Swift Documentation

Release 2.2

LLVM project

2017-11-11

Contents

1 Index Invalidation Rules in the Swift Standard Library 1

2 Access Control 5

3 Driver Design & Internals 9

4 Parseable Driver Output 13

5 Error Handling in Swift 2.0 17

6 Error Handling Rationale and Proposal 29

7 Generics in Swift 55

8 Logical Objects 69

9 Object Initialization 71

10 Pattern Matching 83

11 Stored and Computed Variables 95

12 Swift Intermediate Language (SIL) 101

13 Type Checker Design and Implementation 173

14 Debugging the Swift Compiler 187

CHAPTER 1

Index Invalidation Rules in the Swift Standard Library

1.1 Points to consider

1. Collections can be implemented as value types or a reference types.
2. Copying an instance of a value type, or copying a reference has well-defined semantics built into the language and is not controllable by the user code.

   Consequence: value-typed collections in Swift have to use copy-on-write for data stored out-of-line in reference-typed buffers.
3. We want to be able to pass/return a Collection along with its indices in a safe manner.

   In Swift, unlike C++, indices are not sufficient to access collection data; one needs an index and a collection. Thus, merely passing a collection by value to a function should not invalidate indices.

1.2 General principles

In C++, validity of an iterator is a property of the iterator itself, since iterators can be dereferenced to access collection elements.

In Swift, in order to access a collection element designated by an index, subscript operator is applied to the collection, `C[I]`. Thus, index is valid or not only in context of a certain collection instance at a certain point of program execution. A given index can be valid for zero, one or more than one collection instance at the same time.

An index that is valid for a certain collection designates an element of that collection or represents a one-past-end index.

Operations that access collection elements require valid indexes (this includes accessing using the subscript operator, slicing, swapping elements, removing elements etc.)

Using an invalid index to access elements of a collection leads to unspecified memory-safