S^1}(i')$. As i and i' are respectively any $i \in \mathit{domain}(N_{S^1})$ and any $i' \in \mathit{domain}(N'_{S^1})$ then we know that $[i \to R(i)]N_S = N'_S$.

F, C, and C_f, share very similar arguments. For the fields relation, we know that $\forall(i, n, i_f) \in N_F, (R(i), n, R(i_f)) \in N'_F$ and $\forall(i', n, i_f) \in N'_F, (R^{-1}(i'), n, R^{-1}(i_f)) \in N_F$. Since i and i' are chosen without loss of generality from each environment, we can argue that these two statements can be rewritten to the equivalent subset notation, $[i \to R(i)]N_F \subseteq N'_F$ and $[i \to R^{-1}(i)]N'_F \subseteq N_F$. Now from the same argument as N_R and antisymmetry of

the subset relation we know $[i \to R(i)]N_F = N'_F$, which by our previous claim that N_{R^*} can have no effect on R must be equivalent to $[i \to R_s(i)]N_F = N'_F$.

Logically equivalent arguments can be made for N_C and N_{C_f}, at which point we have shown that every individual component of N and N' must be equal under substitution via R.

Therefore, $N \cong N'$ must hold.

Transitivity

$$N \sqsubseteq N' \wedge N' \sqsubseteq N'' \implies N \sqsubseteq N'' \tag{4.4.3}$$

We can show this holds by showing that, for the R_s that must exist for $N \sqsubseteq N'$ to hold and the R'_s that must exist for $N' \sqsubseteq N''$ to hold, $R''_s = R'_s \circ R_s$ is a valid mapping under which $N \sqsubseteq N''$ holds.

First we can show that $[i \to R''_s(i)]N_R \subseteq N''_R$. From (ENV-LEQ) we know that $[i \to R_s(i)]N_R \subseteq N'_R$ and $[i \to R'_s(i)]N'_R \subseteq N''_R$. Since R''_s is the composition of R_s and R'_s and each of R_s, R'_s and R''_s are injective, we know that $[i \to R''_s(i)]N_R$ must be in terms of the same identifiers as N''_R since R''_s must map all identifiers in N_R to some subset of the identifiers in N''_R. Given that this kind of substitution must therefore reconcile any injective identifier differences from N_R to N''_R, $[i \to R''_s(i)]N_R \subseteq N''_R$ must be hold by transitivity of the underlying subset relation.

Considering the rest of (ENV-LEQ) we choose without loss of generality any $i \in \mathit{domain}(N_{S^1})$, and for convenience we establish the following aliases: $R = N'_{R^*} \circ R_s$, $R' = N''_{R^*} \circ R'_s$, $R'' = N''_{R^*} \circ R''_s$, $i' = R'(i)$, and $i'' = R''(i)$. Note that $R'(R(i)) = R''(i)$ must hold for the same reason as $[i \to R''_s(i)]N_R \subseteq N''_R$.

In this context we show in all cases that $(N, N''); R'' \vdash i \sqsubseteq_i i''$ can be derived from $(N, N'); R \vdash i \sqsubseteq_i i'$ and $(N', N''); R' \vdash i' \sqsubseteq_i i''$.

Before considering topology we consider shape sequencing, taking each possible triple of shapes $s = N_{S^1}(i)$, $s' = N'_{S^1}(i')$ and $s'' = N''_{S^1}(i'')$:

- If $s = \bot$, then by (SHAPE-LEQ-BOT) any shape s'' is acceptable regardless of s'.
- If $s = \text{prim } p$, then since (SHAPE-LEQ-PRIM) only allows exactly the same shape to follow a primitive shape we know that $s' = \text{prim } p$ and $s'' = \text{prim } p$. From this, s and s'' satisfy (SHAPE-LEQ-PRIM) as well.

- If s = object O, then only (SHAPE-LEQ-OBJ) applies, since only an object shape can follow an object shape. This means that s' = object O' and s'' = object O'', and also that $\mathit{domain}(O) = \mathit{domain}(O') = \mathit{domain}(O'')$ from the transitivity of set equality. This leaves the identifiers of the object fields, which must satisfy the conditions in (SHAPE-LEQ-OBJ) by application of $R'(R(i)) = R''(i)$.
- If s = function $f; i_c$, then only (SHAPE-LEQ-FN) applies, since only a function shape can follow a function shape. This means that s' = function $f; i'_c$ and s'' = function $f; i''_c$. We can trivially see that f must stay the same across all three, and by $R'(R(i)) = R''(i)$ i_c and i''_c should also satisfy (SHAPE-LEQ-FN).

Now we can show the remaining topological requirements are met.

For the fields relation, we know from the preconditions that $\forall(i, n, i_f) \in N_F, (R(i), n, R(i_f)) \in N'_F$ and $\forall(i', n, i'_f) \in N'_F, (R'(i'), n, R'(i'_f)) \in N''_F$ have to hold. From the constraints on R we know that either $i' \notin \mathit{range}(R)$ or $R(i) = i'$ for some i, and similarly either $i'_f \notin \mathit{range}(R)$ or $R(i_f) = i'_f$. By function composition, identifiers not in the range of R have no transitive mapping in the range of R'' either. This means that all identifiers in $\mathit{range}(R'')$ must be accessible by some $R'(i')$. This allows us to combine the two preconditions via substitution, resulting in $\forall(i, n, i_f) \in N_F, (R''(i), n, R''(i_f)) \in N''_F$.

The same principles apply to the calls relation: by the same case splitting over i'_a, i' and i'_r in the preconditions $\forall(i_a, i, i_r) \in N_C, (R(i_a), R(i), R(i_r)) \in N'_C$ and $\forall(i'_a, i', i'_r) \in N'_C, (R'(i'_a), R'(i'), R'(i'_r)) \in N''_C$, we can perform an equivalent substitution giving $\forall(i_a, i, i_r) \in N_C, (R''(i_a), R''(i), R''(i_r)) \in N''_C$.

Lastly, a modification of the calls relation case gives the call fingerprint case: replace i_r with I_f and apply the substitution to all elements of I_f.

4.5 Core language grammar

This section provides a grammar that is intended to simplify TreeGen for the purpose of formal analysis. The resulting language, core TreeGen, is not very convenient to use, aiming to expose only the "core" semantics of the language. All missing features can be expressed as syntax sugar, and will be reintroduced by reducing full TreeGen to core TreeGen via syntax-driven rewrites in chapter 5.

Most notably, this core grammar strips out all notion of complex directives from the language. This makes object expressions much simpler, while the apparently lost functionality can be emulated using combinations of `???`, let bindings and the `effect` expression.

The type tagging and high-level control flow mechanisms are also implemented almost entirely via syntax sugar, with the only dedicated semantics being the `type(...)` constructor which maps arbitrary strings to type tags.

Lastly, the file generation primitive is entirely absent. As with other semantic extensions that do not affect the core evaluation mechanism or document structure, it can be described entirely orthogonally to the core semantics. A description of the file generation primitive is given in section 4.7, demonstrating how the semantics and proofs given here can be extended.

⟨*root*⟩ ::= ⟨*expression*⟩

⟨*expression*⟩ ::= '`???`'
| ⟨*literal*⟩
| '`type`' '`(`' ⟨*expression*⟩ '`)`'
| ⟨*name*⟩
| ⟨*function-expression*⟩
| ⟨*expression*⟩ '`/`' ⟨*name*⟩
| '`effect`' ⟨*expression*⟩ '`=`' ⟨*expression*⟩ '`;`' ⟨*expression*⟩
| '`{`' (⟨*name*⟩ '`=`' ⟨*expression*⟩ '`;`')* '`}`'
| ⟨*expression*⟩ '`(`' ⟨*expression*⟩ '`)`'
| '`let`' ⟨*name*⟩ '`=`' ⟨*expression*⟩ '`in`' ⟨*expression*⟩
| '`if`' ⟨*expression*⟩ '`then`' ⟨*expression*⟩ '`else`' ⟨*expression*⟩
| ⟨*expression*⟩ ⟨*binop*⟩ ⟨*expression*⟩
| '`!`' ⟨*expression*⟩
| ⟨*expression*⟩ '`==`' ⟨*expression*⟩

⟨*literal*⟩ ::= '`true`' | '`false`'
| ⟨*integer*⟩
| ⟨*string*⟩

Most of the operations described here correspond directly to previous informal introductions. The listed binary operations are limited to a representative set, respectively: boolean conjunction and disjunction, integer addition, subtraction, multiplication and division, and string concatenation.

⟨*binop*⟩ ::= '`&&`' | '`||`' | '`+`' | '`-`' | '`*`' | '`div`' | '`++`'

There is only one operation that counts as a unary primitive operation: ! corresponds to boolean negation.

In addition to primitive operations, the equality comparison == requires special treatment and is defined separately.

Function expressions are defined separately so that they can be referenced by name: this is the function expression AST referenced by the definition of function shapes.

⟨*function-expression*⟩ ::= \\⟨*name*⟩ '`=>`' ⟨*expression*⟩

4.6 Operational semantics

TreeGen programs do not have a direct syntax-driven evaluation. It is desirable to give the language small-step semantics in order to better reason about concepts like determinism, but TreeGen's admission of mutability does not allow for traditional expression-rewriting small-step semantics.

Instead, we express TreeGen's evaluation as a *chaotic iteration* that generates increasing sequences of program state triples $E = (N, P, D)$. N corresponds to the environments we have already defined, whereas P and D introduce the key aspects of TreeGen's operational semantics: *constraints* and *deferrals*. Both P and D contain unordered collections of constraints and deferrals that represent requirements on the environment. These can be applied idempotently and monotonically in any order as shown below, with the top-level *eval* function using the family of functions $step_k$ to repeatedly apply one of the constraints or deferrals until none of the available operations has any further effect.

$eval(e) =$
  let $(P_0, I_f, i_r) = trans(\emptyset, e)$
  let $E_0 = ((\{(i, \bot) \mid i \in range(I_f)\}, \emptyset, \emptyset, \emptyset, \emptyset), P_0, \emptyset)$
  let $E = E_0$
  while $\exists k \in \mathbb{N}$, $E \neq step_k(E)$
    choose $k \in \mathbb{N}$, $E := step_k(E)$
  assert $\forall d \in E_D$, $\forall i \in waits\_on(E_N, d)$, $E_{N_{S^1}}(i) \neq \bot$
  return $(E_N, E_{N_{R^*}}(i_r))$

This definition $eval(e) = (N, i_r)$ gives the top-level logic for evaluating a TreeGen program. First, the function $trans(\gamma, e) = (P_0, I_f, i_r)$ gives a translation from an expression e under scope $\gamma : names \twoheadrightarrow identifiers$ into an initial set of constraints P_0, a call fingerprint I_f and the identifier of the resulting document's intended root value i_r. Then, E is constructed as a starting environment with all the identifiers in I_f assigned the shape $\bot$, the starting set of constraints P_0 and no further information. Based on this, E is repeatedly rewritten by a nondeterministically chosen element of the function family $step_k$ until there exists no choice of k that leads to any change in E. Given this final E, we need to check that all the deferrals have been resolved: the assertion at the end expresses this – TreeGen programs are not allowed to terminate before all deferrals have had a chance to execute. This is defined in terms of the $waits\_on$ function which gives the set of identifiers that must have a non-$\bot$ shape for a deferral d to have chance to execute. With that assertion taken care of, the overall result is the final environment E_N and the final root identifier i_r with any relevant rewrites in E_{N_R} applied.

eval will be shown to be a function despite its nondeterministic behaviour, thanks to Geser et al.'s theorem on chaotic fixpoint iterations [12] (summarised below). This requires proving the following properties for TreeGen's semantics:

- Program states must form a well-founded partial order over program state sequencing, an extension of environment sequencing that additionally handles P and D.
- For all k, $step_k$ must be shown to be both increasing and monotone.

Proofs for each of these properties will be given alongside precise definitions of the semantics of $step_k$, all constraints and all deferrals. First though the top-level reasoning behind the proof will be given alongside introductory notes on the constraints and deferrals available, since many of the precise semantics are mutually referential.

Top-level proof The top-level proof is based on an existing theorem by Geser et al. on the determinism of chaotic fixpoint iterations [12]. We apply their theorem to program states E and the function family $step_k$.

Their theorem requires a well-founded partial order $(\mathcal{E}, \sqsubseteq)$ (where all environments $E \in \mathcal{E}$) with least element E_0, and a function family $\mathcal{F} = \{step_k \mid k \in \mathbb{N}\}$ of delay-monotone increasing functions. The partial ordering of environments has already been shown, whereas the function family $\mathcal{F}$ describes any step possible for any given program state, and is described in the following sections. Alongside these definitions, all functions $step_k \in \mathcal{F}$ will be shown to be increasing and monotone over environments.

Geser et al. actually require a weaker pre-condition called delay-monotonicity which only states that a given $step_k$ must eventually have the same "overall effect", allowing for an arbitrary *delay* during which different applications of the same step can have transiently differing "overall effects". This is unnecessary for TreeGen's determinism proof since we can show the simpler and stronger condition of plain monotonicity instead, but it can be useful in cases where only delay-monotonicity can be shown.

The other requirement is a fair strategy $\gamma : \mathbb{N} \to \mathbb{N}$, which represents a sequence of steps $k \in range(\gamma),\ step_k \ldots$, such that no progress-making choice $k' \in \mathbb{N}$ is starved of execution indefinitely. In TreeGen's case, this means that we assume an implementation has some way of trying constraints and/or deferrals without indefinitely skipping a particular constraint or deferral that could make progress. This could be achieved in practice via a simple round-robin scheduler or some kind of properly maintained worklist.

In keeping with the classical fixpoint arguments, the overall intuition is that any operation by a given TreeGen program must lead to progress in terms of environment sequencing, and that any operation must have an equivalent effect no matter when it is applied.

Then, the least fixpoint $\mu\mathcal{F}$ of $\mathcal{F}$ exists and is given by $\bigsqcup step_\gamma(E_0)$, and $\mu\mathcal{F}$ is always reached within a finite number of iteration steps (assuming execution terminates at all, which may be prevented by errors or infinitely recursive function calls).

Constraints The set of possible constraints is defined by the following grammar:

⟨*constraint*⟩ ::= '`shape`' '`(`' ⟨*shape*⟩ '`,`' ⟨*identifier*⟩ '`)`'
| '`merge`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*identifier*⟩ '`)`'
| '`project`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*name*⟩ '`,`' ⟨*identifier*⟩ '`)`'
| '`call`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*identifier*⟩ '`,`' ⟨*identifier*⟩ '`)`'
| '`defer`' '`(`' ⟨*deferral*⟩ '`)`'

- The constraint `shape`(s, i_r) constraints the identifier i_r to refer to a value whose shape is at least s, while enforcing any relevant uniqueness rules. This is used to unconditionally give an identifier a known shape.
- The next constraint is `merge`(i, i'), which corresponds to the merge operation that has already been defined informally. It rewrites a given program state E to another state E' such that all instances of the identifier i are replaced with the identifier i', accounting for all of the structural integrity rules for environments. This includes the constraint itself, meaning that after executing, $merge(i, i')$ will instead read $merge(i', i')$, which represents a no-op.

- The `project`(i, n, i_f) constraint implements field projection: it constrains i_f be the identifier of i's field named n.
- The `call`(i_a, i_f, i_r) constraint implements function call semantics: it constrains i_r to refer to the result of calling i_f with argument i_a. This constraint has a related function call deferral that is used in cases where the call actually needs evaluating.
- The last constraint is `defer`(d), which leads into the concept of deferrals. This constraint ensures that D contains deferral d, meaning that d should eventually execute via the deferral semantics described next.

Deferrals Deferrals represent any kind of specialised computation available in TreeGen that has been previously described as operating via "deferred evaluation". Almost all deferrals only act on *stable information*, which is an important shortcut to showing a given deferral's execution must be increasing and monotone.

Specifically, stable information can be:

- Primitive shapes, since by (SHAPE-LEQ-PRIM) the only shape that can follow a primitive shape is that same primitive shape.
- Object shape fields, since by (SHAPE-LEQ-OBJ) object shapes can only be followed by object shapes that have the same set of fields.
- Function shape function expressions, since by (SHAPE-LEQ-FN) a function shape can only be followed by another function shape that has references the same function expression.

The set of possible deferrals is defined via the following grammar:

⟨*deferral*⟩ ::= ⟨*identifier*⟩ '`:=`' ⟨*identifier*⟩ '`(`' ⟨*identifier*⟩ '`)`'
| '`if`' ⟨*identifier*⟩ '`then`' { ⟨*operation*⟩* } '`else`' { ⟨*operation*⟩* }
| ⟨*identifier*⟩ '`:=`' '`!`' ⟨*identifier*⟩
| ⟨*identifier*⟩ '`:=`' '`type`' '`(`' ⟨*identifier*⟩ '`)`'
| ⟨*identifier*⟩ '`:=`' ⟨*identifier*⟩ ⟨*binop*⟩ ⟨*identifier*⟩
| ⟨*identifier*⟩ '`:=`' ⟨*identifier*⟩ '`==`' ⟨*identifier*⟩

Deferrals *wait on* one of more of the identifiers they reference, requiring them to have some shape other than $\perp$ before the deferral should be evaluated. The function *waits_on*,

defined below, gives the set of waited on identifiers for each kind of deferral in terms of the deferral itself and a given environment. All but one of the definitions are syntactically self-explanatory, the exception being equality comparison whose definition in terms of *stably_reachable_subvalues* will be explained alongside the deferral's in-detail semantics. If *waits_on* is not purely a function of d, then assume that it produces an increasing set of identifiers modulo substitution, for increasing sequences of environments N.

$$
\begin{aligned}
&\mathit{waits\_on}(N, [\![i_r := i_f(i_c)]\!]) = \{i_f\} \\
&\mathit{waits\_on}(N, [\![\texttt{if } i_c \texttt{ then } p_t \texttt{ else } p_f]\!]) = \{i_c\} \\
&\mathit{waits\_on}(N, [\![i_r := !i_c]\!]) = \{i_c\} \\
&\mathit{waits\_on}(N, [\![i_r := \texttt{type}(i_t)]\!]) = \{i_t\} \\
&\mathit{waits\_on}(N, [\![i_r := i_1 \ b \ i_2]\!]) = \{i_1, i_2\} \\
&\mathit{waits\_on}(N, [\![i_r := i_1 == i_2]\!]) = \\
&\quad \mathit{stably\_reachable\_subvalues}(\emptyset, N, i_1) \cup \mathit{stably\_reachable\_subvalues}(\emptyset, N_{,2})
\end{aligned}
$$

Each individual deferral works as follows:

- $i_r := i_f(i_a)$ implements the execution of a function body: it waits on i_f, then performs the evaluation of the function body associated with the function-shaped value with identifier i_f.
- `if` i_c `then` p_t `else` p_f is the conditional branching deferral: assuming i_c's shape is a primitive boolean, if i_c has the primitive boolean shape true, then the constraints p_t are executed, otherwise the constraints p_f are executed.
- $i_r := !i_c$ means primitive negation: if i_c is a primitive boolean true, then i_r becomes primitive boolean false, and vice-versa.
- $i_r :=$ `type`(i_t) constrains i_r to be a type-shaped value: i_t must have a primitive string shape, which the operation wraps in a primitive type shape.
- $i_r := i_1 \ b \ i_2$ means a primitive binary operation: it waits on both i_1 and i_2 to be primitive values, then constrains i_r to be the result of primitive operation b applied to i_1 and i_2.

- $i_r := i_1 == i_2$ means equality comparison: it waits on both i_1 and i_2, and performs a specialised algorithm to check whether i_1 and i_2 stably refer to the same value or not.

4.6.1 Program state

Program state's structural rules can be expressed as fairly straightforward extensions to the rules for environments.

$$\begin{aligned} E \cong E' = &\exists R_s, E_N \cong E'_N, \\ &[i \to R_s(i)](E_P, E_D) = (E'_P, E'_D) \end{aligned} \tag{4.6.1}$$

Firstly, eq. (4.6.1) shows an extension of environment congruence to state congruence. R_s represents the same R_s chosen in the definition of $E_N \cong E'_N$. All the structural properties are maintained, with P and D following the same rules as F, C and C_f.

$$\frac{\begin{array}{cc} \exists R_s, E_N \sqsubseteq E'_N & R = R_s \circ E_{N_{R^*}} \\ [i \to R(i)]E_P \subseteq E'_P & [i \to R(i)]E_D \subseteq E'_D \end{array}}{E \sqsubseteq E'} \quad \text{(STATE-LEQ)}$$

Likewise, the rule (STATE-LEQ) extends sequencing from environments to shapes. R_s is the same R_s chosen by the definition of $E_N \sqsubseteq E'_N$, and the composition R defines the full set of substitutions mapping identifiers in E to those in E'. Note that because R includes the non-injective function $E_{N_{R^*}}$, it is entirely possible for $[i \to R(i)]E_P$ to shrink relative to E_P. For instance, performing the substitution $i_1 \to i_2$ on $P = \{[\![\texttt{merge}(i_1, i_2)]\!], [\![\texttt{merge}(i_2, i_2)]\!]\}$ would yield the smaller set of constraints $\{[\![\texttt{merge}(i_2, i_2)]\!]\}$, since once i_1 is replaced with i_2, the two elements in the original set are no longer distinct. The same can be applied to D.

Now that we have extended the definition itself, we can also extend the proofs, showing that $E \sqsubseteq E'$ gives a partial ordering over program states. In each case we extend the existing proofs involving E_N to account for the added E_P and E_D.

Reflexivity Since we know from eq. (4.6.1) that $[i \to R_s(i)](E_P, E_D) = (E'_P, E'_D)$, we can claim that $[i \to R(i)]E_P \subseteq E'_D$ and $[i \to R(i)]E_D \subseteq E'_D$ from (STATE-LEQ) are trivially satisfied via existing arguments from the proof for environments. We know that E_{N_R} does not meaningfully contribute to R, so substituting by R is equivalent to substituting by R_s.

R_s has already been shown to be bijective, so we can rely on the reflexivity of the subset relation.

Antisymmetry Since we know from the proof for environments that R from $E_N \sqsubseteq E'_N$ (STATE-LEQ) has a corresponding inverse R^{-1} for $E'_N \sqsubseteq E_N$, we can rewrite $[i \to R^{-1}(i)]E'_P \subseteq E_P$ to $E'_P \subseteq [i \to R(i)]E_P$. From there we can invoke antisymmetry of the underlying subset relation to get $[i \to R_s(i)]E_P = E'_P$, since $E_{N_{R^*}}$ has been shown to have no effect on R. A similar argument can be made to show $[i \to R(i)]E_D = E'_D$.

Transitivity This can be argued from a similar function composition argument to the rest of the proof for environments. Since all substitutions in R and R' may either preserve distinct identifiers or lose some distinct identifier by mapping multiple identifiers to the same identifier, the composition R'' will map $\mathit{domain}(R)$ to some subset of the distinct identifiers mapped to by R'. This means $[i \to R''(i)E_P \subseteq [i \to R'(i)]E'_P$, which combined with $[i \to R'(i)]E'_P \subseteq E''_P$ gives $[i \to R''(i)E_P \subseteq E''_P$. This argument can be repeated for E_D.

4.6.2 Translation

This section gives a precise definition for $\mathit{trans}(\gamma, e) = (P, I_f, i_r)$, relating an expression e and a scope γ to a set of constraints P, a tuple of fresh initially $\bot$-shaped identifiers I_f and the root identifier of the value that is semantically the result of the expression e, i_r. This also doubles as a tour by example of the in-context uses of constraints and deferrals.

Fresh identifiers A common notation used mostly in *trans* is $\mathbf{fresh\_id}_k$, which refers to a globally unique fresh identifier. Note that k only needs to be locally unique: $\mathbf{fresh\_id}_1$ referenced in one definition should not be considered the same as $\mathbf{fresh\_id}_1$ referenced in another definition.

Simple literals

There are two simple literal expressions: unknown expressions and primitive literals.

$$\mathit{trans}(\gamma, [\![???]\!]) =$$
$$(\{[\![\texttt{shape}(\bot, \mathbf{fresh\_id}_r)]\!]\}, (\mathbf{fresh\_id}_r), \mathbf{fresh\_id}_r)$$

The unknown expression translates to a fresh identifier that is unconstrained, thus having shape $\bot$ unless otherwise indicated.

$trans(\gamma, [\![l]\!])$ if $\textbf{is\_literal}(l) =$
$(\{[\![\texttt{shape}(\text{prim } l, \textbf{fresh\_id}_r)]\!]\}, (\textbf{fresh\_id}_r), \textbf{fresh\_id}_r)$

The primitive literal expression constrains its result to have a primitive shape. The shape constraint will take care of things like uniqueness rules as and when they apply.

Type constructor expressions

$trans(\gamma, [\![\texttt{type}(e)]\!]) =$
let $(P, I_f, i_e) = trans(\gamma, e)$
let $i_r = \textbf{fresh\_id}_r$
return $(P \cup \{[\![\texttt{defer}(\texttt{type}(i_e) = i_r)]\!])[\![]\!]\}, (I_f \ldots, i_r), i_r)$

The type constructor expression is the first example of the defer constraint: its translation emits a deferral that should eventually ensure that the fresh identifier $\textbf{fresh\_id}_r$ will refer to a type tag based on the assumed eventual literal shape of i_e.

Name expressions

$trans(\gamma, [\![n]\!])$ if $\textbf{is\_name}(n) =$
$(\emptyset, (), \gamma(n))$

The name expression simply looks into γ for the identifier associated with the name n. For any valid program n should always be in $domain(\gamma)$.

Function expressions

$trans(\gamma, [\![\backslash\backslash n => e]\!]) =$
let $(i_r, i_c) = (\textbf{fresh\_id}_r, \textbf{fresh\_id}_c)$
let $O = \{(n, \gamma(n)) \mid n \in closure(\backslash\backslash n => e)\}$
return $(\{[\![\texttt{shape}(\text{object } O, i_c)]\!], [\![\texttt{shape}(\text{function } \backslash\backslash n => e; i_c, i_r)]\!]), (i_r, i_c), i_r)$

Function expressions emit the constraints necessary to build both a function-shaped value $\textbf{fresh\_value}_r$ and the corresponding closure object $\textbf{fresh\_value}_c$. The function shape just embeds the function expression itself, while the closure object captures all the identifiers in γ referenced by the function expression's free variables $\mathit{closure}(\backslash\backslash n => e)$.

The set of free variables for any expression is defined by the $\mathit{closure}(e)$ function given below.

$$\mathit{closure}([\![???]\!]) = \emptyset$$
$$\mathit{closure}([\![l]\!]) \text{ if } \textbf{is\_literal}(l) = \emptyset$$
$$\mathit{closure}([\![\texttt{type}(e)]\!]) = \mathit{closure}(e)$$
$$\mathit{closure}([\![n]\!]) \text{ if } \textbf{is\_name}(n) = \{n\}$$
$$\mathit{closure}([\![\backslash\backslash n => e]\!]) = \mathit{closure}(e) \setminus \{n\}$$
$$\mathit{closure}([\![e/n]\!]) = \mathit{closure}(e)$$
$$\mathit{closure}([\![\texttt{effect } e_l = e_r; e_b]\!]) = \mathit{closure}(e_l) \cup \mathit{closure}(e_r) \cup \mathit{closure}(e_b)$$
$$\mathit{closure}([\![\{d\}]\!]) = \left(\bigcup_{[\![n=e]\!] \in d} \mathit{closure}(e) \right) \setminus \{n \mid [\![n = e]\!] \in d\}$$
$$\mathit{closure}([\![e_f(e_a)]\!]) = \mathit{closure}(e_f) \cup \mathit{closure}(e_a)$$
$$\mathit{closure}([\![\texttt{let } n = e_v \texttt{ in } e_b]\!]) = (\mathit{closure}(e_v) \cup \mathit{closure}(e_b)) \setminus \{n\}$$
$$\mathit{closure}([\![\texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f]\!]) = \mathit{closure}(e_c) \cup \mathit{closure}(e_t) \cup \mathit{closure}(e_f)$$
$$\mathit{closure}([\![e_1\ b\ e_2]\!]) \text{ if } \textbf{is\_binop}(b) = \mathit{closure}(e_1) \cup \mathit{closure}(e_2)$$
$$\mathit{closure}([\![!e]\!]) = \mathit{closure}(e)$$
$$\mathit{closure}([\![e_1 == e_2]\!]) = \mathit{closure}(e_1) \cup \mathit{closure}(e_2)$$

Projection expressions

$$\begin{aligned}
&\mathit{trans}(\gamma, [\![e/n]\!]) = \\
&\quad \text{let } (P, I_f, i_e) = \mathit{trans}(\gamma, e) \\
&\quad \text{let } \textbf{fresh_id}(i) \\
&\quad \text{return } (P \cup \{[\![\texttt{project}(i_e, n, \textbf{fresh_id}_r)]\!]\}, (I_f \ldots, \textbf{fresh_id}_r), \textbf{fresh_id}_r)
\end{aligned}$$

The projection expression adds a constraint that the fresh identifier $\textbf{fresh\_id}_r$ be the result of projecting field n from the value i_e.

Effect expressions

$$
\begin{aligned}
&\mathit{trans}(\gamma, [\![\texttt{effect}\ e_1 = e_2; e_b]\!]) = \\
&\quad \text{let } (P_1, I_{f_1}, i_1) = \mathit{trans}(\gamma, e_1) \\
&\quad \text{let } (P_2, I_{f_2}, i_2) = \mathit{trans}(\gamma, e_2) \\
&\quad \text{let } (P_b, I_{f_b}, i_b) = \mathit{trans}(\gamma, e_b) \\
&\quad \text{return } (P_1 \cup P_2 \cup P_b \cup \{[\![\texttt{merge}(i_1, i_2)]\!]\}, (I_{f_1} \ldots, I_{f_2} \ldots, I_{f_b} \ldots), i_b)
\end{aligned}
$$

The effect expression is essentially a proxy for the programmer to request an additional merge constraint: each expression's constraints are emitted alongside an additional constraint that the identifiers i_1 and i_2 be merged.

Object expression

$$
\begin{aligned}
&\mathit{trans}(\gamma, [\![\{d\}]\!]) = \\
&\quad \text{let } O = \{(n, \textbf{fresh\_id}_n) \mid [\![n = e_n]\!] \in d\} \\
&\quad \text{let } \gamma' = O \cup \{(n, i_f) \mid (n, i_f) \in \gamma, n \notin \mathit{domain}(O)\} \\
&\quad \text{let } (P, I_f) = \mathit{obj\_trans}(\gamma', d) \\
&\quad \text{return } (P \cup \{[\![\texttt{shape}(\text{object } O, i)]\!]\}, (I_f \ldots, \mathit{range}(O) \ldots, \textbf{fresh\_id}_r), \textbf{fresh\_id}_r)
\end{aligned}
$$

Object expressions work in layers: as was described informally, each of the names defined inside the object expression is collected and given a matching fresh identifier. This defines O which will become part of the object shape. Then the scope for sub-expressions γ' is defined as containing all bindings in γ and O, with O taking priority for bindings in both. Once all the scoping is resolved the constraints for each name-expression pair are collected via the helper *obj_trans*, then finally a constraint that the expression's result, a fresh identifier $\textbf{fresh\_id}_r$, should have shape object O is added.

$obj\_trans(\gamma, [\![]\!]) = (\emptyset, ())$
$obj\_trans(\gamma, [\![n = e; d]\!]) =$
  let $(P, I_f) = obj\_trans(\gamma, d)$
  let $(P_e, I_{f_e}, i_e) = trans(\gamma, e)$
  return $(P \cup P_e \cup \{[\![\texttt{merge}(i_e, \gamma(n)]\!]\}, (I_f \ldots, I_{f_e} \ldots))$

obj_trans finds the constraints for each sub-expression and emits them in sequence alongside constraints that the result of each sub-expression be merged with the identifier associated with that particular name. The extra merge constraints are the same as those described informally, ensuring that that the sub-expression's result is the same value as the value bound to the field name n.

Function call expressions

$trans(\gamma, [\![e_f(e_a)]\!]) =$
  let $(P_f, I_{f_f}, i_f) = trans(\gamma, e_f)$
  let $(P_a, I_{f_a}, i_a) = trans(\gamma, e_a)$
  let $i_r = \mathbf{fresh\_id}_r$
  return $(P_f \cup P_a \cup \{[\![\texttt{call}(i_a, i_f, i_r)]\!]\}, (I_{f_f} \ldots, I_{f_s} \ldots, i_r), i_r)$

Function call expressions delegate almost all of their behaviour to the function call constraint, meaning that the translation itself is quite simple. Constraints for the function and argument sub-expressions are emitted alongside the function call constraint.

Let expressions

$trans(\gamma, [\![\texttt{let } n = e_v \texttt{ in } e]\!]) =$
  let $\gamma' = \{(n, \mathbf{fresh\_id}_v)\} \cup \{(n', i_n) \mid (n', i_n) \in \gamma, n' \neq n\}$
  let $(P_v, I_{f_v}, i_v) = trans(\gamma', e_v)$
  let $(P, I_f, i) = trans(\gamma', e)$
  return $(P_v \cup P \cup \{[\![\texttt{merge}(i_v, \mathbf{fresh\_id}_v)]\!]\}, (I_{f_v} \ldots, I_f \ldots), i)$

Let expressions bind the name n to the fresh value $\textbf{fresh\_id}_v$, translate both sub-expressions in that scope and emit a merge constraint that the result of e_v and $\textbf{fresh\_id}_v$ be the same value. Note that unlike in the main language where there are multiple scoping options, these core expressions always perform a fully recursive binding, meaning that n can be referenced in e_v and e. This is because it is easier to disallow recursive references via term rewriting than it is to emulate them via term rewriting given a non-recursive core let binding.

If expressions

$$
\begin{aligned}
&\mathit{trans}(\gamma, [\![\texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f]\!]) = \\
&\quad \text{let } (P_c, I_{f_c}, i_c) = \mathit{trans}(\gamma, e_c) \\
&\quad \text{let } (P_t, I_{f_t}, i_t) = \mathit{trans}(\gamma, e_t) \\
&\quad \text{let } (P_f, I_{f_f}, i_f) = \mathit{trans}(\gamma, e_f) \\
&\quad \text{let } P_r = \{[\![\texttt{defer}(\texttt{if } i_c \texttt{ then } P_t \texttt{ else } P_f)]\!], [\![\texttt{merge}(i_t, i_f)]\!]\} \\
&\quad \text{return } (P \cup P_r, (I_{f_c} \ldots, I_{f_t} \ldots, I_{f_f} \ldots), i_t)
\end{aligned}
$$

Conditionals delegate most of their behaviour to the conditional branching deferral. The constraints from all the sub-expressions are computed, but only those for the conditional itself are emitted directly. The rest are packaged into the conditional branching deferral, such that depending on whether i_c gains a primitive true or false shape either P_t or P_f are actually added to the program state.

Aside from the expression's actual behaviour this formulation has a quirk: after the deferral a merge is given between i_t and i_f, with the i_t identifier being returned as the result. This is a shortcut to avoid the equivalent but longer formulation of declaring a fresh result value and adding merges between that identifier and either i_r or i_f to P_t or P_f respectively. The shortcut works because we know that only one of P_t or P_f will ever be applied. Merging i_f and i_t ensures that regardless of which set of constraints was applied i_t will refer to that subexpression's result.

Note also that all identifiers from both true and false branches are made part of the expression's fingerprint, meaning that at the expense of having extra fresh identifiers left over from whichever branch we did not take we can guarantee that any generated identifiers are captured regardless of conditional execution.

Binary and unary operation expressions

$trans(\gamma, [\![e_1\ b\ e_2]\!])$ if $\textbf{is\_binop}(b) =$
  let $(P_1, I_{f_1}, i_1) = trans(\gamma, e_1)$
  let $(P_2, I_{f_2}, i_2) = trans(\gamma, e_2)$
  let $i_r = \textbf{fresh\_id}_r$
  return $(P_1 \cup P_2 \cup \{[\![\texttt{defer}(i_r := i_1\ b\ i_2)]\!]\}, (I_{f_1} \ldots, I_{f_2} \ldots, i_r), i_r)$

$trans(\gamma, [\![!e]\!]) =$
  let $(P, I_f, i) = trans(\gamma, e)$
  return $(P \cup \{[\![\texttt{defer}(\textbf{fresh\_id}_r :=!i)]\!]\}, (I_f \ldots, \textbf{fresh\_id}_r), \textbf{fresh\_id}_r)$

Both of these expressions are really just the same thing with two different arities. In each case all of the sub-expressions' constraints are emitted while the primitive operation is dispatched via the related deferral (either unary negation or the general binary operation).

$trans(\gamma, [\![e_1 == e_2]\!])$ if $\textbf{is\_binop}(b) =$
  let $(P_1, I_{f_1}, i_1) = trans(\gamma, e_1)$
  let $(P_2, I_{f_2}, i_2) = trans(\gamma, e_2)$
  let $i_r = \textbf{fresh\_id}_r$
  return $(P_1 \cup P_2 \cup \{[\![\texttt{defer}(i_r := i_1 == i_2]\!]\}, (I_{f_1} \ldots, I_{f_2} \ldots, i_r), i_r)$

Equality comparison is not a primitive operation so it is described separately. At the translation level it is identical to the primitive binary operation, the difference coming from the fact that since equality can compare more than just primitive values we need a different algorithm to implement it in *undefer*. To clearly reflect this distinction we keep the definitions for equality expressions, translations and deferrals separate from the other binary operations.

4.6.3 The step function family

This section describes the family of step functions, where nondeterminism is modeled by $k \in \mathbb{N}$. choose_k is used to express a consistent selection from some set based on k.

$step_k(E) =$
  if $k \bmod 2 = 1$
    choose$_k$ $p \in E_P$
      $constrain(p, E)$
  else
    choose$_k$ $d \in E_D$
      if $\forall i \in waits\_on(E_N, d), E_{N_{S^1}}(i) \neq \bot$
        $undefer(d, E)$
      else E

For *eval* to be deterministic we need to show that for all k, $step_k$ is both increasing and monotone, as defined by defs. 4.6.2 and 4.6.3. For this, we rely on the increasingness and monotonicity of both $constrain(p, E)$ and $undefer(d, E)$.

$$E \sqsubseteq step_k(E) \tag{4.6.2}$$

$$E \sqsubseteq E' \implies step_k(E) \sqsubseteq step_k(E') \tag{4.6.3}$$

From the definition's structure, we can note that for all odd k $step_k$ directly corresponds to the function $constrain(p, E)$ for some selection of $p \in E_P$.

For all even k, $step_k$ corresponds, via a slight indirection to $undefer(d, E)$. This indirection can be trivially shown to preserve increasingness as long as the result of $waits\_on(E_N, d)$ is increasing itself under substitution by R^*, since if $\forall i \in waits\_on(E_N, d), E_{N_{S^1}}(i) \neq \bot$ then the result directly relies on increasingness of $undefer$. Otherwise, the result is exactly E, which by reflexivity of the sequencing relation must satisfy def. 4.6.2.

Showing monotonicity of the indirection is a little more involved, since there are cases where in def. 4.6.3 E' could satisfy $\forall i \in waits\_on(E'_N, d), E'_{N_{S^1}}(i) \neq \bot$ whereas for E there might exist $i \in waits\_on(E_N, d)$ such that $E_{N_{S^1}}(i) = \bot$. We can break this down into the possible cases:

- First, neither E nor E' satisfy the condition and in both cases $step_k(E) = E$. This satisfies monotonicity by reflexivity of the sequencing relation.

- Second, as mentioned above E does not satisfy the condition but E' does. This gives us $E \sqsubseteq E' \implies E \sqsubseteq \mathit{undefer}(d, E')$. By transitivity of the sequencing relation we can infer that $E \sqsubseteq E' \implies E' \sqsubseteq \mathit{undefer}(d, E')$, which directly corresponds to the increasingness of *undefer*.
- It is not possible to have a case where E satisfies the condition but E' does not – this would require a value to have a shape that is not $\bot$ then change to have the shape $\bot$. This is impossible by the shape sequencing rules. Only (SHAPE-LEQ-BOT) mentions $\bot$,and it only allows a value to stay $\bot$-shaped or change to another shape, not change back.
- The third possible case is if both E and E' satisfy the condition. In that case, we directly rely on the monotonicity of *undefer*.

4.6.4 Constraints

The semantics of constraints are given by the piecewise defined function $\mathit{constrain}(p, E) = E'$. Each kind of constraint will be given precise semantics alongside proofs that that constraint's execution is both increasing and monotonic over state sequencing, as defined by defs. 4.6.4 and 4.6.5.

$$E \sqsubseteq \mathit{constrain}(p, E) \tag{4.6.4}$$

$$E \sqsubseteq E' \implies \mathit{constrain}(p, E) \sqsubseteq \mathit{constrain}([i \to E'_{N_{R^*}}(i)]p, E') \tag{4.6.5}$$

Notice that def. 4.6.5 takes special care to account for constraint substitutions: to ensure that both applications of *constraint* are referring to notionally the "same" constraint p, we apply possible substitutions in E'_{N_R} to ensure that the "later" p refers to the correct identifiers for E'.

Merge

The merge constraint ensures that one identifier i_1 is replaced by another identifier i_2 while preserving environment invariants. This is achieved in two stages: the local stage, where all instances of i_1 are replaced with i_2, and a new combined shape is computed; and the global stage, where any uniqueness violations are addressed.

$$
\begin{aligned}
&\mathit{next\_shape}(s_1, s_2) = \\
&\quad (s_1, s_2) \text{ match} \\
&\qquad \text{case } (\bot, s) \to (s, \emptyset) \\
&\qquad \text{case } (s, \bot) \to (s, \emptyset) \\
&\qquad \text{case } (\text{prim } p, \text{prim } p) \to (\text{prim } p, \emptyset) \\
&\qquad \text{case } (\text{object } O_1, \text{object } O_2) \text{ if } \mathit{domain}(O_1) = \mathit{domain}(O_2) \to \\
&\qquad\quad (\text{object } O_2, \{[\![\texttt{merge}(i_f, O_2(n))]\!] \mid (n, i_f) \in O_1, i_f \neq O_2(n)\}) \\
&\qquad \text{case } (\text{function } f; i_{c_1}, \text{function } f; i_{c_2}) \to \\
&\qquad\quad (\text{function } f; i_{c_2}, \{[\![\texttt{merge}(i_{c_1}, i_{c_2})]\!]\})
\end{aligned}
$$

First and foremost, $\mathit{next\_shape}(s_1, s_2) = (s', m_s)$ represents the creation of a new shape s' by combining the shapes s_1 and s_2. m_s is a sequence of shape-based *follow-up merges*. The valid shape combinations are as follows, corresponding directly to the different shape sequencing rules:

- First, we have the commutative cases of merging the $\bot$ shape with any other shape. This should succeed as long as the value with shape $\bot$ follows the shape invariants for shape s: given N must be a valid environment, the value with shape s must already respect that shape's invariants. This condition is because the value with shape $\bot$ has no reason to follow shape s's invariants up to this point. To make sure only programs that respect this condition succeed, we require asserting that all the shape invariants hold, as expressed by the predicate *shape_invariants* defined below.
- Primitive shapes can only be combined with the same primitive shape.
- Object shapes can be merged as long as all the field names match. Follow-up work is needed to ensure that the field values are merged as well, which is taken care of by the generated follow-up merges m_s. The object shape O_2 is returned as a temporary measure – the field names should always be correct, but the field value identifiers may be temporarily incorrect until the follow-up merges execute.
- Function shapes can merge if their ASTs f are equal. In this case, their closure objects c_1 and c_2 must also become equal. That is why a follow-up merge between c_1 and c_2 is emitted.

If none of the cases apply, then the merge is considered to have failed. If a new shape is successfully completed, and assuming the follow-up merges complete and the shape invariants given below pass, then an environment incorporating the resulting fresh shape will be valid and greater than or congruent with the starting environment.

$shape\_invariants(N, s, i) =$
  s match
    case $\bot \rightarrow$ *true*
    case prim $p \rightarrow$
      $\nexists(i, n, i_f) \in N_F \wedge \nexists(i_a, i, i_r) \in N_C$
    case object $O \rightarrow$
      $(\nexists(i, n, i_f) \in N_F, n \notin domain(O)) \wedge \nexists(i_a, i, i_f) \in N_C$
    case function $f; i_c \rightarrow$
      $\nexists(i, n, i_f) \in N_F$

$shape\_invariants(N, s, i)$ checks that the value with identifier i follows shape s's invariants. The conditions expressed correspond directly to those defined by defs. 4.2.7 to 4.2.10.

$$
\begin{aligned}
&\mathit{constrain}([\![\texttt{merge}(i_1, i_2)]\!], E) = \\
&\quad \text{if } i_1 = i_2 \text{ then } E \\
&\quad \text{else} \\
&\qquad \text{let } (s', m_s) = \mathit{next\_shape}(E_{N_{S^1}}(i_1), E_{N_{S^1}}(i_2)) \\
&\qquad \text{let } S' = [i_1 \to i_2](\{(i, s) \mid (i, s) \in E_{N_S}, i \notin \{i_1, i_2\}\} \cup \{i_2 \to s'\}) \\
&\qquad \text{let } (F', C', C'_f, P', D') = [i_1 \to i_2](E_{N_F}, E_{N_C}, E_{N_{C_f}}, E_P, E_D) \\
&\qquad \text{let } R' = E_{N_R} \cup \{(i_1, i_2)\} \\
&\qquad \text{let } m_S = \{[\![\texttt{merge}(i'_1, i'_2)]\!] \mid (i'_1, s) \in S', (i'_2, s) \in S', s \neq \bot \wedge i'_1 \neq i'_2\} \\
&\qquad \text{let } m_F = \{[\![\texttt{merge}(i'_1, i'_2)]\!] \mid (i_2, n, i'_1) \in F', (i_2, n, i'_2) \in F', i'_1 \neq i'_2\} \\
&\qquad \text{let } m_C = \{[\![\texttt{merge}(i'_1, i'_2)]\!] \mid (i_a, i_f, i'_1) \in C', (i_a, i_f, i'_2) \in C', i'_1 \neq i'_2\} \\
&\qquad \text{let } m_{C_f} = \{[\![\texttt{merge}(I_{f_1}(k), I_{f_2}(k))]\!] \mid (i_a, i_f, I_{f_1}) \in C'_f, (i_a, i_f, I_{f_2}) \in C'_f, \\
&\qquad\qquad\qquad I_{f_1} \neq I_{f_2}, k \in \mathit{domain}(I_{f_1})\} \\
&\qquad \text{let } P_m = m_s \cup m_S \cup m_F \cup m_C \cup m_{C_f} \\
&\qquad \text{let } E' = ((S', F', C', C'_f, R'), P_m \cup P', D') \\
&\qquad \text{assert } \mathit{shape\_invariants}(E'_N, s', i_2) \\
&\qquad \text{for each } [\![\texttt{merge}(i'_1, i'_2)]\!] \in P_m \\
&\qquad\quad E' := \mathit{constrain}([\![\texttt{merge}(E'_{N_{R^*}}(i'_1), E'_{N_{R^*}}(i'_2))]\!], E') \\
&\qquad \text{return } E'
\end{aligned}
$$

The top-level merge operation starts with one trivial case: if i_1 and i_2 are already equal, then no work is required to substitute i_1 with i_2. This case just returns the existing E.

In the case where work is necessary, first a combined shape s' is computed alongside any shape-driven follow-up merges m_s. If this succeeds, then we remove the mapping from i_1 to its shape in E_{N_S}, remap i_2 to the new shape s', add the substitution record (i_1, i_2) to E_{N_R}, and rewrite all the components of the program state excluding R such that all occurrences of i_1 are replaced by i_2.

All of these operations combined give the components of a successor environment E', which is locally valid as long as the *shape_invariants* assertion passes. E' may however still contain uniqueness violations. None of the components mention i_1 any more except for R', which records that i_1 has been substituted for i_2. Additionally, i_2 must now have shape s', which by correctness of *next_shape* must satisfy shape sequencing from both i_1 and i_2's previous shapes.

What remains is to fix the uniqueness violations via *follow-up merges*. Each of m_s, m_F, m_C and m_{C_f} contained in P_m represent additional constraints required to remove any remaining uniqueness violations. Each of these sets of constraints are directly based on resolving inversions of the conditions listed by defs. 4.2.2 to 4.2.5, with the expectation that if each of those follow-up merges can be resolved then by definition no uniqueness violations can be present. Note that, unlike other constraints that can be added to the program state during program execution, these constraints must be executed immediately – environment ordering is only defined over environments that have no uniqueness violations, so the result of applying the main merge constraint must include that of all follow-up merges. Note also that the merge constraint semantics themselves are formulated to avoid relying on interpretations of the environment that are invalidated by uniqueness violations: only the always-valid interpretation $E_{N_{S^1}}$ is referenced. This means that follow-up merges can safely follow the same semantics as regular merges.

Termination argument The merge constraint is recursive via its follow-up merges, so we need to argue that it should terminate. Based on the constraint's structure a simple argument can be made:

- If the identifiers i_1 and i_2 are equal, then the constraint trivially produces the same program state, terminating immediately.
- In all other cases, every successful application of the merge operations reduces the number of identifiers in the environment by one: in all successful cases, i_1 is unconditionally removed from all further program evaluation (except E'_{N_R}, but that is not actively used by evaluation), including mentions of the merge constraint itself. Once some merge $\texttt{merge}(i_1, i_2)$ has executed, it will be rewritten to $\texttt{merge}(i_2, i_2)$, meaning that attempting to execute it again will have no effect.

  Since merge constraints do not generate any fresh identifiers, this means that no matter how many or what kind of follow-up merges are generated, there is a finite, decreasing set of identifiers available to merge. If that set is exhausted without errors, then the one remaining identifier must be equal to itself, meaning every remaining follow-up merge operation must have been rewritten to merge that single identifier with itself and must have no effect.

  Note that when referencing follow-up merges, they are mapped over the most up-to-date rewrite history $E'_{N_{R^*}}$, ensuring that follow-up merges cannot use identifiers that other follow-up merges have added to E'_{N_R}.

Increasing argument The merge operation can be argued increasing as defined by def. 4.6.4 by considering two cases:

- In the case that the identifiers being merged are equal, the merge operation does nothing, so by reflexivity of environment sequencing the application of a merge constraint must be increasing in this case.
- In the case that the identifiers i_1 and i_2 being merged are not equal, the merge constraint's execution is increasing by a combination of E_{N_R} and the construction of *next_shape*. Assuming that *next_shape* finds a shape that satisfies shape sequencing from both i_1 and i_2, the only sequencing conditions that may not be satisfied are due to uniqueness violations preventing a well-defined identifier mapping from existing.

  Having shown that merge constraint application is increasing when no follow-up merges are involved, we can use induction: having proven the base case, assume that all follow-up merges are increasing. Since follow-up merges must by definition resolve all uniqueness violations while being otherwise assumed correct, the environment after applying all uniqueness violations must form a valid sequence with the starting environment.

Monotonicity argument The merge operation can be argued monotonic as defined by def. 4.6.5 by splitting into cases depending on when the merge constraint executed.

- If a merge constraint p has been executed in neither E nor E', then both for E and E', the identifier i_1 will be rewritten to i_2, with a record of this rewrite added to R. Since the rewrites and addition to R are the same, so far the sequencing relation between environments must be preserved.

  The other local part of the merge operation is the application of *next_shape*, where the shapes given to i_2 in each case need considering. We know initially that separately the original shapes of i_1 and i_2 form a valid sequence from E to E'. We also know from *next_shape* that the new shapes given to i_2 must be the least possible shapes that follow both i_1 and i_2. Since each next shape is the least possible next shape this preserves the original relationship between pairs (i_1, i_2) across E and E'.

  The above is sufficient for cases where there are no follow-up merges necessary, with simple substitution being sufficient to produce a valid F, C and C_f. We can extend this to all merge constraint executions by the same induction as the increasing argument: we can assume that the follow-up merges are monotonic. From the precondition $E \sqsubseteq E'$ we know that there are at least as many relations involving i_1, i_2 in

E' as in E, which means that performing the same merge in E' will require at least an equivalent set of follow-up merges to performing that merge in E. This means that the resulting P will satisfy the subset-based sequencing constraint on the set of constraints. Combining this with each of the immediately executed follow-up merges being assumed to be monotonic (and therefore increasing the program state in the same way irrespective of order), then merge constraint execution is monotonic in this case.

- It is not possible for a merge constraint p to have already executed in E but not E'. If it had executed in E, then E_{N_R} would have one more element than E'_{N_R}, which is not allowed by environment sequencing.

- If a merge constraint p has not executed in E but has in E', then E_{N_R} will grow by an element that is already in E'_{N_R}. By the same argument as in the first case, *next_shape* gives the least next shape, which must be sequenceable before the next shape already in E'. Lastly, by a similar inductive argument to the first case, each recursive follow-up merge added to E_P must already have an equivalent in E'_P. Since we assume their execution to be monotonic, we know that each follow-up merge will increase the program state in a consistent way regardless of order, meaning the overall merge operation must be monotonic in this case too.

- If a merge constraint p has already executed in both E and E', then trivially the monotonicity condition can be rewritten $E \sqsubseteq E' \implies E \sqsubseteq E'$, which is a tautology.

Shape constraint

$$
\begin{aligned}
&\mathit{constrain}([\![\texttt{shape}(s, i_r)]\!], E) = \\
&\quad \text{let } (s', m_s) = \mathit{next\_shape}(s, E_{N_{S^1}}(i_r)) \\
&\quad \text{let } S' = \{(i, s) \mid (i, s) \in E_{N_S}, i \neq i_r\} \cup \{(i_r, s')\} \\
&\qquad \text{let } m_S = \{[\![\texttt{merge}(i'_1, i'_2)]\!] \mid (i'_1, s) \in S', (i'_2, s) \in S', s \neq \bot \wedge i'_1 \neq i'_2\} \\
&\quad \text{let } F' = \text{if } s' = \text{object } O \\
&\qquad\quad E_{N_F} \cup \{(i_r, n, i_f) \mid (n, i_f) \in O\} \\
&\qquad \text{else } E_{N_F} \\
&\quad \text{let } E' = ((S', F', E_{N_C}, E_{N_{C_f}}, E_R), m_s \cup E_P, E_D) \\
&\quad \text{assert } \mathit{shape\_invariants}(E'_N, s', i_r) \\
&\quad \text{for each } [\![\texttt{merge}(i_1, i_2)]\!] \in (m_s \cup m_S) \\
&\qquad E' := \mathit{constrain}([\![\texttt{merge}(E'_{N_{R^*}}(i_1), E'_{N_{R^*}}(i_2))]\!], E') \\
&\quad \text{return } E'
\end{aligned}
$$

The shape constraint ensures that i_r has a shape that forms a valid sequence with shape s. It does this in a very similar way to the merge constraint, relying on *next_shape* to find a new shape for i_r that both forms a valid sequence with i_r's existing shape and the shape s, if it exists.

As a special case, if the resulting shape is an object shape then it is possible that E_{N_F} does not contain fields that are listed in the object shape. This would happen if i_r were $\bot$-shaped and the shape constraint were increasing it to an object shape, in which case i_r would need to gain fields matching the object shape's. To fix the problem, we take the union of E_{N_F} and the appropriate field relation instances. Either this does nothing, if the records are already present, or the appropriate instances will be added.

Because some shape changes also come with follow-up merges, the shape constraint relies on the follow-up merge mechanism defined by the merge constraint. In fact, this means the shape constraint can share almost all the same proof logic as the merge constraint, albeit not necessarily needing all the cases that merging two identifiers does. The key difference is that the base case does not include a substitution, so it does not increase E_{N_R}. Instead, the base case only increases $E_{N_{S^1}}(i_r)$ and, in case of s being an object shape, E_{N_F}. The initial set of possible follow-up merges is also smaller, since as we are not directly merging two identifiers there are fewer possible uniqueness violations: we can only cause them for fields and values whose shapes have uniqueness rules.

Projection

$$
\begin{aligned}
&\mathit{constrain}([\![\texttt{project}(i, n, i_r)]\!], E) = \\
&\quad \text{if } (i, n, i_f) \in E_{N_F} \\
&\qquad \mathit{constrain}([\![\texttt{merge}(i_f, i_r)]\!], (E_N, E_P \cup \{[\![\texttt{merge}(i_f, i_r)]\!]\}, E_D)) \\
&\quad \text{else if } N_{S^1}(i) = \bot \\
&\qquad ((E_{N_S}, E_{N_F} \cup \{(i, n, i_r)\}, E_{N_C}, E_{N_{C_f}}, E_R), E_P \cup \{[\![\texttt{merge}(i_r, i_r)]\!]\}, E_D) \\
&\quad \text{else } \textbf{error}
\end{aligned}
$$

Projection ensures i_r refers to i's field with name n. There are two possible cases:

- If E_{N_F} already contains a record of i having a field named n, then the field value i_f needs to be merged with i_r in order for the value i_r to be the same as the field value i_f. The merge is executed immediately in order for the constraint application to remain monotone, since otherwise i_r would only eventually be the same value as i_f, violating the postcondition of executing the merge in the interim.
- Otherwise, if i has shape $\bot$, then a new instance of the field relation must be added. Note that this cannot cause uniqueness violations, since this case can only execute if there exists no conflicting instance of the field relation. The seemingly redundant merge constraint is added for formal convenience, as will be demonstrated below.

Trying to project from a value that neither has the field already nor is $\bot$-shaped can only be an error.

Increasing argument Projection must be increasing, since in either successful case it can only increase program state. If there is already a record of a field i_f in value i, then a merge constraint is added and immediately executed, resulting in an increase to E_P alongside any other increases caused by the merge constraint. Otherwise, if i is $\bot$-shaped, a new field record is added to E_{N_F} and an equivalent (but redundant in this case) merge constraint is added to E_P.

Monotonicity argument For monotonicity, we must argue that for any E or E' a given instance of the projection constraint has to execute consistently in both. This has three cases:

- If there is a record $(i, n, i_f) \in E_{N_F}$, then by the sequencing relation, there must also be a corresponding one in E'_{N_F}. This means that for either E or E', any given projection constraints must add equivalent merge constraints to both E_P and E'_P, and in both cases immediately execute them to equivalent effect (from the monotonicity of the merge constraint).

- If there is a record $(i, n, i_f) \in E'_{N_F}$ but not E_{N_F}, then in E, i must be $\bot$-shaped or the program will fail. This means that in E that record will be added alongside a redundant merge constraint between i_r and itself, while in E' an equivalent merge constraint between i_f and i_r will be added and immediately executed, ensuring that (i, n, i_r) is in the resulting environment's fields relation rather than any other (i, n, i_f) that was there originally.

  As a result, in both cases the resulting program states will contain the appropriate fields relation and an already-executed merge constraint that has been rewritten to be between the field value and itself.

- If there is no relevant record in E_{N_F} or E'_{N_F}, then as long as i is $\bot$-shaped an appropriate record and a merge constraint from i_r to i_r will be added in both cases.

Function call

$$
\begin{aligned}
&\mathit{constrain}([\![\texttt{call}(i_f, i_a, i_r)]\!], E) = \\
&\quad \text{if } (i_a, i_f, i'_r) \in N_C \\
&\qquad \mathit{constrain}([\![\texttt{merge}(i'_r, i_r)]\!], (E_N, E_P \cup \{[\![\texttt{merge}(i'_r, i_r)]\!]\}, E_D)) \\
&\quad \text{else} \\
&\qquad \text{assert } N_S(i_f) = \bot \vee N_S(i_f) = \text{function } f; i_c \\
&\qquad \text{let } N' = (E_{N_S}, E_{N_F}, E_{N_C} \cup \{(i_a, i_f, i_r)\}, E_{N_{C_f}}, E_{N_R}) \\
&\qquad \text{return } (N', E_P \cup \{[\![\texttt{merge}(i_r, i_r)]\!]\}, E_D \cup \{[\![i_r := i_f(i_a)]\!]\})
\end{aligned}
$$

Function calls are structurally similar to projections, but applying to function calls instead.

- The same as for projection, if there is already a function call record in E_{N_C}, then merge the recorded result value with i_r. As with projection, the merge constraint is executed immediately in order to preserve monotonicity of the call constraint.

- If there is no record in E_{N_C}, then as long as i_f has one of the callable shapes – $\bot$ or a function – a record should be added to E_{N_C} and a deferral should be added to E_D. The record in E_{N_C} takes care of recording that the call has been started, whereas the deferral in E_D takes care of ensuring that the call eventually completes.

Increasing argument Function call execution must always be increasing, since in either branch, one or both of E_P and E_D are increased in ways that satisfy (STATE-LEQ).

Monotonicity argument As with projection, for monotonicity we show that for any E and E' such that $E \sqsubseteq E'$, applying the same function call constraint produces a consistent result.

- If there is no existing function call record in either E_{N_C} or E'_{N_C}, then both E and E' will be increased in the same way, ensuring that E_D contains the appropriate function call deferral, that E_P contains a merge from i_r to i_r, and that C contains the relevant fresh calls relation. In any such case, the consistent additions must satisfy (STATE-LEQ).

- If there exists a record $(i_a, i_f, i'_r) \in E'_{N_C}$ but not in E_{N_C}, then both cases are kept consistent by the redundant merge from i_r to i_r being added to E. For E application of the call constraint will add the function call deferral to D, the function call record (i_a, i_f, i_r) to E_{N_C}, and a redundant merge constraint from i_r to the function call result (also i_r). For E' application of the call constraint will add and immediately execute a merge constraint from i_r to the function call result. Since i'_r must have been added by the first case of some other function call constraint's execution (only function call constraints can add to C), merging i_r with i'_r will ensure that both a function call record and a deferral in terms of i_r exist, ensuring that the result is consistent with adding a fresh record to E_{N_C}.

- If there exists a record $(i_f, i_a, i'_r) \in E_{N_C}$ then by the sequencing relation there must be a corresponding record in E'_{N_C}. In that case, both executions for E and E' ensure a merge constraint from i_r to i'_r will be added to P and immediately executed. By the monotonicity of the merge constraint each result should be consistent with the other.

Defer

$$\textit{constrain}([\![\texttt{defer}(d)]\!], E) = \\ (E_N, E_P, E_D \cup \{d\})$$

The defer constraint is quite simple: it just ensures that E_D contains the deferral d. Via the mechanism in $\textit{step}_k$, this ensures that the deferral must execute eventually.

Increasing argument Ensuring d is in E_D trivially satisfies (STATE-LEQ) since this will leave E_D either the same or greater by one element.

Monotonicity argument Ensuring d is in E_D also trivially satisfies monotonicity.

- If for E the defer constraint has no effect then, it must also have no effect in E', as by state sequencing if the deferral d is in E_D then $d \in E'_D$ must also be true. In this case monotonicity holds by reflexivity of state sequencing.
- If for E the defer constraint has an effect but for E' it does not, then d must already be in E'_D. Therefore, adding d to E_D will not invalidate state sequencing.
- If the defer constraint has an effect in both E and E', then it must have the same effect in both: adding d to E_D and E'_D. Increasing both E_D and E'_D in the same way also preserves state sequencing.

Note that the logic presented here serves as an example of the unstated, underlying logic for all other reasoning about set unions.

4.6.5 Deferrals

This section describes the precise semantics of deferral execution via the piecewise-defined function $\textit{undefer}(d, E) = E'$, as well as giving arguments for increasingness and monotonicity of that execution as given by defs. 4.6.6 and 4.6.7.

$$E \sqsubseteq \textit{undefer}(d, E) \tag{4.6.6}$$

$$E \sqsubseteq E' \implies \mathit{undefer}(d, E) \sqsubseteq \mathit{undefer}([i \to E'_{N_{R^*}}(i)]d, E') \tag{4.6.7}$$

Unlike constraints, there is a shortcut to proving these properties for all deferrals other than function call deferrals: stable information. If a deferral's execution relies on stable information and its only effect is adding constraints, then it must be both increasing and monotonic. Definition 4.6.8 gives a formal definition of the property involved: once some deferral d has been applied (after all its waited-on values became known), then applying it again should not affect the program state. If $\mathit{waits\_on}$ is not purely a function of d then we assume it must produce an increasing set of identifiers under renaming.

$$\begin{aligned} &\mathit{undefer}(d, E) \sqsubseteq E' \wedge (\forall i \in \mathit{waits\_on}(E_N, d), E_{N_{S^1}}(i) \neq \bot) \\ &\quad \implies \mathit{undefer}(d, E') = E' \end{aligned} \tag{4.6.8}$$

Note that in both arguments below as well as all of the semantics for deferrals we assume that $\forall i \in \mathit{waits\_on}(E_N, d), E_{N_{S^1}}(i) \neq \bot$ since step_k could not apply a given deferral otherwise.

Increasing argument (stable) Thanks to the condition that deferrals acting on stable information must only add constraints, this argument is quite easy to make. Consider the application of some deferral d. If program state E has not had d applied yet, then $\mathit{undefer}(d, E)$ can only increase E_P, which satisfies (STATE-LEQ). If program state E has had d applied, then by def. 4.6.8 the condition can be rewritten to $E \sqsubseteq E$, which holds by reflexivity of the sequencing relation.

Monotonicity argument (stable) Considering valid applications of some deferral d satisfying the stable information criteria leaves us with two cases:

- If d has already been applied in E, then the entire monotonicity condition can be rewritten to $E \sqsubseteq E' \implies E \sqsubseteq E'$ thanks to stable information. This statement is tautologically true.
- If d is applied in E' but not E, then we have $E \sqsubseteq E' \implies \mathit{undefer}(d, E) \sqsubseteq E'$. Since we also know that $\mathit{undefer}(d, E)$ can only add constraints, then since $E_P \subseteq [i \to E'_{N_{R^*}}(i)]E'_P$ and from d having already been applied we know E'_P must already contain the constraints relevant to applying d, then we can safely state $\mathit{undefer}(d, E) \sqsubseteq E'$.

Function call

$$
\begin{aligned}
&\mathit{undefer}([\![i_r := i_f(i_a)]\!], E) = \\
&\quad \text{if } (i_a, i_f) \in \mathit{domain}(E_{N_{C_{f^1}}}) \\
&\qquad \text{let } m = [\![\texttt{merge}(i_r, E_{N_{C^1}}(i_a, i_f))]\!] \\
&\qquad \text{return } \mathit{constrain}(m, (E_N, E_P \cup \{m\}, E_D)) \\
&\quad \text{else} \\
&\qquad \text{let function } \backslash\backslash n => e_b; i_c = E_{N_{S^1}}(i_f) \\
&\qquad \text{let } O = \{(n, \mathbf{fresh\_id}_n) \mid n \in \mathit{closure}(\backslash\backslash n => e_b)\} \\
&\qquad \text{let } \gamma = \{(n, i_a)\} \cup O \\
&\qquad \text{let } (P, I_f, i_b) = \mathit{trans}(\gamma, e_b) \\
&\qquad \text{let } S' = \{(i, \bot) \mid i \in \mathit{range}(I_f) \cup \mathit{range}(O)\} \cup E_{N_S} \\
&\qquad \text{let } N' = (S', E_{N_F}, E_{N_C}, \{(i_a, i_f, I_f)\} \cup E_{N_{C_f}}, E_R) \\
&\qquad \text{return } (N', \{[\![\texttt{shape}(\text{object } O, i_c)]\!], [\![\texttt{merge}(i_r, i_b)]\!]\} \cup P \cup E_P, E_D)
\end{aligned}
$$

The function call deferral performs the steps necessary to call the known function value i_f. It is the only way for fresh identifiers to be added to the environment, so its operation takes a little more care than other deferrals.

In particular, to avoid repeated executions of the deferral generating an infinite growth of fresh identifiers $E_{N_{C_f}}$ is used as a proxy for whether the deferral has already been executed or not. Since the execution path that introduces fresh identifiers to the environment also adds the pair (i_a, i_f) to $\mathit{domain}(E_{N_{C_{f^1}}})$, if (i_a, i_f) is already present then we know that the function body has already been expanded for this function call. Instead of doing that again, we instead ensure there is a merge constraint between i_r and the existing recorded result.

The other path is to actually use *trans* to expand the function body into fresh identifiers and constraints. First the correct scope γ is constructed, containing all of the fields in the closure object alongside a binding from the argument name to i_a. The closure object is not waited-on by function call deferrals so we use an indirect way to get its fields: we generate entirely fresh fields in O based on the function expression's free variables and add a constraint that i_c has an object shape with those fields. This means the function call deferral can end up executing before the shape constraint forcing i_c to have the right object shape, removing what would otherwise be an order dependent pair of constraint and deferral (the closure object and the call deferral).

In this context, *trans* generates a set of constraints P, a fingerprint I_f and a result identifier i_b. Each identifier in I_f is introduced to the environment with shape $\bot$, I_f is added to $E_{N_{C_f}}$ and a merge constraint is added between the function body's result i_b and the value i_r.

Increasing argument This can be argued in two cases, one for each branch:

- The first branch calls through to the semantics of the merge constraint, so this branch holds by the increasing property of the merge constraint.
- The second branch is increasing by definition: it introduces a group of fresh identifiers, a set of fresh constraints, an extra record in $E_{N_{C_f}}$, and an additional merge and shape constraint. The fresh identifiers satisfy (ENV-LEQ) by not appearing at all in E_N, while the addition to $E_{N_{C_f}}$ is valid due to the branch condition implying the absence of any existing record in $E_{N_{C_f}}$ that would conflict. The addition of the shape constraint on i_c is increasing by the usual argument for adding constraints.

Monotonicity argument This can be shown in three cases, the fourth permutation being impossible due to environment sequencing:

- If neither E nor E' satisfy $(i_a, i_f) \in \mathit{domain}(E_{N_{C_{f^1}}})$, then both cases will perform the second branch, adding a group of fresh identifiers and constraints. In each case, the set of constraints and identifiers must be equivalent, since $\mathit{trans}(\gamma, e_b)$ depends only on the function expression embedded in i_f's shape and nothing else. Looking into the definition of (STATE-LEQ), we can argue that it is possible to choose an R_s that maps from one set of fresh identifiers to the other. From there, all the additions to E and E' satisfy the sequencing relation: the constraints satisfy the subset relation under renaming, and the fresh identifiers satisfy identifier sequencing by all having shape $\bot$ and no other topology.
- If E' satisfies $(i_a, i_f) \in \mathit{domain}(E_{N_{C_{f^1}}})$, but E does not, then we have to show that the immediately-executed merge constraint on E' has an equivalent effect to everything added to E.

  For $(i_a, i_f) \in \mathit{domain}(E_{N_{C_{f^1}}})$ to be satisfied, there must have already been some execution of a function call deferral $[\![i_f(i_a) = i'_r]\!]$. Knowing this, if we execute a merge operation between i_r and i'_r then the resulting program state will be rewritten

to be as if that deferral were executed as $[\![i_f(i_a) = i_r]\!]$. This ensures that the result for E' is equivalent to the full operation applied to E.

- If both E and E' satisfy $(i_a, i_f) \in \mathit{domain}(E_{N_{C_{f1}}})$, then both cases directly execute a merge constraint, the results being equivalent by the monotonicity of the merge constraint.

Conditional branching

$\mathit{undefer}([\![\texttt{if } i_c \texttt{ then } P_t \texttt{ else } P_f]\!], E) =$
  $E_{N_{S^1}}(i_c)$ match
    case prim `true` $\rightarrow (E_N, P_t \cup E_P, E_D)$
    case prim `false` $\rightarrow (E_N, P_f \cup E_P, E_D)$

The conditional branching deferral allows for conditional branching. i_c must be one of the primitive shapes `true` or `false`. If the shape is `true`, then the constraints P_t are emitted, otherwise the constraints P_f are emitted.

Definition 4.6.8 is satisfied, since by shape sequencing it is impossible for a primitive shape to change: any re-applications will consistently re-perform the union of the same P_t or P_f with E_P, leading to an unchanged program state.

Negation

$\mathit{undefer}([\![i_r :=\!!i_c]\!], E) =$
  let $s' = E_{N_{S^1}}(i_v)$ match
    case prim `true` $\rightarrow$ prim `false`
    case prim `false` $\rightarrow$ prim `true`
  return $(E_N, \{[\![\texttt{shape}(s', i_r)]\!]\} \cup E_P, E_D)$

This deferral implements the unary operation of boolean negation. Assuming the shape is primitive boolean `true` or `false`, the deferral performs a standard boolean negation. Once it has done this, it propagates the result by emitting a shape constraint that requires i_r to have the negated boolean shape s'.

Definition 4.6.8 is satisfied by shape sequencing: since $E_{N_{S^1}}(i_v)$ has to be a primitive shape any re-applications of this kind of deferral will yield the same set union as the first, and performing the same set union twice has no effect the second time.

Type construction

$$
\begin{aligned}
&\mathit{undefer}([\![i_r := \texttt{type}(i_t)]\!], E) = \\
&\quad \text{let prim } p = E_{N_{S^1}}(i_t) \\
&\quad \text{return } (E_N, \{[\![\texttt{shape}(\text{prim } \texttt{type } p, i_r)]\!]\} \cup E_P, E_D)
\end{aligned}
$$

Type construction can also be viewed as a unary primitive operator, albeit a specialised one.

The value with identifier i_t is assumed has the shape of a primitive string. This string shape is used to create a type-shaped value by constraining the value i_r to have a type shape corresponding to the primitive string shape of the value i_t.

Definition 4.6.8 is satisfied for this deferral by exactly the same logic as negation.

Primitive binary operation

$$
\begin{aligned}
&\mathit{undefer}([\![i_r := i_1\ b\ i_2]\!], E) = \\
&\quad \text{let prim } p_1 = E_{N_{S^1}}(i_1) \\
&\quad \text{let prim } p_2 = E_{N_{S^1}}(i_2) \\
&\quad \text{let } p_r = p_1\ b\ p_2 \\
&\quad \text{return } (E_N, \{[\![\texttt{shape}(\text{prim } p_r, i_r)]\!]\} \cup E_P, E_D)
\end{aligned}
$$

Binary operations on primitive values are generalised in order to consolidate the cases for each possible binary operation. Binary operations use multiple deferral wait on both i_1 and i_2, since both shapes must be primitive shapes in order for such an operation to make sense.

The overall semantics of primitive binary operations can apply to essentially arbitrary primitive operations as long as the result is only based on the combination of two primitive shapes. In this instance, b is assumed to be one of the operations defined in the formal grammar, but really, b could be any operation over primitive shapes. If the operation is well-defined over p_1 and p_2, then the result is communicated by constraining i_r to have shape prim p_r.

Definition 4.6.8 is satisfied for this deferral because i_1 and i_2 must both have primitive shapes which by shape sequencing can never change once they are set. The only possible

execution of the deferral adds a shape constraint based on those two shapes, so that shape constraint must in turn be consistent in future applications of the deferral. This means that for any subsequent execution of the deferral, that constraint must already be present, resulting in an unchanged environment.

Equality comparison

Equality comparison is perhaps the most subtle deferral, owing to how it satisfies the stable information requirement.

Given any pair of identifiers, it's technically easy to tell if they refer to the same value: are the identifiers equal? The problem is that, given the merge operation, such a result is unstable – two values that were once not the same could later become the same, potentially invalidating the results of previous equality comparisons.

A simple way to ensure a stable result for equality comparison is to require that the entire reachable subgraphs of the values being compared are stable: either the values are both primitive shaped, or they are object shaped with only known fields, or they are function shaped with (known) closure objects containing only known values. Note that this definition ignores the calls relation – calling a function does not intrinsically change it, and it would be very confusing to the programmer if adding a call to a function implicitly changed the result of comparing that function to another.

$$
\begin{aligned}
&\mathit{stably\_reachable\_subvalues}(V, N, i) = \\
&\quad \text{if } i \in V \text{ then } \emptyset \\
&\quad \text{else } N_S(i) \text{ match} \\
&\qquad \text{case } \bot \to \{i\} \\
&\qquad \text{case prim } p \to \{i\} \\
&\qquad \text{case object } O \to \{i\} \cup \bigcup_{(n, i_f) \in O} \mathit{stably\_reachable\_subvalues}(V \cup \{i\}, N, i_f) \\
&\qquad \text{case function } f; i_c \to \{i\} \cup \mathit{stably\_reachable\_subvalues}(V \cup \{i\}, N, i_c)
\end{aligned}
$$

The function $\mathit{stably\_reachable\_subvalues}(V, N, i)$ helps implement this rule by returning a set of the identifiers that are *stably reachable* from the value with identifier i, including i itself. This is the function used in the definition of *waits_on* for this deferral. If all of the stably reachable identifiers have a shape other than $\bot$, then no further changes can occur

to that subgraph and we can safely compare two identifiers in that subgraph for equality. V is used to handle cycles – if an identifier is being reached from itself, then not only can the value itself not be $\bot$-shaped but sibling recursions will be taking care of finding any other reachable unknowns, so we can break the cycle by returning $\emptyset$.

Our treatment of each possible shape is in the interest of providing an increasing set of all identifiers whose shape must not be $\bot$ in order for the deferral to give a stable result. We can explain this case by case for each shape:

- $\bot$-shaped values can be returned on their own, since we need no further evidence that we need to wait for at least one value's shape to not be $\bot$.
- Primitive shapes have no fields, and as mentioned above we are not counting the calls relation. Since we know from shape sequencing that primitive shapes can only be followed by exactly the same shape, it is safe to just return a primitive-shaped value's identifier.
- Object shapes have a fixed set of fields: an object shape can only be followed by an object shape with fields that also follow a valid sequence. This means that we can return the object-shaped value's identifier and safely recurse over this fixed set of field values, applying a separate instance of this correctness argument to those.
- Function shapes have no fields and have one closure object. Again, function shapes can only be followed by another function shape whose closure object recursively follows a valid sequence, so we can safely return the function-shaped value's identifier and recurse on the closure object.

Knowing all this, we can describe how `undefer` implements equality comparison.

$$
\begin{aligned}
&\mathit{undefer}(\llbracket i_r := i_1 == i_2 \rrbracket, E) = \\
&\quad \text{let } p_r = \text{if } i_1 = i_2 \text{ then } \texttt{true} \text{ else } \texttt{false} \\
&\quad \text{return } (E_N, \{\llbracket \texttt{shape}(\text{prim } p_r, i_r) \rrbracket\} \cup E_P, E_D)
\end{aligned}
$$

The overall operation simply compares i_1 and i_2 for equality, since by the *waits_on* condition, we know that both i_1 and i_2 must already refer to stable subgraphs. By the shape uniqueness defined in def. 4.2.2, if the identifiers are equal, then so are any reachable values, so we can decide equality by comparing the identifiers i_1 and i_2. This result is then communicated by constraining i_r's shape.

Definition 4.6.8 is satisfied because both i_1 and i_2 must refer to stable subgraphs, and otherwise only a shape constraint on i_r is added to E_P.

4.7 File output overlay

The core language on its own is very flexible and could plausibly have a variety of uses. For TreeGen's intended purpose though, we need to generate text files by consuming documents generated by TreeGen. This section documents how to extend the core language's semantics to support this, illustrating in the process how this process could be repeated for any extension whose semantics can be expressed via loosely coupled additional deferrals and constraints.

For file output, core TreeGen is extended with one piece of global state and a new construct `file_generate(name, contents)`. The new global is called the *file space* $\mathcal{F}$, which uniquely maps file names to file contents. It is added to the existing 3-tuple handled by the *eval* function, meaning that a new evaluation that works with files $eval_{\mathcal{F}}(e) = (N, \mathcal{F}, i_r)$ can be defined that produces a file space $\mathcal{F}$ in addition to the regular *eval*'s outputs.

$eval_{\mathcal{F}}(e) =$
  let $(P_0, I_f, i_r) = trans(\emptyset, e)$
  $E := ((\{(i, \bot) \mid i \in range(I_f)\}, \emptyset, \emptyset, \emptyset, \emptyset), P_0, \emptyset, \emptyset)$
  while $\exists k \in \mathbb{N},\ E \neq step_k(E)$
    choose $k \in \mathbb{N},\ E := step_k(E)$
  assert $\forall d \in E_D,\ \forall i \in waits\_on(d),\ E_{N_{S^1}}(i) \neq \bot$
  return $(E_N, E_{\mathcal{F}}, E_{N_{R^*}}(i_r))$

This new $eval_{\mathcal{F}}$ simply adds $\mathcal{F}$ to the program state, extending program state sequencing to include $E_{\mathcal{F}} \subseteq E'_{\mathcal{F}}$. Since $\mathcal{F}$ does not involve identifiers at all, no special provision for substitutions is needed. Conveniently, all the other constraints and deferrals can be considered to pass $\mathcal{F}$ along without modification.

In order to actually operate on $\mathcal{F}$, some additional constructs can be given: a grammar for the new expression, alongside a corresponding deferral and set of deferral semantics. `file_generate(name, contents)` is an expression that, assuming `name` and `contents` will eventually refer to primitive strings, will ensure that $\mathcal{F}$ contains a pair mapping the string `name` to the string `contents`. As with the rest of TreeGen's semantics, the operation is idempotent: re-creating a file with exactly the same contents has no effect, whereas recreating a file with the same name but different contents is an error in the same way that requiring `effect 1 = 2` is an error.

First, we can give the operation the additional grammar entry one might expect. In actual practice, such a function would be appropriately namespaced by being nested inside objects accessible from `predef` as in `predef/file/generate`, but formally all that is needed is the following grammatical addition.

⟨*expression*⟩ ::= '`file_generate`' '`(`' ⟨*expression*⟩ '`,`' ⟨*expression*⟩ '`)`'

Since this operation relies on primitive values, a deferral will be needed, like for primitive binary operations, to safely access those values.

⟨*deferral*⟩ ::= '`file_generate`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*identifier*⟩ '`)`'

Now that all of these definitions are given, we can build additional cases for *trans* and *undefer*.

$$
\begin{aligned}
&\mathit{trans}(\gamma, [\![\texttt{file\_generate}(e_n, e_c)]\!]) = \\
&\quad \text{let } (P_n, I_{f_n}, i_n) = \mathit{trans}(e_n) \\
&\quad \text{let } (P_c, I_{f_c}, i_c) = \mathit{trans}(e_c) \\
&\quad \text{let } P = \{[\![\texttt{defer}(\texttt{file\_generate}(i_n, i_c))]\!], [\![\texttt{shape}(\text{object } \emptyset, \mathbf{fresh\_id}_r)]\!]\} \\
&\quad \text{return } (P_n \cup P_c \cup P, (I_{f_n} \ldots, I_{f_c} \ldots, \mathbf{fresh\_id}_r), \mathbf{fresh\_id}_r)
\end{aligned}
$$

The translation of `file_generate` into constraints mirrors that for primitive binary operations, merging the constraints from the two sub-expressions and adding a deferral to wait on i_n and i_c. Unlike binary operations, however, the result in this case is irrelevant, so a fresh result value is constrained to an empty object shape as a kind of unit value.

$$
\begin{aligned}
&\mathit{undefer}([\![\texttt{file\_generate}(i_n, i_c)]\!], E) = \\
&\quad \text{let prim } n = E_{N_{S^1}}(i_n) \\
&\quad \text{let prim } c = E_{N_{S^1}}(i_c) \\
&\quad \text{assert } \nexists (n, c') \in E_{\mathcal{F}},\ c \neq c' \\
&\quad \text{return } (E_N, E_P, E_D, \{(n, c)\} \cup E_{\mathcal{F}})
\end{aligned}
$$

The deferral semantics for file generation just add (n, c) to $\mathcal{F}$, with an assertion that there does not exist a conflicting record giving a file n contents c'.

Since the file generation deferral adds strictly no constraints and only operates on $\mathcal{F}$, we can use a simple extension of the stable information argument to prove that it is both increasing and monotonic. To start, we rely on the stable information arguments to exclude all parts of the program state except $\mathcal{F}$. This gives a simple proof in terms of only $\mathcal{F}$.

Increasing argument We can argue the deferral execution is increasing by noticing that each deferral performs a union between $E_{\mathcal{F}}$ and $\{(n, c)\}$. This union can only increase $\mathcal{F}$ in terms of the subset relation, so (non-crashing) application of the deferral must be increasing.

Monotonicity argument For monotonicity, we can argue that, aside from the assertions, all applications of the file generation deferral have a consistent effect: ensuring (n, c) is in $E_{\mathcal{F}}$. This gives us this effective relation for monotonicity $\mathcal{F} \subseteq \mathcal{F}' \implies \mathcal{F} \cup \{(n, c)\} \subseteq \mathcal{F}' \cup \{(n, c)\}$, which holds via fundamental set axioms.

Chapter 5

Reducing TreeGen to core TreeGen

In order to build the core TreeGen language used in chapter 4 back up to the full TreeGen language, some layers of translation are required. This chapter describes how to reduce the full TreeGen language to the core language via term rewriting. This will be done by first giving the full language grammar alongside related lexical details, then by giving rewrites that progressively remove more and more features from the language until we reach core TreeGen.

Note that the rewrites given do not describe a single unambiguous algorithm – a reasonable implementation would likely not implement many of TreeGen's features via syntactic rewriting anyway. For instance, it would be much more user-friendly to provide dedicated execution tracing for common issues like pattern-based constructs failing to reach a valid match (see chapter 6 for such a tracing model). Rather, these rewrites should give sufficient information to infer the set of equivalent behaviours allowed of an implementation of the full language.

A note on name substitution

Many of the following sections use the notation $[n \to n']e$ to indicate name substitution from n to n' in expression e. This does not imply naive substitution: it implies substitution only where n is a free variable in e. For instance, the substitution $[n \to n']\texttt{let}\ n = 1\ \texttt{in}\ n$ becomes $\texttt{let}\ n = 1\ \texttt{in}\ n$ rather than $\texttt{let}\ n' = 1; \texttt{in}\ n'$, since the name n is not a free variable in the targeted expression and therefore should not be affected by the substitution.

5.1 Full TreeGen grammar

To start with, almost all of the full TreeGen language can be described using the following context-free grammar, additional lexical details and other exceptions being described in the following subsection.

⟨*expression*⟩ ::= '`???`'
| ⟨*literal*⟩
| ⟨*name*⟩
| ⟨*dollar-name*⟩
| ⟨*expression*⟩ '`/`' ⟨*name*⟩
| '`(`' ⟨*expression*⟩ '`)`'
| '`\\`' ⟨*pattern*⟩ '`=>`' ⟨*expression*⟩
| '`\\`' '`match`' '`{`' (⟨*match-case*⟩ '`;`')* '`}`'
| ⟨*expression*⟩ '`match`' '`{`' (⟨*match-case*⟩ '`;`')* '`}`'
| '`effect`' ⟨*expression*⟩ ('`=`' ⟨*expression*⟩)* '`;`' ⟨*expression*⟩
| '`assert`' ⟨*expression*⟩ '`;`' ⟨*expression*⟩
| '`{`' (⟨*directive*⟩ '`;`')* '`}`'
| ⟨*expression*⟩ '`::`' ⟨*expression*⟩
| '`[`' (⟨*expression*⟩ '`,`' ⟨*expression*⟩)* '`,`'?)? '`]`'
| ⟨*expression*⟩ '`(`' ⟨*expression*⟩ '`)`'
| '`let`' ⟨*pattern*⟩ '`=`' ⟨*expression*⟩ '`in`' ⟨*expression*⟩
| '`if`' ⟨*expression*⟩ '`then`' ⟨*expression*⟩ '`else`' ⟨*expression*⟩
| ⟨*expression*⟩ ⟨*binop*⟩ ⟨*expression*⟩
| '`!`' ⟨*expression*⟩
| ⟨*expression*⟩ '`==`' ⟨*expression*⟩

This grammar is almost a superset of that given in chapter 4, with the exception of the user-inaccessible type constructor `type(...)` being absent, as this grammar describes the user-facing language. In addition to the core language, it includes the concepts introduced in chapter 3 that were missing in chapter 4: dollar names, list syntax, matches, the full set of possible directives, and patterns.

⟨*directive*⟩ ::= ⟨*pattern*⟩ '`=`' ⟨*expression*⟩
| '`effect`' ⟨*expression*⟩ ('`=`' ⟨*expression*⟩)*
| '`assert`' ⟨*expression*⟩
| '`singleton`' ⟨*name*⟩
| '`type`' ⟨*name*⟩ '`=`' ⟨*pattern*⟩ ('`as`' ⟨*expression*⟩)?

⟨*match-case*⟩ ::= ‘`case`’ ⟨*pattern*⟩ ‘`=>`’ ⟨*expression*⟩
  | ‘`case`’ ‘`*`’ = ⟨*expression*⟩

⟨*pattern-name*⟩ := ⟨*name*⟩ | ⟨*dollar-name*⟩ | ‘`_`’

⟨*pattern*⟩ ::= ⟨*pattern-name*⟩ (‘`:`’ ⟨*expression*⟩)? (‘`:=`’ ⟨*pattern*⟩)?
  | ‘`let`’ ⟨*pattern*⟩
  | ⟨*pattern*⟩ ‘`if`’ ⟨*expression*⟩
  | ‘`(`’ ⟨*pattern*⟩ ‘`)`’
  | ⟨*pattern*⟩ ‘`::`’ ⟨*pattern*⟩
  | ‘`{`’ (⟨*pattern-dir*⟩ ‘`;`’)* ‘`}`’
  | ‘`[`’ (⟨*pattern*⟩ (‘`,`’ ⟨*pattern*⟩)*)* ‘`]`’

⟨*pattern-dir-name*⟩ ::= ‘`let`’? ⟨*pattern-name*⟩
  | ⟨*name*⟩ ‘`->`’ ‘`let`’? ⟨*pattern-name*⟩

⟨*pattern-dir*⟩ ::= ‘`if`’ ⟨*expression*⟩
  | ⟨*pattern-dir-name*⟩ (‘`?=`’ ⟨*expression*⟩)? (‘`:`’ ⟨*expression*⟩)? (‘`:=`’ ⟨*pattern*⟩)?

5.1.1 Grammatical and lexical refinements

Some details of TreeGen either complicate the grammar unnecessarily or are more lexical than formally meaningful, so they are discussed separately from the grammar.

The let pattern prefix

The grammar given for patterns is simplified to avoid duplication. The grammar given allows constructions like `let let x = 2 in x`, which makes no sense other than as a confusing edge case. The purpose of allowing let to appear as a prefix to a pattern is specifically for the left-hand side of assignment directives, where one might want to let bind one part of the pattern and not another (e.g., the pattern assignment directive in `{ [let x, y] = [1, 2] }`, being roughly equivalent to `{ let x = 1; y = 2 }`). All other instances of the pattern grammar should be interpreted as if the let cases were absent.

Even when let is allowed, repeating it where it is redundant is forbidden: the directive `[let x, let y] = [1, 2]` makes sense whereas in `[let [let x, y]] = [[1, 2], 3]` the inner let does not afford any useful interpretation. The inner let is either redundant or programmer error, and TreeGen chooses to interpret it as the second option.

Let agreement

In addition to the rules for let pattern prefixes, there is the issue of *let agreement*. Consider listing 5.1 – is `foo` a field of the root object or not? This is an example of a program violating let agreement. Since there is no consistent interpretation for this kind of program all valid programs must follow let agreement: within the same object expression, bindings of the same name cannot mix having and not having a let prefix.

```
{
    foo = 1
    let foo = 1
}
```

Listing 5.1: A program that violates the let agreement rule

Optional punctuation

All semicolons in the grammar are optional, meaning that developers can use them for added readability and omit them when the program is already clearly readable, usually when a newline provides the required visual separation instead. Programs are parseable with all the optional punctuation missing in any case, as long as the parser covers the special case discussed in section 5.1.1. This helps keep programs clean-looking while avoiding most of the complexity of heavily whitespace-sensitive grammars.

For an example of where it would help a human to include a semicolon, consider the single-line object expression `{ x = 1 y = 2 }`, where the two directives `x = 1` and `y = 2` are difficult to visually separate. Thanks to optional semicolons, we can rewrite this expression as `{ x = 1; y = 2 }`, giving the reader a cue that there are two directives there. In general, the stylistic recommendation is to use semicolons when listing more than one separate item on a single line, and to omit semicolons when listing items on multiple lines.

Along a similar theme, list expressions allow a trailing comma after the last element. This is a common feature of list expressions, avoiding the need to add and remove commas when moving list elements about in a multiline list literal.

Lastly, to avoid a pair of redundant braces in every TreeGen program, it is conventional to not write the braces surrounding the root object expression. This makes listing 5.2 a valid TreeGen program defining a root object with fields `a` and `b`, despite the lack of braces normally required by the object expression.

```
a = 1
b = 2
```

Listing 5.2: Demonstration of optional root object braces in full TreeGen

Function call sugar

A trivial but convenient addition to function call syntax is to allow skipping the parentheses when the argument is either an object expression or a list expression. For example, `fn [1,2] = fn ([1,2])` or `fn {a = 1; b = 2} = fn ({a = 1; b = 2})`. Both of these cases are very common in practice, with the omission of function call brackets allowing for a very natural simulation of both positional and keyword-style argument passing despite TreeGen functions only supporting a single argument.

No new line between function and argument

The most obscure lexical detail when parsing TreeGen is a workaround to what would otherwise make TreeGen programs only parseable with unlimited lookahead. If you omit semicolons as has been done in many examples, it is possible to write a somewhat contrived program like listing 5.3.

```
1 foo = \\ x => x + 1
2 _ = foo
3 (bar) = foo
```

Listing 5.3: A contrived snippet demonstrating a lexical issue in full TreeGen

At a parsing level, it takes an arbitrary lookahead to figure out that line 3 is an assignment directive binding `bar` via a pattern that happens to be surrounded by parentheses, not a continuation of line 2 where `(bar)` is the argument part of a call expression.

This local ambiguity can be solved via the same heuristic a human reader might use: it's on a new line and the previous line looks complete, so it's unlikely to be a continuation of a call expression. TreeGen's single whitespace-sensitive lexical rule is that function calls where the gap between the function and the beginning of the argument syntax contains a new line are forbidden, making it possible to discard the less likely option immediately upon reaching a new line. As a consequence, this enforces the coding convention pictured in listing 5.4 that leaves the argument's opening parenthesis, bracket or brace on the same line as the function expression itself.

```
// valid
_ = foo {
    a = 1
    b = 2
}
// not valid
_ = foo
{
    a = 1
    b = 2
}
```

Listing 5.4: The correct and incorrect ways to write multiline function calls

5.2 Eliminating list syntax

Eliminating list syntax first removes several purely cosmetic syntax cases from all subsequent eliminations. List syntax exists both in expressions and patterns, and the eliminations for each variant have a sort of symmetry.

First we can eliminate the `[...]` syntax. Equation (5.2.1) shows the elimination of the expression version, rewriting it into the other `::`-based form. All lists written `[...]` are implicitly finite, so the sequence ends with the empty list singleton `predef/list/nil`.

$$[e_1, \ldots, e_n] \longrightarrow e_1 :: \cdots :: e_n :: \texttt{predef/list/nil} \tag{5.2.1}$$

The pattern variant in eq. (5.2.2) is almost identical, the only change being that instead of generating an end of list we check for it using a wildcard type check to avoid binding any names.

$$[p_1, \ldots, p_n] \longrightarrow p_1 :: \cdots :: p_n :: \texttt{\_: predef/list/nil} \tag{5.2.2}$$

Next, we can individually eliminate all instances of the `::` syntax.

For the expression variant given in eq. (5.2.3), we translate `::` into a call to the list cons node constructor `predef/list/cons`. The translation is actually quite subtle, due to scoping: e_h and e_t might reference arbitrary names including `head` or `tail`. To avoid this, we use a combination of fresh variable names and dollar names.

To construct the argument with fields `head` and `tail`, we first use let directives to bind the head and tail values to fresh names, ensuring that there can be no conflict between the bound names and e_h or e_t. Note that this conflict avoidance step is inevitable, since even if we used dollar names to avoid an accidentally recursive definition, one of the directives would come second and risk a conflict with the dollar name defined by the first.

The second step then is to bind the two fresh names to the actual field names, knowing that since the right hand side expressions are fresh names they can no longer conflict with `head` or `tail`. We still have to use dollar names for the field names, since otherwise `head` and `tail` would be in scope for all directives including those mentioning e_h and e_t, regardless of how many indirections and fresh names we introduce.

Note that $\mathbf{fresh}_1$ and $\mathbf{fresh}_2$ refer to two distinct fresh variable names that are not otherwise mentioned by the program. How they are acquired is not important – many common algorithms and conventions exist for this.

$$e_h :: e_t \longrightarrow \begin{array}{l}\texttt{predef/list/cons}\ \{\texttt{let}\ \mathbf{fresh}_1 = e_h; \\ \quad \texttt{let}\ \mathbf{fresh}_2 = e_t;\ \$\texttt{head} = \mathbf{fresh}_1;\ \$\texttt{tail} = \mathbf{fresh}_2\}\end{array} \tag{5.2.3}$$

The elimination of the pattern `::` given in eq. (5.2.4) is much simpler by comparison, thanks to wildcard patterns. The rewrite gives a pattern that typechecks against `predef/list/cons` then uses a block sub-pattern to apply p_h and p_t to the `head` and `tail` fields respectively. Since every possible name binding is replaced by the wildcard `_`, there is no need to worry about extra indirections or fresh variable names.

$$p_h :: p_t \longrightarrow \begin{array}{l}\_ : \texttt{predef/list/cons} := \\ \quad \{\ \texttt{head} \texttt{->\_} := p_h;\ \texttt{tail} \texttt{->\_} := p_t\}\end{array} \tag{5.2.4}$$

5.3 Eliminating patterns

Patterns themselves do not have a direct rewrite – the effect of writing a pattern is context dependent to an extent. Pattern elimination is described by the function $elim(n_s, p, n_r, e_b)$, which eliminates pattern p where e_b is the "body expression" in which all of p's bindings should take effect. n_r is the name to which the an object satisfying the pattern protocol described below should be bound in e_b. n_s is the name of the scrutinee from that same protocol. There is also a variant $elim_d$ that applies in the context of assignment directives.

Patterns and many derived constructs follow the *pattern protocol*. The pattern protocol is a scheme for signaling success or failure given some input. A pattern or other compatible

object takes a *scrutinee* and outputs both a result and a *check*. The check is a boolean indicating whether the scrutinee was accepted. If the check is true, then the result refers to a value that can be used, and if false, the result is unspecified and likely unknown. Both functions and patterns follow this protocol once eliminated.

5.3.1 Stage 1: scoping

All patterns have an initial scoping setup stage which does two things: gather and initially bind the variables bound by the pattern to `???`, and build a renaming scheme Γ that allows some expressions embedded in the pattern to ignore the newly bound names. This splits the definition of *elim* into *elim* and $elim_2$, with one calling the other as shown below.

The definition uses two functions *closure* and *bindings*. $closure(p)$ yields a set of names, an extension of the function with the same name defined in chapter 4. *bindings* yields a set of *bindings* which are names optionally prefixed with `let`. Note that the let prefixing cannot happen outside of the later definition of $elim_d$, so in this case, *bindings* can be thought of as returning just a set of names. Precise definitions for both these functions are given in a following section.

$$
\begin{aligned}
&elim(n_s, p, n_r, e_b) = \\
&\quad \text{let } \Gamma = \{(n, \mathbf{fresh}_n) \mid n \in closure(p)\} \\
&\quad \text{let } e'_b = elim_2(n_s, \Gamma, p, n_r, e_b) \\
&\quad \text{let } e''_b = b_1 \ldots b_n \in bindings(p), \texttt{let } b_1 = \ \texttt{???}\ \texttt{in} \ldots \texttt{let } b_n = \ \texttt{???}\ \texttt{in } e'_b \\
&\quad \text{in } (n_1, n_{f_1}) \ldots (n_n, n_{f_n}) \in \Gamma, \texttt{let } n_{f_1} = n_1 \texttt{ in} \ldots \texttt{let } n_{f_n} = n_n \texttt{ in } e''_b
\end{aligned}
$$

In *elim*, we first rebind any names referenced by pattern p's closure to fresh names in Γ in order to avoid scoping conflicts where names bound by the pattern shadow names referenced by parts of the pattern. As a real, if contrived, example, consider listing 5.5. On line 5 one would typically mean "bind field `$t` to the result of expression `$t {}`, where the value has to be of type `$t`", where `$t` describes the elsewhere-defined type and not the name we are in the process of binding.

Once we've taken care of setting up Γ and its associated bindings, we generate bindings to `???` for all of the names bound by the pattern, so that the names are available throughout the rewritten pattern. Only then are we prepared to rewrite specific parts of the pattern via $elim_2$.

```
{
    type t = _
    $t = t
    _ = {
        $t: $t = $t {}
    }
}
```

Listing 5.5: A demonstration of the rationale for how type checks are scoped in patterns

The variation for directives $elim_d$ operates under the same principles as $elim$, but instead of emitting let expressions, it emits assignment directives to be placed in an enclosing object expression. Since any bound names will naturally be in scope for the appropriate sibling directives (this may vary if dollar names are involved), there are no equivalents to n_r and e_b. Note that since the bindings $b_1 \dots b_n \in bindings(p)$ may include the prefix `let`, this means that the assignment directives will vary from simple name assignments to let assignments – how let assignments can be eliminated is conceptually orthogonal and will be explained later.

$$
\begin{aligned}
&elim_d(n_s, p) = \\
&\quad \text{let } \Gamma = \{(n, \mathbf{fresh}_n) \mid n \in closure(p)\} \\
&\quad \text{let } e'_b = [\![\texttt{let } \mathbf{fresh}_b = elim_2(n_s, \Gamma, p, \mathbf{fresh}_r, \mathbf{fresh}_r)]\!] \\
&\quad \text{let } e''_b = b_1 \dots b_n \in bindings(p),\ [\![b_1 = \ ???; \dots; b_n = \ ???;\ e'_b]\!] \\
&\quad \text{in } (n_1, n_{f_1}) \dots (n_n, n_{f_n}) \in \Gamma, [\![\texttt{let } n_{f_1} = n_1; \dots \texttt{let } n_{f_n} = n_n;\ e''_b]\!]
\end{aligned}
$$

Notice also how $elim_d$ adapts the results of $elim_2$ in order to avoid having to define a new version $elim_{2_d}$ for directives: $elim_2$ is given $\mathbf{fresh}_r$ as both a name to which to bind the innermost pattern protocol object and a body, meaning that the result of the whole expression coming from $elim_2$ will be the result of expanding pattern p. That object is then bound to n_r via a let directive so that the directives in d can access it.

5.3.2 Pattern scoping rules

Patterns extend the existing scoping rules for the core language, building on *closure* as defined in chapter 4. Unlike for expressions, the closure mechanism for patterns requires some extra functions: $closure_{pre}$, $closure_{post}$ and $bindings$. These functions respectively define the set of names that cannot be shadowed, those that can, and the names defined by the given pattern.

The top-level *closure* combines all of these, keeping all the unshadowable names from $closure_{pre}$ while removing any bound names from $closure_{post}$.

$closure(p)$ if **is_pattern**$(p) =$
  let $b = bindings(p)$
  in $closure_{pre}(p) \cup (closure_{post}(p) \setminus \{n \mid n \in b \vee \texttt{let}\ n \in b\})$

Pre-closure

Almost all the names in a pattern are unshadowable, except for those in e_c from the conditional refinements. In order to express a condition over the bound names, those names have to be in scope.

Note that the name pattern is deconstructed as if its parts were fully independent (just a name/wildcard, something then a typecheck, something then a sub-pattern) rather than just optional in order to avoid combinatorically exploring all possible instantiations of the optional parts.

$closure_{pre}([\![\_]\!]) = \emptyset$
$closure_{pre}([\![n]\!])$ if **is_name**$(n) = \emptyset$
$closure_{pre}([\![p : e_t]\!]) = closure_{pre}(p) \cup closure(e_t)$
$closure_{pre}([\![p := p']\!]) = closure_{pre}(p) \cup closure_{pre}(p')$
$closure_{pre}([\![\texttt{let}\ p]\!]) = closure_{pre}(p)$
$closure_{pre}([\![p\ \texttt{if}\ e_c]\!]) = closure_{pre}(p)$
$closure_{pre}([\![\{d \ldots\}]\!]) = \bigcup_{d \ldots} dir\_closure_{pre}(d)$

Due to the heterogenous nature of block patterns, a subordinate function *dir_closure* is defined for each pattern directive. The **is_binding**(b) predicate is a check that b is one of the possible instances of the grammar element pattern-dir-name.

$dir\_closure_{pre}([\![b]\!])$ if $\mathbf{is\_binding}(b) = \emptyset$
$dir\_closure_{pre}([\![p^{?=}e_d]\!]) = closure_{pre}(p) \cup closure(e_d)$
$dir\_closure_{pre}([\![p : e_t]\!]) = closure_{pre}(p) \cup closure(e_t)$
$dir\_closure_{pre}([\![p := p']\!]) = closure_{pre}(p) closure_{pre}(p')$
$dir\_closure_{pre}([\![\texttt{if } e_c]\!]) = \emptyset$

Post-closure

Conversely, $closure_{post}$ captures only e_c's closure. It otherwise follows the same structure as the *pre* version.

$closure_{post}([\![\_]\!]) = \emptyset$
$closure_{post}([\![n]\!])$ if $\mathbf{is\_name}(n) = \emptyset$
$closure_{post}([\![p : e_t]\!]) = closure_{post}(p)$
$closure_{post}([\![p := p']\!]) = closure_{post}(p) \cup closure_{post}(p')$
$closure_{post}([\![\texttt{let } p]\!]) = closure_{post}(p)$
$closure_{post}([\![p \texttt{ if } e_c]\!]) = closure(e_c) \cup closure_{post}(p)$
$closure_{post}([\![\{d \dots\}]\!]) = \bigcup_{d \dots} dir\_closure_{post}(d)$

$dir\_closure_{post}$ likewise shares the same structure, while only collecting $closure(e_c)$ from conditional refinements.

$dir\_closure_{post}([\![b]\!])$ if $\mathbf{is\_binding}(b) = \emptyset$
$dir\_closure_{post}([\![p^{?=}e_d]\!]) = closure_{post}(p)$
$dir\_closure_{post}([\![p : e_t]\!]) = closure_{post}(p)$
$dir\_closure_{post}([\![p := p']\!]) = closurepost(p) \cup closure_{post}(p')$
$dir\_closure_{post}([\![\texttt{if } e_c]\!]) = closure(e_c)$

Bindings

Lastly, *bindings* captures which variables are actually bound by a given pattern. For the most part, *bindings* binds any names mentioned in non-expression position, except if it reaches a let prefix. In that case, it prefixes any bindings belonging to the inner pattern with `let`. This should only apply one let prefix to any name, since nested let prefixes are forbidden at the syntactic level.

$$
\begin{aligned}
&\mathit{bindings}([\![\_]\!]) = \emptyset \\
&\mathit{bindings}([\![n]\!]) \text{ if } \mathbf{is\_name}(n) = \{n\} \\
&\mathit{bindings}([\![p : e_t]\!]) = \mathit{bindings}(p) \cup \{n\} \\
&\mathit{bindings}([\![p := p']\!] = \mathit{bindings}(p) \cup \mathit{bindings}(p') \\
&\mathit{bindings}([\![\texttt{let } p]\!]) = \{\texttt{let } n \mid n \in \mathit{bindings}(p)\} \\
&\mathit{bindings}([\![p \texttt{ if } e_c]\!]) = \mathit{bindings}(p) \\
&\mathit{bindings}([\![\{d \ldots\}]\!]) = \bigcup_{d \ldots} \mathit{dir\_bindings}(d)
\end{aligned}
$$

As with the two closure functions the bindings for object pattern directives are given separately via *dir_bindings*. This time, the binding b matters, though, so the actual previously singular binding case is split out into the various possible cases involving wildcards, let prefixes and rebindings.

$$
\begin{aligned}
&\mathit{dir\_bindings}([\![\_]\!]) = \emptyset \\
&\mathit{dir\_bindings}([\![n]\!]) \text{ if } \mathbf{is\_name}(n) = \{n\} \\
&\mathit{dir\_bindings}([\![\texttt{let } n]\!]) = \{\texttt{let } n\} \\
&\mathit{dir\_bindings}([\![n\texttt{->}n']\!]) = \{n'\} \\
&\mathit{dir\_bindings}([\![n\texttt{->let } n']\!]) = \{\texttt{let } n'\} \\
&\mathit{dir\_bindings}([\![p \texttt{ ?= } e_d]\!]) = \mathit{dir\_bindings}(p) \\
&\mathit{dir\_bindings}([\![p : e_t]\!]) = \mathit{dir\_bindings}(p) \\
&\mathit{dir\_bindings}([\![p := p']\!]) = \mathit{dir\_bindings}(p) \cup \mathit{bindings}(p') \\
&\mathit{dir\_bindings}([\![\texttt{if } e_c]\!]) = \emptyset
\end{aligned}
$$

5.3.3 Stage 2: individual pattern semantics

The second and last stage of eliminating patterns is to give the individual pattern semantics. This involves considering each individual case, defining how $elim_2$ should rewrite a given pattern p in terms of n_s, Γ, n_r and e_b.

Note that a feature common to all rewrites is that all pattern sub-expressions outside of conditionals must be rewritten via Γ to avoid scoping issues.

Name patterns

The first pattern to cover is the name binding pattern, which has several optional components. Since a name binding can be read effectively from left to right, it makes sense to try to split up the semantics into segments.

The first segment is either a name binding or a wildcard, so it can have two individual cases. They differ by whether or not the scrutinee n_s is merged with the bound name n – if there is no n there is no merge. Aside from that the result is unconditional success: the value in n_r becomes bound to n_s, and the check is bound to the literal true.

$$
\begin{aligned}
&elim_2(n_s, \Gamma, [\![\_]\!], n_r, e_b) = \\
&\quad [\![\texttt{let } n_r = \{\$\texttt{value} = n_s;\ \$\texttt{check} = \texttt{true}\} \texttt{ in } e_b]\!] \\
&elim_2(n_s, \Gamma, [\![n]\!], n_r, e_b) \text{ if } \textbf{is\_name}(n) = \\
&\quad [\![\texttt{effect } n_s = n;\ \texttt{let } n_r = \{\$\texttt{value} = n_s;\ \$\texttt{check} = \texttt{true}\} \texttt{ in } e_b]\!]
\end{aligned}
$$

Following the name the first optional part of a name pattern is a type check. Given some prefix p that would be the name pattern discussed above, a new body e'_b is generated that extends e_b with the code to perform the typecheck. In e'_b we expect the previous pattern's checks to have already been performed with the results stored in $\textbf{fresh}_r$. As is common for all patterns with sub-patterns, we first short-circuit out if the sub-pattern failed, which accounts for the first line of e'_b.

Next comes the logic: using `predef/object/has_field`, we check to see if the value of e_t, bound to $\textbf{fresh}_t$ to avoid multiple syntactic copies of e_t, has a field called `unapply`. If so then it's a type of type-like object with specialised typechecking behaviour, so we bind n_r to the result of calling `unapply`. If not, then the only other valid choice is that it is

some kind of partial function, so we generate a call to it. Both of these cases rely on the expanded representation of partial functions which must return an object following the pattern protocol.

$$
\begin{array}{l}
\mathit{elim}_2(n_s, \Gamma, [\![p : e_t]\!], n_r, e_b) = \\
\quad \text{let } e'_b = [\![\\
\qquad \texttt{if } !(\texttt{fresh}_r/\texttt{check}) \texttt{ then } \mathbf{fresh}_r \texttt{ else} \\
\qquad \texttt{let } \mathbf{fresh}_t = [i \to \Gamma(i)]e_t \texttt{ in} \\
\qquad \texttt{let } n_r = \texttt{if predef/object/has\_field\{} \\
\qquad\quad \texttt{obj} = \mathbf{fresh}_t \\
\qquad\quad \texttt{name} = \text{“}\texttt{unapply}\text{”} \\
\qquad\quad \} \\
\qquad\quad \texttt{then } \mathbf{fresh}_t/\texttt{unapply}(\mathbf{fresh}_r/\texttt{value}) \\
\qquad\quad \texttt{else } \mathbf{fresh}_t(\mathbf{fresh}_r/\texttt{value}) \\
\qquad \texttt{in } e_b \\
\quad]\!] \\
\quad \text{in } \mathit{elim}_2(n_s, \Gamma, p, \mathbf{fresh}_r, e'_b)
\end{array}
$$

The second and last optional part for a name pattern is the sub-pattern. It is executed as “try p first and only attempt p' if p succeeds”, and has a similar recursive reference to elim_2 as the type check. This yields a relatively simple rule, generating an e'_b that checks to see if p has failed and only if p has succeeded attempts the code generated by rewriting p'. Notice how the scrutinee of p' is actually the result of p – usually this makes no difference but if p contained a typecheck then that typecheck's result could be used to control which value p' inspects.

$$
\begin{array}{l}
\mathit{elim}_2(n_s, \Gamma, [\![p := p']\!], n_r, e_b) = \\
\quad \text{let } e'_b = [\![\\
\qquad \texttt{if } !(\mathbf{fresh}_r/\texttt{check}) \texttt{ then } \mathbf{fresh}_r \texttt{ else} \\
\qquad \mathit{elim}_2(\mathbf{fresh}_r/\texttt{value}, \Gamma, p', n_r, e_b) \\
\quad]\!] \\
\quad \text{in } \mathit{elim}_2(n_s, \Gamma, p, \mathbf{fresh}_r, e'_b)
\end{array}
$$

Let prefixes

The let prefix has no effect aside from altering scoping. Its expansion is the same as if the let prefix were absent.

$$elim_2(n_s, \Gamma, [\![\texttt{let}\ p]\!], n_r, e_b) = elim_2(n_s, \Gamma, p, n_r, e_b)$$

Conditional refinements

Conditional refinement has the usual situation for having a sub-pattern p – the semantics generate a new e'_b that first checks to make sure p matched, and if so binds n_r to an object with the same value but a check defined by e_c. Note that e_c is not rewritten by Γ because all of the bound pattern variables should be in scope for the conditional. Notice also that the directive with e_c is given first with the dollar name `$check` to avoid the name `check` ending up in e_c's scope.

$$\begin{aligned}
&elim_2(n_s, \Gamma, [\![p\ \texttt{if}\ e_c]\!], n_r, e_b) = \\
&\quad \text{let } e'_b = [\![\\
&\qquad \texttt{if}\ !(\mathbf{fresh}_r/\texttt{check})\ \texttt{then}\ \mathbf{fresh}_r\ \texttt{else} \\
&\qquad \texttt{let}\ n_r = \{\$\texttt{check} = e_c;\ \$\texttt{value} = \mathbf{fresh}_r/\texttt{value}\} \\
&\qquad \texttt{in}\ e_b \\
&\quad]\!] \\
&\quad \text{in } elim_2(n_s, \Gamma, p, \mathbf{fresh}_r, e'_b)
\end{aligned}$$

Block patterns

Block patterns have some specialised semantics. Instead of operating entirely sequentially like the other pattern elements which require the previous condition to be checked before moving on, block patterns split out and check all of the pattern directives concurrently, bringing all of the resulting conditions $n_{c_1} \ldots n_{c_n}$ together in a conjunction at the end.

Unfortunately, there is no practically useful value for a block pattern to yield, so ??? is given unconditionally. This is because there is no situation where the result of a block pattern can actually be used: all the expressions that use patterns only use the bound

names, not the result, and the only suffix that a block pattern can have is a conditional refinement that also only uses the bound names.

$$
\begin{array}{l}
elim_2(n_s, \Gamma, [\![\{d \ldots\}]\!], n_r, e_b) = \\
\quad \text{let } (d', C) = dirs\_elim(n_s, \Gamma, d) \ldots \\
\quad \text{in } [\![\\
\quad\quad \texttt{let } n_r = \{d' \ldots;\ \$\texttt{value} = ??? \\
\quad\quad\quad \$\texttt{check} = n_{c_1} \ldots n_{c_n} \in C, [\![n_{c_1} \mathbin{\&\&} \ldots \mathbin{\&\&} n_{c_n}]\!]\} \\
\quad\quad \texttt{in } e_b \\
\quad]\!]
\end{array}
$$

Each pattern directive is eliminated by the function $dirs\_elim(n_s, \Gamma, d) = (d', C, B)$ which maps the scrutinee n_s, the renamings Γ, and the pattern directives d to a collection of directives d' that includes assignments to conditional names $n_c \in C$. Together these assignments represent all of the conditionals that contribute to the pattern's check.

The function is defined piecewise to document each case of pattern directive, starting with the empty case of no pattern directives, in which case there are no conditionals and no bindings. Note that we consider $n_{c_1} \mathbin{\&\&} \ldots \mathbin{\&\&} n_{c_n}$ to mean `true` if C is empty, as is conventional for reduction over boolean conjunction.

$$
\begin{array}{l}
dirs\_elim(n_s, \Gamma, [\![]\!]) = \\
\quad ([\![]\!], \emptyset)
\end{array}
$$

The second case is the conditional refinement directive, which mirrors the non-directive version. It assigns the result of e_c to $\mathbf{fresh}_c$, which it adds to C to record that this was a pattern condition. e_c is not rewritten via Γ, since again, the pattern variables should be in scope for conditional refinements.

$$
\begin{array}{l}
dirs\_elim(n_s, \Gamma, [\![d;\ \texttt{if } e_c]\!]) = \\
\quad \text{let } (d', C) = dirs\_elim(n_s, \Gamma, d) \\
\quad \text{in } ([\![d';\ \texttt{let } \mathbf{fresh}_c = e_c]\!], C \cup \{\mathbf{fresh}_c\})
\end{array}
$$

The third case is the most complex, as it mirrors name patterns and contains an extra case for default values. Like the other case, it binds the condition to $\mathbf{fresh}_c$, but unlike

the other case, it also has to handle the various cases possible for d_1, which requires a separate helper $dir\_elim(n_s, \Gamma, d_1, n_r)$ which carries over n_s and Γ, mapping d_1 to a series of directives d_1' that contain an assignment to the name n_r. n_r has the same semantics as in $elim_2$, in that it should refer to a pattern protocol object. As with $dirs\_elim$, the function is defined piecewise below to illustrate each possible case of the pattern directive.

$$
\begin{aligned}
&dirs\_elim(n_s, \Gamma, [\![d;\ d_1]\!]) \text{ if } \mathbf{is\_bind\_directive}(d_1) = \\
&\quad \text{let } (d', C) = dirs\_elim(n_s, \Gamma, d) \\
&\quad \text{let } d_1' = dir\_elim(n_s, \Gamma, d_1, \mathbf{fresh}_r) \\
&\quad \text{in } ([\![d';\ d_1']\!], C \cup \{\mathbf{fresh}_r/\texttt{check}\})
\end{aligned}
$$

Unlike most parts of the pattern directive, default values require special handling. Normally parts of a name pattern can be chained together with a notion of "the previous part's value" and n_r, but if an optional field is present, the logic for whether the field is present needs to mix with the logic for the binding itself. The definition below shows how this mix can be achieved when a default value is in play, abstracting over variations on b using some convenience notations.

Instead of recursing on b, we check to see if the field name b refers to is present for n_s, written $\mathbf{field\_name}(b)$. If so, we access the field's value and bind it to $\mathbf{fresh}_r$. If not, we bind the result of the default expression e_d to it, substituting via Γ to scope e_d outside of the other pattern values. Then, we construct a pattern protocol object whose check is unconditionally true and whose value is $\mathbf{fresh}_r$. The last piece of logic abstracts over the name that b should bind the extracted value to, if there is one. If there is, we use an effect directive to merge $\mathbf{fresh}_r$ and the bound name n. Otherwise, there is nothing else to do.

$$
\begin{aligned}
&dir\_elim(n_s, \Gamma, [\![b \mathbin{?=} e_d]\!], n_r) \text{ if } \mathbf{is\_binding}(b) = \\
&\quad [\![\ \texttt{let } \mathbf{fresh}_r = \texttt{if predef/object/has\_field}\{ \\
&\qquad\quad \texttt{src} = n_s;\ \texttt{name} = \text{“}\mathbf{field\_name}(b)\text{”}\} \\
&\qquad \texttt{then } n_s/\mathbf{field\_name}(b) \\
&\qquad \texttt{else } [n \to \Gamma(n)]e_d \\
&\qquad \texttt{let } n_r = \{\texttt{check} = \texttt{true};\ \texttt{value} = \mathbf{fresh}_r\}]\!] \\
&\quad \text{if } \mathbf{bound\_name}(b) = n \\
&\qquad [\![\texttt{effect } \mathbf{fresh}_r = n]\!] \\
&\quad \text{else } [\![]\!]
\end{aligned}
$$

Correspondingly, there is also a variant for processing bindings that are not suffixed with a default expression. It is a strict subset of the previous definition, unconditionally projecting out the field name from b. As before, there is also a condition for whether b binds a name or not.

$$
\begin{aligned}
&\mathit{dir\_elim}(n_s, \Gamma, [\![b]\!], n_r) \text{ if } \mathbf{is\_binding}(b) = \\
&\quad [\![\ \texttt{let}\ n_r = \{\texttt{value} = n_s/\mathbf{field\_name}(b);\ \texttt{check} = \texttt{true}\}]\!] \\
&\quad \text{if } \mathbf{bound\_name}(b) = n \\
&\qquad [\![\texttt{effect}\ n_r/\texttt{value} = n]\!] \\
&\quad \text{else } [\![]\!]
\end{aligned}
$$

The next part corresponds to the typecheck element of a name pattern. It duplicates nearly all of the corresponding logic, the main difference being that it assigns to n_r via a let directive instead of the nested let expressions from elim_2.

$$
\begin{aligned}
&\mathit{dir\_elim}(n_s, \Gamma, [\![d : e_t]\!], n_r) = \\
&\quad \text{let } d' = \mathit{dir\_elim}(n_s, \Gamma, d, \mathbf{fresh}_r) \\
&\quad \text{in } [\![\texttt{let}\ n_r = \texttt{if}\ !(n_r/\texttt{check})\ \texttt{then}\ \mathbf{fresh}_r\ \texttt{else} \\
&\qquad\qquad\qquad \texttt{let}\ \mathbf{fresh}_t = e_t\ \texttt{in} \\
&\qquad\qquad\qquad \texttt{if}\ \texttt{predef/object/has\_field}\{\texttt{src} = \mathbf{fresh}_t \\
&\qquad\qquad\qquad\quad \texttt{name} = \text{“}\texttt{unapply}\text{”}\} \\
&\qquad\qquad\qquad \texttt{then}\ \mathbf{fresh}_t/\texttt{unapply}(\mathbf{fresh}_r/\texttt{value}) \\
&\qquad\qquad\qquad \texttt{else}\ \mathbf{fresh}_t(\mathbf{fresh}_t/\texttt{value})]\!]
\end{aligned}
$$

The last optional part of a name pattern directive is the sub-pattern, which again mirrors the corresponding sub-pattern from name directives. First we check that $\mathbf{fresh}_r$ holds a successful pattern protocol object, then we apply pattern p using elim_2 using a similar structure to that of a simplified elim_d.

$$
\begin{aligned}
&\mathit{dir\_elim}(n_s, \Gamma, [\![d := p]\!], n_r) = \\
&\quad \text{let } d' = \mathit{dir\_elim}(n_s, \Gamma, d, \mathbf{fresh}_r) \\
&\quad \text{in } [\![\texttt{let}\ n_r = \texttt{if}\ !(n_r/\texttt{check})\ \texttt{then}\ \mathbf{fresh}_r\ \texttt{else} \\
&\qquad\qquad\qquad \mathit{elim}_2(\mathbf{fresh}_r/\texttt{value}, \Gamma, p, \mathbf{fresh}_{r_2}, \mathbf{fresh}_{r_2})]\!]
\end{aligned}
$$

5.4 Eliminating match expressions

Given the rules for eliminating patterns, match expressions can be rewritten via a mixture of *elim* and if expressions. For simplicity, the rewrites here only make match expressions smaller, expressing the semantics of each step of a match expression independently. Applying the rewrites repeatedly to the remaining subexpression will fully eliminate the match expressions.

$$e_s \texttt{ match } \{\texttt{case } p \texttt{ => } e_b;\ c \ldots\} \longrightarrow \begin{array}{l} \texttt{let } \mathbf{fresh}_s = e_s \texttt{ in} \\ \quad \mathit{elim}(\mathbf{fresh}_s, p, \mathbf{fresh}_r, [\![\\ \quad\quad \texttt{if } \mathbf{fresh}_r\texttt{/check} \\ \quad\quad \texttt{then } e_b \\ \quad\quad \texttt{else } \mathbf{fresh}_s \texttt{ match } \{c \ldots\} \\ \quad]\!]) \end{array}$$

This first rewrite eliminates a regular match case with a pattern p. First the scrutinee e_s is bound to $\mathbf{fresh}_s$ to avoid syntactic duplication, then the pattern is expanded with the pattern protocol object bound to $\mathbf{fresh}_r$. Match expressions don't need the value field – they just need the check. If the check is true, then the branch is taken and the overall result is e_b, the branch's body. If the check is false, then the remaining cases should be tried in order. This is written as making a new match on $\mathbf{fresh}_s$ with the remaining cases c.

$$e_s \texttt{ match } \{\texttt{case } * = e_c;\ c \ldots\} \longrightarrow \begin{array}{l} \texttt{let } \mathbf{fresh}_s = e_s \texttt{ in} \\ \texttt{let } \mathbf{fresh}_c = e_c(\mathbf{fresh}_s) \texttt{ in} \\ \texttt{if } \mathbf{fresh}_c\texttt{/check} \\ \texttt{then } \mathbf{fresh}_c\texttt{/value} \\ \texttt{else } \mathbf{fresh}_s \texttt{ match } \{c \ldots\} \end{array}$$

The other kind of case is the delegate case, which operates similarly to a pattern typecheck. The scrutinee (again bound to $\mathbf{fresh}_s$) is passed to the function or function-like e_c, and the resulting pattern protocol object is bound to $\mathbf{fresh}_c$. If the check is true, then the function is considered to have "matched", and the match expression's result is the value produced. If the check is false, then as before, the other cases c are tried.

$$e_s \texttt{ match } \{\} \longrightarrow \texttt{assert false; ???}$$

The last case is the empty match: with no cases, or if we have already rewritten all the other cases to be outside the match expression, then there are no more branches to try. This is an error, expressed here as a false assertion that yields an unknown value which can never meaningfully be accessed due to the assertion being hard-coded to false.

5.5 Eliminating non-core function features

This section covers all details of functions that do not directly correspond to core TreeGen, mostly relating to patterns and matches.

5.5.1 Function calls

There are two key features of full TreeGen's function calls that need to be eliminated by rewriting callsites: being able to call an object with an `apply` field, and the pattern protocol. These rules are unique in that they are recursive, so we describe them in terms of the recursive function definition given in listing 5.6. We assume this function is somehow accessible (it could be located under `predef`), but realistically, this kind of logic would be built directly into an implementation's function call semantics.

Note that we use a pattern in `fn_call`'s definition for ease of readability, not any particular semantic gain. The pattern could be avoided by replacing all instances of `$fn` and `$arg` by projections from some unnamed single argument.

```
fn_call \\ { $fn; $arg } =>
    let result =
        if predef/object/has_field { src = $fn; name = "apply" }
        then fn_call { fn = $fn/apply; arg = $arg }
        else $fn ($arg)
    in assert result/check; result/val
```

Listing 5.6: The recursively defined semantics for function calls

The first concern is to check for an `apply` field, in which case we must recursively apply the function call logic to that field. Once we have found an `$fn` that does not have an `apply` field, we use the core TreeGen function call operation to call that with `$arg`.

The second concern is to hide the pattern protocol, which we do by asserting that the eventual function call succeeded, that is, that `result/check` is true. Assuming that assertion holds, we make the overall result of `fn_call` be `result/val`, the function call's semantic return value.

$$
e_f(e_a) \longrightarrow \begin{array}{l} \texttt{let}\ \mathbf{fresh}_f = e_f\ \texttt{in} \\ \texttt{let}\ \mathbf{fresh}_a = e_a\ \texttt{in} \\ \texttt{fn\_call}\ \{\texttt{fn} = \mathbf{fresh}_f;\ \texttt{arg} = \mathbf{fresh}_a\} \end{array}
$$

Given that all of this is built into a separate function, the actual rewrite needed to function expressions is quite straightforward: call the function call implementation.

5.5.2 Functions with patterns

$$
\backslash\backslash\ p \texttt{ => } e_b \longrightarrow \begin{array}{l} \backslash\backslash\ \mathbf{fresh}_a \texttt{ => } \mathit{elim}(\mathbf{fresh}_a, p, \mathbf{fresh}_r, [\![\\ \quad \texttt{if}\ \mathbf{fresh}_r\texttt{/check} \\ \quad \texttt{then}\ \{\$\texttt{value} = e_b;\ \$\texttt{check} = \texttt{true}\} \\ \quad \texttt{else}\ \{\$\texttt{value} = ???;\ \$\texttt{check} = \texttt{false}\} \\ [\![) \end{array}
$$

The rules for functions with just one pattern p follow almost directly from eliminating that pattern. The single argument from core TreeGen functions is bound to $\mathbf{fresh}_a$, and the pattern p is expanded with the protocol object bound to $\mathbf{fresh}_r$. If the check is true then e_b is executed with all the pattern-bound names in scope, wrapped in a protocol object that we know must be successful because the pattern is known to have matched. If the check is false, then a false protocol object is returned instead and e_b is not executed.

5.5.3 Match functions

Elimination for a match function is very similar to that for a match expression, except that if no more cases are available, then a false pattern protocol object is returned rather than there being a runtime error.

$$
\backslash\backslash\ \texttt{match}\ \{c \ldots\} \longrightarrow \begin{array}{l} \backslash\backslash\ \mathbf{fresh}_a \texttt{ => } \mathbf{fresh}_a\ \texttt{match}\ \{ \\ \quad c \ldots;\ \texttt{case \_ => }\{\texttt{check} = \texttt{false};\ \texttt{value} = ???\} \end{array}
$$

As an attempt to avoid messily duplicating most of the information from match expression elimination, the above abuse of notation describes roughly the intended rewrite on top of the semantics of the match expression.

Note that this rewrite only makes sense if you consider each case to yield a full protocol object, and not just the result like for match expressions. Rewrites that illustrate this by altering each case c are given below.

$$\text{case } p \texttt{ => } e_b \longrightarrow \text{case } p \texttt{ => } \{\$\texttt{value} = e_b;\ \$\texttt{check} = \texttt{true}\}$$

$$\text{case } * = e_f \longrightarrow \begin{array}{l}\text{case } * = \\ \quad \backslash\backslash\ \mathbf{fresh}_a \texttt{ => } \{\$\texttt{value} = e_f(\mathbf{fresh}_a);\ \$\texttt{check} = \texttt{true}\}\end{array}$$

Notice that the rewrite for delegate cases is highly suboptimal, introducing an entirely new function just to alter the case's return value. Any reasonable implementation would effectively inline these rewrites into a variant of the match expression semantics.

5.6 Eliminating non-core directives

The full TreeGen language has many more directives than core TreeGen, with each one being expressible as a syntax sugar of some sort.

5.6.1 Pattern assignments

Pattern assignments use $elim_d$ in order to be translated into a series of pattern-free directives, with the bindings split out into simple assignments using only names (possibly dollar names) and let directives.

$$p = e \longrightarrow \texttt{let } \mathbf{fresh}_s = e;\ elim_d(\mathbf{fresh}_s, p)$$

5.6.2 Type declarations

TreeGen's type system can be entirely rewritten into a combination of core operations and the core-only `type(...)` constructor. Each directive defines a type tag alongside some related functions.

Singleton directives

The singleton directive is the structurally simplest directive, with its rewrite being very close to the informal one given in chapter 3.

$$\texttt{singleton}\ n \longrightarrow \begin{array}{l} n = \{\%\texttt{type} = \texttt{type}(\text{“}n\text{”}) \\ \quad \texttt{unapply} = \backslash\backslash\ \mathbf{fresh}_a \Rightarrow \{ \\ \quad\quad \texttt{check} = \mathbf{fresh}_a == n \\ \quad\quad \texttt{value} = n\}\} \end{array}$$

Notice that this rewrite takes advantage of the pattern protocol's stipulation that the value is unrestricted if check is false – the value is set to always be n.

Type directives

Type directives define a type tag, an `apply` function and an `unapply` function. These generally match the informal descriptions in chapter 3.

$$\texttt{type}\ n = p \longrightarrow \texttt{type}\ n = \mathbf{fresh}_a := p\ \texttt{as}\ \mathbf{fresh}_a$$

There are two versions of the type directive, one with an extra `as` clause and one without. The `as` clause is used to customise the result of the `apply` function – it operates in the scope of pattern p and the result of the expression will be used to build the result of the `apply` function rather than the `apply` function's actual argument. The one without can be considered equivalent to one with an `as` clause that implements identity semantics – prefix the pattern p with a fresh binding $\mathbf{fresh}_a$ for the entire scrutinee, then make the `as` clause a reference to $\mathbf{fresh}_a$.

$$\texttt{type}\ n = p\ \texttt{as}\ e_b \longrightarrow \begin{array}{l} n = \{\texttt{let}\ \mathbf{fresh}_t = \texttt{type}(\text{“}n\text{”}) \\ \quad \texttt{apply} = \backslash\backslash\ \mathbf{fresh}_a \Rightarrow \\ \quad\quad \mathit{emit}(\mathbf{fresh}_a, p, \mathbf{fresh}_r, [\![\\ \quad\quad\quad \{\$\texttt{value} = \texttt{predef/object/updated}\{\$\texttt{src} = e_b \\ \quad\quad\quad\quad \$\texttt{updated} = [[\text{“}\%\texttt{type}\text{”}, \mathbf{fresh}_t]]\} \\ \quad\quad\quad\quad \$\texttt{check} = \texttt{assert}\ \mathbf{fresh}_r/\texttt{check};\ \texttt{true}\}]\!]) \\ \quad \texttt{unapply} = \backslash\backslash\ \mathbf{fresh}_a \Rightarrow \{ \\ \quad\quad \texttt{check} = \texttt{if}\ \texttt{predef/object/has\_field}\{\texttt{src} = \mathbf{fresh}_a; \\ \quad\quad\quad\quad \texttt{name} = \text{“}\%\texttt{type}\text{”}\} \\ \quad\quad\quad \texttt{then}\ \mathbf{fresh}_a/\%\texttt{type} == \mathbf{fresh}_t \\ \quad\quad\quad \texttt{else}\ \texttt{false} \\ \quad\quad \texttt{value} = \mathbf{fresh}_a\}\} \end{array}$$

The `apply` function takes an argument $\mathbf{fresh}_a$ and tests it against pattern p. Unlike regular functions, instead of forwarding any pattern failures to the caller the `apply` function asserts that the check must pass and adds the `%type` field to the object that comes from evaluating the expression e_b in the pattern's scope.

The reason `apply` functions assert rather than the usual check is as a heuristic method to break dependency cycles. In real-world TreeGen code, it's possible to end up with a circular cross-reference which some patterns try to typecheck. The typechecking then creates a boolean circular dependency between the `apply` function's pattern and the result of the same `apply` function, resulting in a deadlocked program. The cycle can be broken by simply assuming that all type applications should succeed – there seems to be no practical use for matching on a type `apply`.

The `unapply` function on the other hand is conceptually simpler, checking that the argument $\mathbf{fresh}_a$ is an object with a field called `%type` that is equal to the generated type tag $\mathbf{fresh}_t$. If so, the check for the resulting pattern protocol object is true and the value is equal to the argument $\mathbf{fresh}_a$, and if not, the check is false and the value is irrelevant (though it still happens to be $\mathbf{fresh}_a$).

Tagging heuristics

Notice that the type tag in the rewrites for type declarations is just the name n. This is the least specific version of type tagging, which works well for short examples, but becomes

problematic in larger programs. Listing 5.7 gives an example of a case where two singletons would refer to the same type under this simple naming scheme. A more useful version of type tagging might encode details of the lexical context of the type directive into the type tag, such as a heuristic for finding a path from the root object. This could disambiguate the two singletons as `foo/x` and `bar/x`.

```
foo = {
    singleton x
}
bar = {
    singleton x
}
```

Listing 5.7: An example of conflicting type tags

No tagging scheme can be complete, since due to cross-references, there may be more than one path to any given type or singleton, but lexical approximations like those discussed here should be sufficient for many cases in practice.

5.6.3 Assert and effect directives

Both assert and effect directives can be viewed as helpers that make it easier to mix assert and effect expressions with other directives. Both can be eliminated via a simple rewrite to an assignment that has no effect, like a let directive that assigns to a fresh name.

$$\texttt{assert}\ e_c \longrightarrow \texttt{let}\ \textbf{fresh}\ = \texttt{assert}\ e_c;\ \{\}$$

$$\texttt{effect}\ e_1 = \cdots = e_n \longrightarrow \texttt{let}\ \textbf{fresh}\ = \texttt{effect}\ e_1 = \cdots = e_n;\ \{\}$$

5.7 Eliminating assertions

Assertions can be eliminated via a simple trick in two stages: first, use a conditional to wait on e_c and map it to either true or false (as opposed to directly using e_c, since merging it with `true` might affect the rest of the program's execution), then use an effect to merge the result of the conditional with `true`. If the merge succeeds ,that means the assertion was successful; if not, the program fails.

$$\texttt{assert}\ e_c;\ e_b \longrightarrow \texttt{effect}\ (\texttt{if}\ e_c\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \texttt{false}) = \texttt{true};\ e_b$$

Of course, a reasonable implementation would provide a native assert operation with a clear error message, but this shows that the formal model can express assertions.

5.8 Eliminating n-ary effect expressions

Core TreeGen only supports effect expressions that merge two subexpressions rather than the one-or-more supported by full TreeGen.

$$\texttt{effect}\ e_1 = \cdots = e_n;\ e_b \longrightarrow \begin{array}{l} \texttt{let}\ \mathbf{fresh}_e = e_1\ \texttt{in} \\ \texttt{effect}\ \mathbf{fresh}_e = e_2; \ldots; \texttt{effect}\ \mathbf{fresh}_e = e_n; \\ e_b \end{array}$$

The rewrite to eliminate these is quite simple – bind the first expression being merged to a fresh name $\mathbf{fresh}_e$, and emit as many effect expressions as needed to merge all the other expressions into the value bound to that fresh name.

Note that if there is only one single value to merge, then no effect expressions are emitted at all – the single value would just be bound to $\mathbf{fresh}_e$, with e_b as the let expression body.

5.9 Eliminating dollar names

Dollar names have a scoping effect in two positions: let expressions and assignment directives. Both of these scoping effects can be achieved using careful rewrites. Note that while dollar names can also be used in function parameters, they have no special effect there, and can already be considered as regular names.

$$\texttt{let}\ \$\texttt{n} = e_v\ \texttt{in}\ e_b \longrightarrow \texttt{let}\ \mathbf{fresh}_n = e_v\ \texttt{in}\ [\$n \to \mathbf{fresh}_n]e_b$$

For a let expression, a dollar name can be replaced with a plain name by replacing it with a fresh name in the context where it is accessible, in the let expression body. This ensures that mentions of the dollar name in e_v continue to refer to what they originally

referred to, while in e_b all mentions of $\$n$ will be replaced by a fresh regular name that cannot be in use anywhere else in the program, including in e_v.

$$\{d_1 \ldots;\ \texttt{let}\ \$n = e_v;\ d_2 \ldots\} \longrightarrow \begin{array}{l}\{d_1 \ldots;\ \texttt{let}\ \mathbf{fresh}_n = e_v; \\ \quad [\$n \to \mathbf{fresh}_n] d_2 \ldots\}\end{array}$$

The same goes for let directives assigning to dollar names – just rename the dollar name to a fresh normal name and rewrite all the following directives to use that instead.

Plain assignment directives are more subtle, since the resulting field must get the right name for the resulting object to remain the same. This means that for a dollar name $\$n$, the rewritten name has to be n on the left-hand side of the resulting assignment directive, forcing us to instead rewrite all existing mentions of `n` that might be affected.

$$\{d_1 \ldots;\ \$n = e_v;\ d_2 \ldots\} \overset{n \in \texttt{closure}(e)}{\longrightarrow} \begin{array}{l}\texttt{let}\ \mathbf{fresh}_n = \texttt{n}\ \texttt{in} \\ \quad \{[n \to \mathbf{fresh}_n] d_1 \ldots; \\ \quad\ n = [n \to \mathbf{fresh}_n] e_v; \\ \quad\ [n \to \mathbf{fresh}_n, \$n \to n] d_2 \ldots\}\end{array}$$

The potentially affected mentions of `n` lie in the right-hand sides of the directives d_1, as well as the right-hand side of the directive under consideration e_v. For d_2, the name $\$n$ is supposed to be in scope, so it's fine for n to refer to the bound value there (both $\$n$ and n would mean the same field n), but not in d_1 or e_v.

If n is in the overall expression's closure, then we need to preserve the original references in d_1 and e_v, so the rewrite given binds the outer value for n to $\mathbf{fresh}_n$ and rewrites d_1 and e_v to refer to that instead.

Note that this case implicitly excludes cases where any other directive assigns to the non-dollar name n – that would immediately remove n from the overall expression's closure. Note also that this would have a strange interaction with multiple assignments to $\$n$ – once the first assignment to $\$n$ has been rewritten, any others will reach the second case below, which is fine since all external references to n will have already been taken care of.

$$\{d_1 \ldots;\ \$n = e_v;\ d_2 \ldots\} \overset{n \notin \texttt{closure}(e)}{\longrightarrow} \{d_1 \ldots\ n = e_v;\ [\$n \to n] d_2 \ldots\}$$

If there are no external references to n at all, then either n is not referenced at all, or it is assigned to by one of the other directives in d_1 or d_2. If n is already assigned, then rewriting $\$n$ to n is fine, since both refer to the same field name. If n from this scope

is simply not mentioned at all, then it is equally safe to just rewrite all correctly scoped mentioned of $\$n$ to mention n instead.

5.10 Eliminating let directives

Now that dollar names have been eliminated and many other features have been expressed in whole or in part using let directives, we can eliminate this convenience. Dollar names being eliminated is important to this section, since we rely on full lexical order independence of directives, which is only true for a system as simple as what we're left with. We have only assignments and let directives with non-dollar names, so moving directives around should have no effect on scoping or program evaluation.

$$
\begin{aligned}
& (n, e_v) \in \mathbf{pairs}(d), [\![\texttt{let } \mathbf{fresh}_n = ??? \texttt{ in}]\!] \dots \\
& (n, e_v) \in \mathbf{pairs}(d), [\![\texttt{effect } \mathbf{fresh}_n = [n \to \mathbf{fresh}_n \dots] e_v; \ ]\!] \dots \\
\{d \dots\} \longrightarrow \{ & \\
& \quad [\![n = e_v]\!] \in d, [\![n = \mathbf{fresh}_n]\!] \dots \\
& \}
\end{aligned}
$$

To eliminate effect directives while allowing the right-hand sides to refer to any of the bound names, we have to rewrite every object expression by lifting every assignment (let or otherwise) out of the object expression. The notation $\mathbf{pairs}(d)$ is a shorthand for extracting each pair of name n and bound expression e_v from the sequence of directives d, regardless of whether n is bound as a let directive or not.

- First, each of these lifted assignments are given fresh names, and mapped to surrounding let expressions in order to have a name in scope for each bound name in the object expression.
- Then, the value bound by each assignment, e_v, is merged with the unknown value generated to represent represent it in the previous step. This ensures that each e_v can access an equivalent scope to the scope available to it before the rewrite.
- Finally, an object expression is generated that binds each fresh name corresponding to a non-let assignment to the right field name.

This was the last remaining feature to eliminate. With this, all rewrites necessary to move from full TreeGen to core TreeGen are given.

Chapter 6

Implementation strategy

When building an implementation for TreeGen, performance becomes a significant issue. Many of the formal semantics rely heavily on rewriting the entire program state to substitute identifiers, which means that when taken naively, execution of even a modestly complex program can become intractable. This section addresses this by describing the strategies and data structures used to build a real-world interpreter for the language, as well as some pitfalls we encountered.

We begin by documenting core data structures and ideas needed to efficiently represent program state, then provide some performance commentary alongside benchmarks illustrating some performance tradeoffs that require careful treatment. Lastly, we discuss the practicalities of providing usable execution traces to go with error messages, since due to TreeGen's unusual control flow, one cannot just rely on the traditional notion of a "stack trace".

6.1 Substitution via union-find

A core data structure used by our prototype to avoid performing substitutions is the union-find structure. Considering each value in TreeGen as a set, the merge operation can be framed as a "set union" between the two values with additional rules for merging the values' properties like shape, field relation, and call relation. It is shown by [34] that amortised performance of set union is proportional to the inverse Ackermann function, which is in practice indistinguishable from linear performance in the number of sets. This means that for all substitutions where the substituted value can be safely dereferenced through the

(a) The initial environment with three values @1, @2 and @3 and a field `x`

(b) The environment with @3 and @2 merged, preserving @2

Figure 6.1: A sequence of environments where just relying on union-find is sufficient

union-find structure on access, we have an effectively-constant time implementation of identifier substitution.

Note that to properly reach the efficiency result given above, we need the version of union-find that is weighted and implements path compression. This requires the merge operation to choose which of the left or right identifiers to keep as "representative" identifier using weight as a criterion, unlike the formal definition of the merge operation that just arbitrarily chooses to keep the right-hand side identifier.

To illustrate a situation where this strategy for substitutions works, consider the sequence of environments in figs. 6.1a and 6.1b, which describes a scenario between three values. For simplicity, we ignore the shapes of the individual values – all that matters is that they are mergeable. Each value has an identifier, notated @1, @2 or @3, and in the initial case, @1 has @3 as a field named "x". The labels @1, @2, and @3 give each node a particular identity, based on the natural number-like identifiers defined in chapter 4; they are necessary here because the data structures we discuss include explicit pointers based on these identities.

In the second environment, we have merged values @3 and @2, with @3 replacing @2. Since we are using union-find to perform substitutions, the value @2 remains, with the intended substitution notated as the added dashed arrow. In this case, no further work is required: since the replaced value is the field @3, we can reliably follow @1's field relation with name `x` to reach @3, at which point we can dereference the union-find pointer to get the actual substituted field value @2.

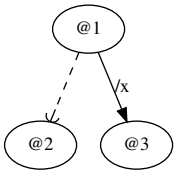


Figure 6.2: An alternate version of fig. 6.1b where @1 and @2 merged instead of @3 and @2, still preserving @2

6.1.1 Union-find is not sufficient in general

Not all substitutions are equal: some identifiers are used as keys in mappings, and the directed elements of some relations do not allow simply following the union-find pointers without some additional updating.

This is illustrated in the sequence of environments in figs. 6.1a and 6.2, which describes a different scenario of three values. The values are in the same starting configuration as previously, but in the second case we have merged @1 and @2 rather than @3 and @2, with @2 replacing @1. Since we are using the union-find structure, however, the "old" identifier @1 is still present, but notated as substituted out using the dashed arrow. This poses the problem that the existing field relation is still in terms of @1, even though the "representative" value is now supposed to be @2: the union-find pointer is not positioned in such a way that adding a union-find dereference to a query for @2's field `x` will allow the correct field value to be reached.

This and many cases like it motivate the following section's description of schemes for performing the minimum necessary substitutions to support the union-find algorithm.

6.2 Necessary substitutions and additional data structures

As illustrated in the previous section, while union-find takes care of many possible substitutions, there are several that require extra adjustments to account for relations that are either not fully preserved under union find, cause very inefficient merge operations that must scan more program state than necessary, or both.

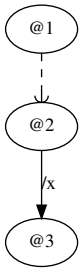


Figure 6.3: A corrected version of fig. 6.2, rewriting the field relation to come from the representative @2 rather than the defunct @1

6.2.1 Fields

A representative case where the union-find structure alone cannot maintain relational integrity for the fields relation is given by section 6.1.1. To ensure that the field relation's integrity is properly maintained during a union find-based merge between values i_1 and i_2, it is required that any instances of the field relation describing fields belonging to either i_1 or i_2 be rewritten to be in terms of the chosen representative value. Figure 6.3 illustrates this by giving a correctly rewritten version of fig. 6.1a with the field relation with name `x` correctly transferred to @2.

Note that the process for combining and generating follow-up merges for each value's fields is almost the same as that in the formal semantics. As stated formally only fields belonging to i_1 or i_2 need considering, and additionally the effects of the substitution from i_1 to i_2 can be achieved by taking the union of those fields without scanning the entire program state.

6.2.2 Function calls

The call relation has a similar problem to the field relation, but due to the relation being ternary, there are some added challenges to implementing it in practice.

Note that call fingerprinting mirrors the call relation in every way that matters, so this applies just as much to call fingerprinting as it does to the call relation.

Decomposition into binary mappings

To map the ternary calls relation onto readily available data structures, we have to split it up into binary mappings. This can be done by considering two separate starting points:

- Starting with a function argument, we want a mapping from functions that are called with the argument to the result of that call.
- Starting with a function, we want a mapping from arguments with which it was called to the result of that call.

At the expense of some repeated information, this could, for example, be achieved via nested hash maps from identifiers to identifiers. As with fields, for these purposes, we do not need to map back from function results to any aspect of a function call, so further mappings are unnecessary.

Necessary rewrites and follow-ups during merging

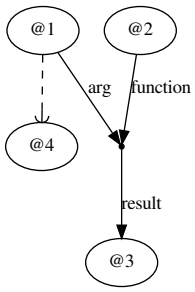
During merging, we need to both ensure that the keys in the mappings we defined above refer to representative values, and that we only scan the minimum necessary calls relations to determine the follow-up merges.

For the follow-up merges, we can significantly reduce what needs scanning by only considering functions or arguments that the two values have in common, while the necessary rewrites are best summarised graphically. Figures 6.4a and 6.4b illustrate the two problematic cases, while fig. 6.5 illustrates the necessary rewrites.

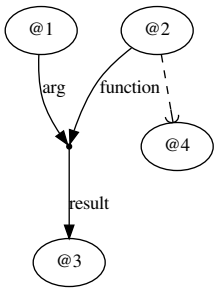
Note that no rewrites are needed if the call result is affected – as shown in fig. 6.6, it is possible to just follow the union-find pointer in this case, since it is not used as a key. As with the field case, when reaching a function result, it is safe to defererence the union-find pointer at that point, requiring no special rewrites for function results.

6.2.3 Object and function uniqueness

The approaches to implementing object and function uniqueness are intertwined, so we discuss them at the same time. Unlike the other two cases, there is more to this than just fixing the relevant relation or providing a simple decomposition of a ternary relation into binary mappings. During a merge operation, it is specified that once the shape relation

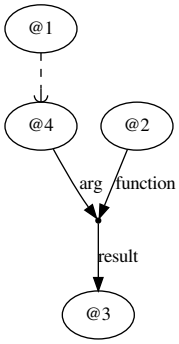


(a) The argument @1 has been rewritten to value @4

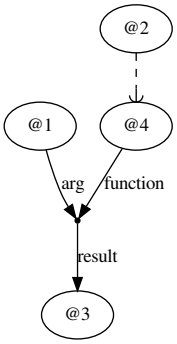


(b) The function @2 has been rewritten to value @4

Figure 6.4: Graphs of the two cases where a rewrite harms the integrity of the call relation



(a) Change the function call's argument from @1 to @4



(b) Change the function call's function from @2 to @4

Figure 6.5: Graphs of the necessary rewrites to figs. 6.4a and 6.4b respectively in order to restore integrity of the function call relation

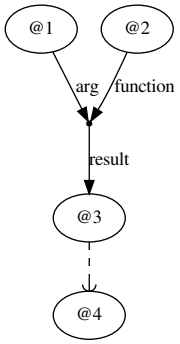


Figure 6.6: Graph showing that adding a rewrite to the result @3 of a function call can be accounted for by following the added union-find pointer

is rewritten, we must scan the entire shape relation for duplicate shapes and collect a set of follow-up merges. Since scanning the shape relation is work-wise equivalent to a full program state scan, we need a more practical way to identify follow-up merges stemming from duplicate function or object shapes.

The object trie

Using a trie structure it is possible to limit the scans necessary to generate follow-up merges to a structurally defined subset of possible shapes. This is done by considering each object shape to define a unique path from the empty object consisting of an ordered sequence of field-value pairs, choosing lexicographical ordering by field name as our ordering. The ordering does not directly use the value identifiers. The object trie stores these paths with the object values as leaves, facilitating their element by element deduplication: if a node in the trie is no longer unique, then its children are merged as far as possible. If that path merging process reaches a pair of leaves, then that pair should be included in a follow-up merge.

Likewise, when generating a new object shape, the trie can be searched in linear time relative to the object field count (that is, path length) to locate a potential existing object with that shape.

Note that this structure requires back-references from object field values to the trie node representing that field-value pair as well as back-references back "up" the trie in order for

merge operations to be able to navigate to all values that could lead to object-related follow-up merges.

The structure defined here has some ideas in common with the Self language's object storage system [2], in that each object has a pointer to a deduplicated "parent", and objects form a tree structure based on field names. Many details of the structure presented here differ, however, as the distinctions between TreeGen's complete value-by-value object deduplication and Self's efficient implementation of prototypal inheritance lead to divergent design decisions.

The overall structure is best illustrated graphically. Consider for instance fig. 6.7, which represents the object `{a = 1; b = 2; c = 3}`. This graph gives a precise illustration of each element of the data structure discussed so far:

- As a simplification, all the nodes in the trie are considered to be leaves, meaning they are all valid TreeGen values. This means that technically fig. 6.7 also represents objects `{a = 1; b = 2}`, `{a = 1}`, and `{}`.
- The trie structure itself is defined via the `fix` relation. The "forward" references along the path are keyed on the next field name and the field identifier. By the semantics of trie data structures, this enforces that for each field-value pair in an object shape, the object value holding that field-value pair as its "last" field-value pair (by ordering of field names) must be unique.

  This also identifies one of the cases where an identifier is used as a key, requiring an explicit rewrite to retain the trie's integrity.
- The "back" references along the trie, while drawn with the same arrows as the forward references, have no keys at all. Any node in the trie must by definition have exactly one parent.
- The `parent` relation from field values to owning trie nodes is encoded as a mapping from pairs of the field name and *the parent value's parent in the trie* to the parent value. Using this mapping to keep those pairs unique is a reformulation of the basic invariants for a trie structure relative to a field value: if two field values merge then their parents should only merge if they share the exact same predecessor in the trie. If this is the case, then the field being merged form a branch in the trie structure which needs unifying as part of the merge. If not, then the parents are part of distinct paths that remain distinct thanks to some other feature higher up in the trie.

There are several specific cases where rewrites are needed to repair the trie structure during a merge:

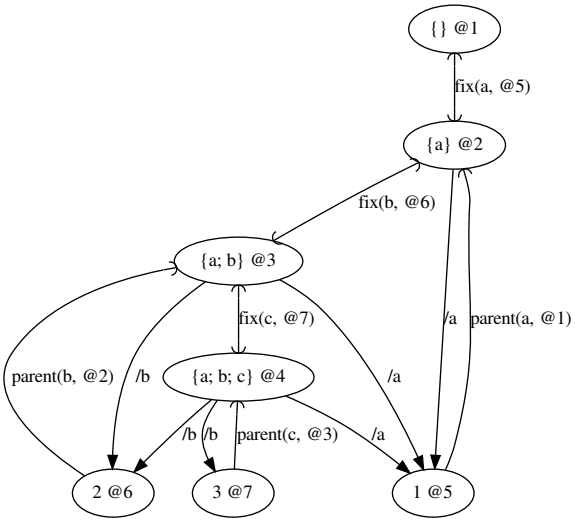
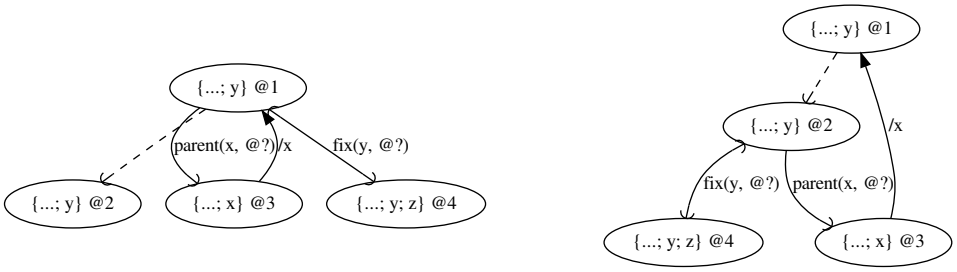


Figure 6.7: An example of an object trie, representing an object with three fields

- Just like the field relation, on rewrite, the `fix` and `parent` relations originating at the rewritten value need rewriting to point from the new representative value. Also like the field relation, values pointed-to do not need particular care during rewriting aside from potential follow-up merges. Figure 6.8 gives an abstract example of the involved rewrites, rewriting both the `parent` and `fix` relations originating from value @1 to originate from value @2 instead. Note that identifiers written @? are considered irrelevant to this scenario and set of rewrites – elaborations involving those will be discussed below.

- For each merged value that represents a node in the trie, its successors' (forward `fix` relations') fields' parent relations need to have their identifiers rewritten to refer to the new representative node. Figure 6.9 gives an example of a value whose parent relations need rewriting, @4, given substitution of value @1 with value @2. Notice how on top of the change to the `fix` relation fig. 6.9b shows that the `parent` relation refers to @2 instead of the now substituted-out value @1.

- For each merged field value that has parents in the trie, those parents' prefixes' successor relations need rewriting to refer to the new field value. Figure 6.10 gives an example of a value whose successors need rewriting, @4, given substitution of the



(a) Before repairing object trie

(b) After repairing object trie

Figure 6.8: An example of object trie repairs directly required by the subsitution of @1 by @2

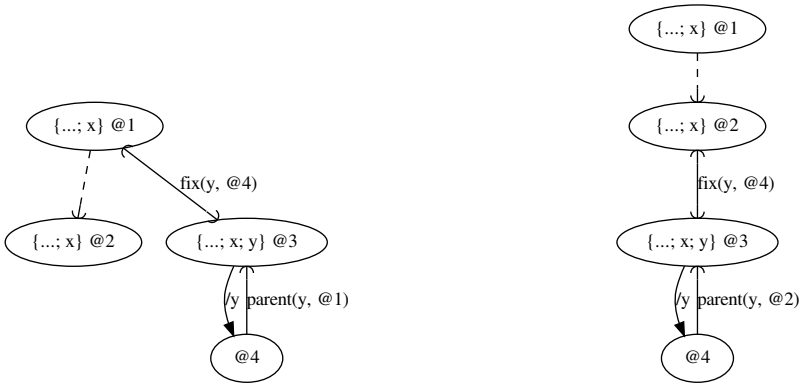
value @1 with value @2. Notice how in addition to the `parent` relation moving from @1 to @2, fig. 6.10b shows the `fix` relation starting at @4 rewritten to refer to the new representative identifier @2 for the field value.

Once the trie structure has been repaired after a given substitution, follow-up merges can also be found in specific places:

- The first place is duplicate keys in the `fix` relation and the `parent` relation pointing away from the merged values – the pairs of values referred to by the duplicate keys give the main set of follow-up merges.
- A less obvious secondary location is duplications caused by the rewrites that fixed the trie: it is possible that a key is rewritten to coincide with an existing key in one of the affected relations, in which case, the resulting pair of values also form a follow-up merge.

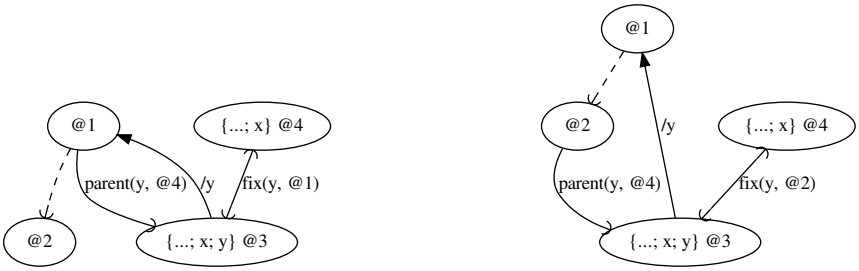
Adding functions to the object trie

Function values can be considered to lie at the end of the path defined by their closure object through the object trie. That way, they can be deduplicated according to the object trie rules without being object-shaped themselves. Figure 6.11 illustrates how the values defined in listing 6.1 would be represented in this way. Function values have no fields of their own, so the only relation needed is `fix`, providing a bidirectional link between the



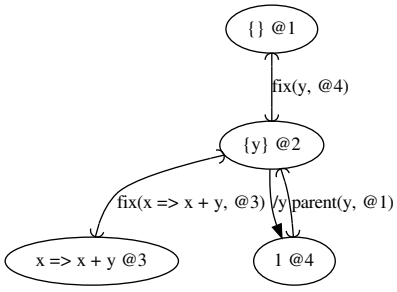
(a) Before repairing object trie (b) After repairing object trie

Figure 6.9: An example of needing to repair the indirect `parent` reference from @4 to substituted-out value @1



(a) Before repairing object trie (b) After repairing object trie

Figure 6.10: An example of needing to repair the indirect `fix` reference from @4 to substituted-out value @1




```
1 let y = 1
2 in \\ x => x + y
```

Listing 6.1: An example of a function value that captures another value in its closure object

Figure 6.11: The values represented by listing 6.1, laid out according to the object trie structure

function value @3 and its closure object @2. Additional rewriting rules are a strict subset of those required by object values.

6.3 Performance effects of execution ordering

While constraint and deferral execution has been proven to have the same result regardless of order, not all orderings are equivalent in terms of performance. Merging a single value with neither fields nor other relations can be considered effectively constant-time because of the union-find structure, but the work necessary to perform the rewrites and follow-up merges required by more general merges operations is not constant.

This gives rise to a performance concern: depending on when a merge constraint is scheduled to be executed, the exact configuration of the environment might be different, leading to more or less work being required to apply the constraint. Analytically, there are several cases where individual merges can have linear performance:

- Any pairs of values with fields will need to merge any coinciding fields as follow-up merges. Scanning the field sets to detect intersections also takes work.
- Any pairs of values involved in the calls relation (either as caller or callee) need to scan for function calls that would coincide were they to merge. The results and fingerprints of all coinciding function calls would need to be merged as a follow-up.

- Any pairs of values involved in the object trie need to scan for the several possible substitutions and necessary follow-up merges discussed alongside the introduction of the object trie.

This performance issue also appears in practice and can be shown by benchmarking TreeGen's prototype implementation, though some effects are hard to empirically separate due to the implementation's lack of direct ability to adjust execution schedules. We instead have to write code to specially exploit the implementation's quirks, which can lead to more than one factor affecting a measurement difference.

In any case, this section aims to illustrate that performance optimisation for TreeGen implementations is non-trivial and a potential avenue for future work.

TreeGen's implementation is written in Scala 2.13, and the benchmarks are performed using the JMH benchmarking suite [5] running on OpenJDK 12.0.2, on a machine running Linux kernel 5.4.51 with an 8-core AMD Ryzen 7 CPU and 16GB of RAM. Each benchmark scenario is run 25 times, with each batch of 5 runs taking place on a different forked JVM instance and prefixed by 5 warm-up iterations repeating the task for a minimum of 10 seconds.

6.3.1 Merging large objects

A relatively easy to isolate but somewhat confounding example of performance degradation due to merging is the scenario of merging two large objects (objects whose fields recursively form trees of other objects with many fields).

We do this by constructing listing 6.2, a setup with two recursive functions `useless1` and `useless2` that each construct deeply nested objects. `useless1` constructs the pattern `{ head = i; tail = { head = i - 1; tail = { ... } } }`, with the `head` field decreasing from `i` till `0`. `useless2` constructs a similar topology, but with the `head` fields being fresh unknown values instead of a descending series of integers. These fresh unknown values are entirely distinct from the integers in the tree defined by `useless1 (i)`, so while all the shapes should be compatible if merged – the trees have the same depth and field names – no uniqueness rules can apply to pre-emptively merge them.

This means that any of the variations given are free to merge them or not based on how they are written. Additionally, `done1` and `done2` remain unknown until `useless1` and `useless2`'s recursions have completed, giving us the option to force a merge to take place after the two trees have been fully constructed.

We benchmark three variations by adding different lines to the end of the common code:

noMerge As a baseline, we wait on `done1` and `done2` then do nothing.

mergeBefore We leave in the code to wait on `done1` and `done2` for parity with the other variations but merge values `a` and `b` immediately without giving them a chance to be fully elaborated.

mergeAfter We wait on `done1` and `done2`, forcing the recursive function calls to `useless1` and `useless2` to complete (the condition required for `done1` and `done2` to become known). This requires the final merge operation to merge the entire resulting object trees in one step.

Figure 6.12 shows the resulting performance of each variation from `i` = 100 to `i` = 10000. As one might expect, the variation with no merge at all is several seconds faster for large i since one might imagine less work would be required. What might be surprising however is how the other two variations have runtimes that are statistically indistinguishable.

The intuition behind this result is that even though we have done the merge before either `a` or `b` had any relationships to slow the merge down, this just pushes the work into each recursive call to `useless1` and `useless2`: we have to merge each `head` field no matter what, the distinction being whether we have to do it as `useless1` and `useless2` are executing or after.

The take-away here is that if your program is written with a constraint that merges the results of two different operations, the merge will take a similar amount of work no matter when it happens. Moving it earlier simply requires that the operations themselves perform multiple smaller merges that take about the same time as the alternative single, large merge.

```
{
    let done1 = ???
    let done2 = ???
    let useless1 = \\ $i =>
        if $i = 0
        then { effect done1 = true }
        else { head = $i; tail = useless1 ($i - 1) }
    let useless2 = \\ $i =>
        if $i = 0
        then { effect done2 = true }
```

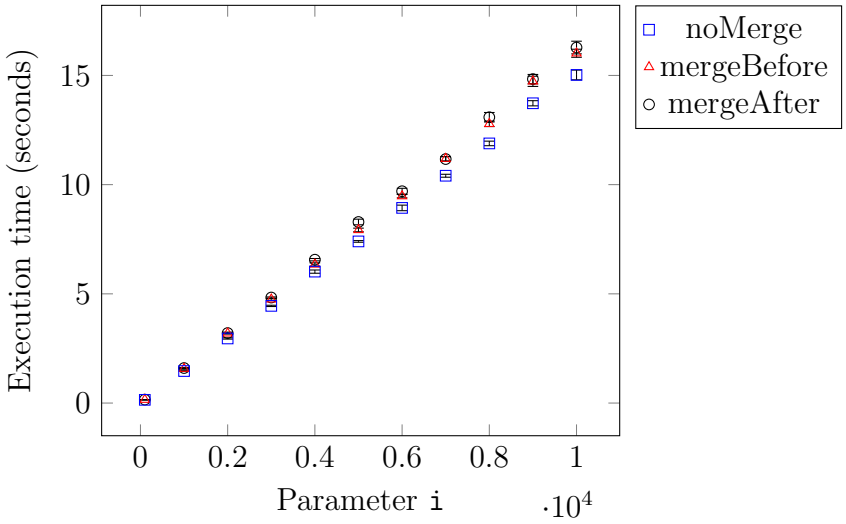


Figure 6.12: Plot of execution time of each variation of listing 6.2 against the integer benchmark parameter `i`

```
        else { head = ???; tail = useless2 ($i - 1) }

    let i = ??? // this is the benchmark parameter to vary object size

    let a = useless1 (i)
    let b = useless2 (i)

    // noMerge:
    // _ = if done1 && done2 then {} else {}
    // mergeBefore:
    // _ = if done1 && done2 then {} else {}; effect a = b
    // mergeAfter:
    // _ = if done1 && done2 then { effect a = b } else {}
}
```

Listing 6.2: A program aiming to demonstrate object merge performance

6.3.2 Interaction between function calls and the object trie

This subsection covers two related benchmarks that aim to demonstrate some potential interactions between function calls and the object trie. Each benchmark is based on a variation of listing 6.3, be that one of the commented out additions in that listing or the

further variations in listing 6.4.

The overall principle is to generate two functions `map1` and `map2` that can both be merged with one another, and used to map arbitrary values to unknown values. This is achieved using `mapmaker`, which itself maps any value to a unique function value – the trick is to force TreeGen to include `$idk` in the function's closure so that we get distinct function values, but to not actually use the value at all and return a fresh unknown value. Then, `useless1` and `useless2` perform recursions over the benchmark parameter `i`, but instead of constructing nested objects, they just create, then immediately discard, objects with fields `a` and `b`. Since each execution of `map1` and `map2` with some `$i` will generate a unique unknown value this will lead to `i` pairs of object values of the form `{ a = $i; b = map1 ($i) }` and `{ a = $i; b = map2 ($i) }` respectively. With this setup, we can now:

noMerge Let those objects be constructed and do nothing else, waiting on `done1` and `done2` for parity with the other variations.

mergeBefore Immediately merge `map1` and `map2` before letting any of `useless1` and `useless2` functions execute, waiting on `done1` and `done2` for parity with the other variations.

mergeAfter Wait on both `done1` and `done2` to be set, and therefore waiting on both `useless1` and `useless2` to have finished, then merge `map1` and `map2`, forcing all of their pairs of return values to merge and therefore combining `i` pairs of objects at the same time via the object trie.

```
{
    let mapmaker = \\\\ $idk => \\\\ $i => let _ = $idk in ???

    let map1 = mapmaker (???)
    let map2 = mapmaker (???)
    let useless1 = \\ $i =>
      let _ = { a = $i; b = map1 ($i) } in
      if i == 0 then { effect done1 = true } else useless1 ($i - 1)

    let useless2 = \\ $i =>
      let _ = { a = $i; b = map2 ($i) } in
      if i == 0 then { effect done2 = true } else useless2 ($i - 1)

    let i = ??? // benchmark parameter

    let done1 = ???
    _ = useless1 (i)

```

```
    let done2 = ???
    _ = useless2 (i)
    // noMerge:
    // _ = if done1 && done2 then {} else {}
    // mergeBefore:
    // _ = if done1 && done2 then {} else {}; effect map1 = map2
    // mergeAfter:
    // _ = if done1 && done2 then { effect map1 = map2 } else {}
}
```

Listing 6.3: A program aiming to demonstrate an interaction between functions, the object trie and merge performance

Listing 6.4 is a small variation on listing 6.3 that attempts to dissociate the effects of the object trie from the effects of function call merging by not binding the fields `b` to be results of `map1` or `map2`. This would prevent merging the two functions from affecting the object trie, which one might imagine would lead to faster execution times.

```
{
    let mapmaker = \\ $idk => \\ $i => let _ = $idk in ???

    let map1 = mapmaker (???)
    let map2 = mapmaker (???)
    let useless1 = \\ $i =>
      let _ = { a = $i; b = ??? } in let _ = map1 ($i) in
      if i == 0 then { effect done1 = true } else useless1 ($i - 1)

    let useless2 = \\ $i =>
      let _ = { a = $i; b = ??? } in let _ = map2 ($i) in
      if i == 0 then { effect done2 = true } else useless2 ($i - 1)

    let i = ??? // benchmark parameter

    let done1 = ???
    _ = useless1 (i)

    let done2 = ???
    _ = useless2 (i)
    // noMerge:
    // _ = if done1 && done2 then {} else {}
    // mergeBefore:
    // _ = if done1 && done2 then {} else {}; effect map1 = map2
    // mergeAfter:
    // _ = if done1 && done2 then { effect map1 = map2 } else {}
```

```
27 }
```

Listing 6.4: A program based in listing 6.3, varied to try and separate function call merging from the object trie

Figure 6.13 in fact shows that things are not so clear. We do clearly see that the variations of listing 6.3 affect performance, showing a clear distinction between each.

- The version with an early merge is actually fastest, since merging `map1` and `map2` ahead of time causes the program to require only half the number of function calls to be executed – this result suggests that at least in this case a heuristic that forces merge constraints to be executed as early as possible might provide practical benefits.
- The version that does not merge at all is next fastest, which can be explained by the intuition that performing twice the function calls but then skipping a costly merge operation is still faster than performing twice the function calls and then performing a costly merge operation. This in turn would also explain why the variation with a late merge is slowest, since it must do strictly more work.

What is perhaps confusing is that there is no appreciable difference between the execution times for benchmarks in listing 6.3 and benchmarks in listing 6.4. Each variation is very close to the corresponding one in listing 6.3, with the separated versions being very slightly slower for larger `i`.

This could be due to one of two main factors:

- The prototype may be noticeably slower at computing the slightly large function bodies and patterns present in `useless1` and `useless2` in listing 6.4. Note that the pattern implementation in the prototype is entirely unoptimised so that extra wildcard let binding will lead to redundant boolean comparisons (between TreeGen values, not native JVM types) and corresponding deferrals.
- Separating the function calls and the object trie might lead to more values being created. This is unlikely to be a significant issue since the prototype has a highly optimised value store based on integer indexing into an array, but should be considered.

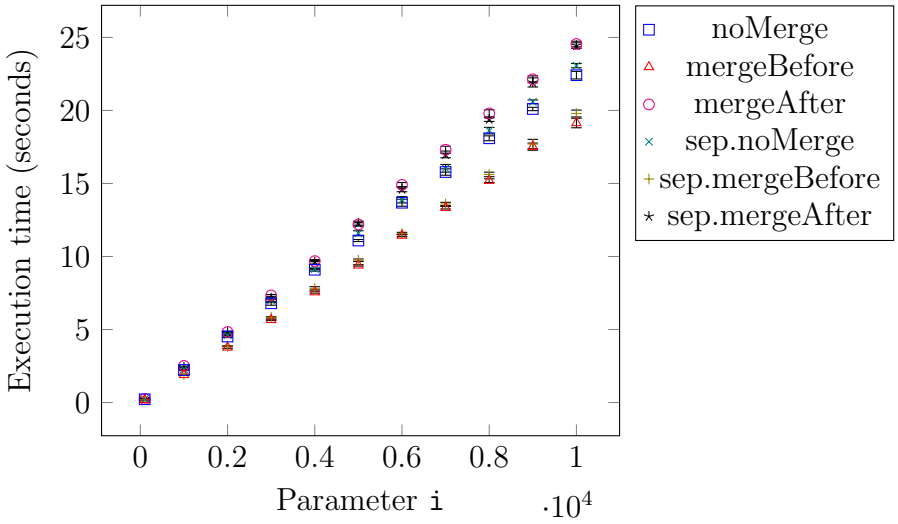


Figure 6.13: Plot of execution time of each variation of listing 6.2 against the integer benchmark parameter `i`

6.4 Execution tracing

When trying to debug or troubleshoot a program in TreeGen, it's important to be able to communicate to the programmer where something "is" in their code. From personal experience the most effective way to do this is in terms of lexically-directed paths from the main expression being evaluated. One exception is patterns and cases, where you not only want to know where you are but how you got there – "Why did the other patterns it tried fail?" has been a common issue when debugging. The other exception is deferrals that never resolve, where a graph representation is needed to determine which deferral(s) to report. This section discusses how an implementation could usefully handle these situations.

6.4.1 Object sub-expressions, bindings, and simple paths

An extremely common case for tracing is object expressions, where the most important information is which part of what assignment was being executed. Listing 6.5 for instance has an error in the bound value on line 3 (you can't add an object and an integer), which the implementation reports as shown in listing 6.6.

```
1 {
2     let foo = 1
```

```
    x = {} + 1
    y = 5
}
```

Listing 6.5: An object expression containing a deliberately incorrect sub-expression to demonstrate error tracing

```
#0 encountered unexpected shape: (&0) {} at 3.5 - 3.11 in error.tgen
    x = {} + 1
        ^^^^^^
#1 during assignment at 3.1 - 3.11 in error.tgen
    x = {} + 1
    ^^^^^^^^^^
#2 importing error.tgen at <internal: init>
```

Listing 6.6: The error produced by running listing 6.5

Conceptually, the same can be said of other kinds of expression containing bindings, such as let expressions.

Notice also that an implementation would need to correctly point out errors inside patterns on the left-hand side of the equals sign as well. Take for example listing 6.7 which showcases this kind of error – in this case, the prototype implementation reports the execution trace as in listing 6.8.

```
{
    let foo = {}
    x: foo/y = 12
    y = 5
}
```

Listing 6.7: An example of an incorrect pattern in an assignment directive

```
#0 invalid field `y` for value: (&0) {} at 3.4 - 3.9 in error.tgen
    x: foo/y = 12
       ^^^^^
#1 match pattern type at 3.4 - 3.9 in error.tgen
    x: foo/y = 12
       ^^^^^
#2 importing error.tgen at <internal: init>
```

Listing 6.8: The error produced by running listing 6.7

As a last kind of simple path, consider function calls: these can be listed mostly as one might expect, though the prototype implementation splits them into “try to call” and “called” in case the call fails due to the called value not being callable. Listing 6.9 gives an example of this, which would result in an error looking like listing 6.10.

```
{
    let boom = \\ _ => {} + 1
    _ = boom (???)
}
```

Listing 6.9: An example of an incorrect function call

```
#0 encountered unexpected shape: (&0) {} at 2.20 - 2.26 in error.tgen
    let boom = \\ _ => {} + 1
                       ^^^^^^
#1 called function at 2.12 - 2.26 in error.tgen
    let boom = \\ _ => {} + 1
               ^^^^^^^^^^^^^^
#2 calling function with argument: (&1) ??? {} at 3.5 - 3.15 in error.
   tgen
    _ = boom (???)
        ^^^^^^^^^^
#3 during assignment at 3.1 - 3.15 in error.tgen
    _ = boom (???)
    ^^^^^^^^^^^^^^
#4 importing error.tgen at <internal: init>
```

Listing 6.10: The error produced by running listing 6.9

6.4.2 Patterns and cases

Unlike with simple expressions where we just need to know lexically where we are, when diagnosing errors in patterns or cases, it's important to build up a history of what was tried in order to figure out what went wrong.

The prototype's approach to this is to build a list of where each pattern failed, including nested patterns. Consider the example in listing 6.11, where the error message listing 6.12 must track both how the match in the listing ran out of cases and how the nested match in `predef/list/unapply` ran out of cases. The result is verbose but complete, and has been useful when debugging large (1kLoc) TreeGen programs with complex nested control flow.

```
{
    let x = 12
    _ = x match {
        case _: predef/list => ???
    }
}
```

Listing 6.11: An example of a match expression that shows pattern failures across unapply boundaries

```
#0 match exhausted, could not match: (&0) 12 at 3.5 - 5.2 in error.tgen
        vvvvvvvvv
    _ = x match {
        case _: predef/list => ???
    }
    ^
#1 backtracking to next branch at 17.5 - 17.40 in <predef>
        case $list := _ :: _: list => $list
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#2 backtracking to next branch at 13.3 - 13.29 in <predef>
      type cons = { head; tail }
      ^^^^^^^^^^^^^^^^^^^^^^^^^
#3 unapplying with value: (&0) at 17.19 - 17.31 in <predef>
        case $list := _ :: _: list => $list
                      ^^^^^^^^^^^^
#4 match pattern type at 17.19 - 17.31 in <predef>
        case $list := _ :: _: list => $list
                      ^^^^^^^^^^^^
#5 checking case branch pattern at 17.10 - 17.31 in <predef>
        case $list := _ :: _: list => $list
             ^^^^^^^^^^^^^^^^^^^^^
#6 backtracking to next branch at 12.3 - 12.16 in <predef>
      singleton nil
      ^^^^^^^^^^^^^
#7 unapplying with value: (&0) at 16.22 - 16.25 in <predef>
        case $list := _: nil => $list
                        ^^^
#8 match pattern type at 16.22 - 16.25 in <predef>
        case $list := _: nil => $list
                        ^^^
#9 checking case branch pattern at 16.10 - 16.25 in <predef>
        case $list := _: nil => $list
             ^^^^^^^^^^^^^^^
#10 called function at 15.13 - 18.4 in <predef>
               vvvvvvvvvv
      unapply = \\ match {
    ...
      }
    ^^^
#11 unapplying with value: (&0) at 4.13 - 4.24 in error.tgen
        case _: predef/list => ???
               ^^^^^^^^^^^
#12 match pattern type at 4.13 - 4.24 in error.tgen
        case _: predef/list => ???
               ^^^^^^^^^^^
```

```
#13 checking case branch pattern at 4.10 - 4.24 in error.tgen
        case _: predef/list => ???
             ^^^^^^^^^^^^^^
#14 during assignment at 3.1 - 5.2 in error.tgen
    vvvvvvvvvvvvv
    _ = x match {
        case _: predef/list => ???
    }
    ^
#15 importing error.tgen at <internal: init>
```

Listing 6.12: The error produced by running listing 6.11

6.4.3 Starvation

Another relatively common error in TreeGen is accidentally making a deferral that can never resolve. This can be troublesome to report, since usually when you have one unresolved deferral, you have several that depend on each other.

The technique used by the prototype implementation is to collect all the starved deferrals while treating them as edges in a graph, linking values a deferral waits on and values that a deferral would make known if it executed. This graph may have cycles, but often many deferrals are part of an acyclic subgraph:

- In the case of an acyclic subgraph, an important intuition is that the programmer really needs to know about the deferrals that make up the roots: if another deferral happens to depend on a root, we don't even know if that's an error unless it persists beyond the error that starved the root deferral.

  That's why it's much more helpful to report only the root deferrals in case of starvation than the entire graph – the entire graph is likely full of deferrals that aren't even symptoms of a problem but rather just happen to depend on a root.

- If a deferral is part of a cycle then it is less straightforward – there is no canonical root to report, so we need to report all deferrals making up the cycle as errors. Perhaps a more advanced implementation could report the cycle's topology as well.

Listing 6.13 is an example of the former situation, where the only starvation is acyclic: it is because `a` will never become known that none of the other operations can ever succeed, so we should report only starved operations directly dependent on `a` such as that on line 3.

Line 4 is equally starved, but we don't need to mention that as we have already reported the cause of starvation more directly.

```
1 {
2     a = ???
3     b = a + 1
4     c = b + 1
5 }
```

Listing 6.13: An example of a program that leads to acyclic starvation

A cyclic starvation looks like listing 6.14, where there is no clear directionality to the starvation. `a` cannot be computed because `b` cannot be computed, and `b` cannot be computed because `a` cannot be computed. As a result, we have to report a starvation cycle alongside all involved deferrals and leave it to the programmer to untangle why some of their operations depend on themselves.

```
1 {
2     a = b + a
3     b = a + b
4 }
```

Listing 6.14: An example of a program that leads to cyclic starvation

6.5 Implementation-related conclusions

This chapter has explored several outcomes stemming from developing an implementation of the TreeGen language.

We have made some progress toward building a practically viable set of data structures to express TreeGen's environment substitutions, but both theoretical consideration of these data structures and empirical measurements show that some of the underlying substitutions can give the merge operation unpredictable performance, linear in environment size and dependent on the exact scheduling of constraint execution. This outcome seems to be unavoidable in general, but it might be useful to further investigate heuristics for building efficient constraint execution orderings. A different approach might also be possible, making minor changes to the TreeGen language that avoid, for instance, the full complexity and performance issues of maintaining the object trie.

On a different note, we have found reasonable success in developing an execution tracing model that can be used to display diagnostic information alongside runtime errors stemming

from the execution of TreeGen programs. This execution tracing logic has been anecdotally effective when debugging multi-file codebases of around 1KLoc, and we believe that its key mechanisms and semantics presented in this chapter may be useful as a basis for error reporting in further prototyping.

Chapter 7

Conclusion and Future Work

In this thesis we have presented the novel impure functional programming language TreeGen, a language designed to support the building of arbitrary code generation utilities, in the particular context of generating API bindings between an existing codebase and many different programming languages and platforms. Of fundamental interest, TreeGen demonstrates an apparently unique feature as a language: it allows a more reliable usage of explicit mutability than any language to which it has been compared.

TreeGen has been given both an informal introduction and a set of formal semantics that operate in terms of a graph-based structured heap and a constraint resolution mechanism. The structured heap provides a basis for the underlying ordering that is enforced on any mutations, and the constraint resolution mechanism has been shown to be deterministic when evaluated via chaotic iteration. This implies not only that any fair execution schedule of the same constraints will give the same result, but that any program that leads to the same set of constraints will also have the same result.

Lastly, we have presented key implementation strategies which we used to build a prototype evaluator. These strategies feature specialised adaptations to the union-find data structure in order provide as efficient a process for rewriting parts of the heap as possible, alongside recommendations for an execution tracing model that has been successfully used to debug TreeGen programs comprised of around 1000 lines of code across multiple files. While the execution tracing model has worked well, benchmarks of the implementation show that the data structures and constraint scheduling heuristics built into the implementation can result in problematic performance penalties in several cases.

7.1 Future work

In response to the performance issues found with TreeGen's current prototype interpreter, here are some future directions to consider:

- Better heuristics could be developed for ordering constraint evaluation, since currently the prototype simply uses an ad-hoc, implementation-defined schedule. Some insights gained from the benchmarks suggest, for example, that moving merge constraints to execute as early as possible might yield a noticeable performance benefit in some cases.
- Looking again at the proofs presented in chapter 4, it seems that the uniqueness conditions applied to non-$\perp$ shaped values might not be necessary conditions for the determinism proof to hold. Removing these conditions might yield a language that, while a little less predictable in terms of navigating recursive code that relies on perfect application of the existing uniqueness rules, might be easier to implement efficiently. This comes from the observation that one of the more complex and performance-impacting implementation constructs is the object trie, and that without it it might be much harder to accidentally form large chains of follow-up merges.

Performance issues nonwithstanding, it might be interesting to consider whether TreeGen would be useful for building deterministic configurations of things other than JSON-like documents and groups of text files. TreeGen's semantics could operate on anything that can be partially ordered relative to the language's existing primitives, and many interesting domains can be made to form useful partial orders.

References

[1] Dave Beazley and SWIG Team. Simplified wrapper and interface generator. http://www.swig.org/, 1996.

[2] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *ACM Sigplan Notices*, 24(10):49–70, 1989.

[3] Unison Computing. The unison language. https://www.unisonweb.org/. Accessed: 2020-07-16.

[4] MDN Contributors. Object.freeze(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze. Accessed: 2020-07-17.

[5] Oracle Corporation. OpenJDK: jmh. https://openjdk.java.net/projects/code-tools/jmh/. Accessed: 2020-07-18.

[6] ctypesgen developers. ctypesgen. https://github.com/davidjamesca/ctypesgen, 2015.

[7] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. Nix: A safe and policy-free system for software deployment. In *LISA*, volume 4, pages 79–92, 2004.

[8] Michael Dyck, Jonathan Robie, and Josh Spiegel. XML path language (XPath) 3.1. W3C recommendation, W3C, March 2017. https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

[9] eXist Solutions. eXist-db - the open-source native xml database. http://exist-db.org/exist/apps/homepage/index.html. Accessed: 2020-07-17.

[10] Apache Software Foundation. Thrift. https://thrift.apache.org/.

[11] Eclipse Software Foundation. Jakarta server pages. https://projects.eclipse.org/projects/ee4j.jsp. Accessed: 2020-07-17.

[12] Alfons Geser, Jens Knoop, Gerald Lüttgen, Oliver Rüthing, and Bernhard Steffen. Non-monotone fixpoint iterations to resolve second order effects. In Tibor Gyimóthy, editor, *Compiler Construction*, pages 106–118, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[13] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. Cross-language interoperability in a multi-language runtime. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(2):1–43, 2018.

[14] The PHP Group. PHP: Hypertext preprocessor. https://www.php.net/. Accessed: 2020-07-17.

[15] haskell gi. haskell-gi. https://github.com/haskell-gi/haskell-gi.

[16] Martin Hirzel and Robert Grimm. Jeannie: Granting java native interface developers their wishes. *ACM Sigplan Notices*, 42(10):19–38, 2007.

[17] Alphabet Inc. Android interface definition language (AIDL). https://developer.android.com/guide/components/aidl.

[18] Dropbox Inc. Djinni. https://www.github.com/dropbox/djinni.

[19] Facebook Inc. immutable.js. https://github.com/immutable-js/immutable-js. Accessed: 2020-07-15.

[20] Microsoft Inc. ASP.NET — open-source web framework for .NET. https://dotnet.microsoft.com/apps/aspnet. Accessed: 2020-07-17.

[21] Shopify Inc. Liquid template language. https://shopify.github.io/liquid/. Accessed: 2020-07-16.

[22] Yehuda Katz. Handlebars. https://handlebarsjs.com/. Accessed: 2020-07-16.

[23] Michael Kay. XSL transformations (XSLT) version 3.0. W3C recommendation, W3C, June 2017. https://www.w3.org/TR/2017/REC-xslt-30-20170608/.

[24] Emmanuel Lambert, Martin Fiers, Shavkat Nizamov, Martijn Tassaert, Steven G Johnson, Peter Bienstman, and Wim Bogaerts. Python bindings for the open source electromagnetic simulator meep. *Computing in Science & Engineering*, 13(3):53–65, 2010.

[25] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

[26] Jim Melton and John Snelson. XQuery update facility 3.0. W3C note, W3C, January 2017. https://www.w3.org/TR/2017/NOTE-xquery-update-30-20170124/.

[27] Alexander Mordvintsev and K. Abid. How opencv-python bindings works? https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_bindings/py_bindings_basics/py_bindings_basics.html#bindings-basics, 2013.

[28] Noel M O'Boyle, Chris Morley, and Geoffrey R Hutchison. Pybel: a python wrapper for the openbabel cheminformatics toolkit. *Chemistry Central Journal*, 2(1):1–7, 2008.

[29] Project Phoenix. Goals of project phoenix. https://wiki.wxpython.org/ProjectPhoenix/ProjectGoals, 2010.

[30] The GNOME Project. GObject. https://developer.gnome.org/platform-overview/stable/tech-gobject.html.en. Accessed: 2020-07-15.

[31] Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit. Automatic generation of library bindings using static analysis. *ACM Sigplan Notices*, 44(6):352–362, 2009.

[32] John Reppy and Chunyan Song. Application-specific foreign-interface generation. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 49–58, 2006.

[33] Josh Spiegel, Jonathan Robie, and Michael Dyck. XQuery 3.1: An XML query language. W3C recommendation, W3C, March 2017. https://www.w3.org/TR/2017/REC-xquery-31-20170321/.

[34] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984.

[35] SWIG Team. Open source projects using swig. http://www.swig.org/projects.html. Accessed: 2020-07-15.

[36] The LLDB Team. Architecture. https://lldb.llvm.org/use/architecture.html. Accessed: 2020-07-15.

[37] Chris Wanstrath. Mustache. http://mustache.github.io/. Accessed: 2020-07-16.

[38] May Wolfgang. XPath-logic and XPathLog: A logic-programming style XML data manipulation language. *Theory and Practice of Logic Programming*, 4(3):239–287, 2004.

[39] Jeremy Yallop, David Sheets, and Anil Madhavapeddy. Declarative foreign function binding through generic programming. In *International Symposium on Functional and Logic Programming*, pages 198–214. Springer, 2016.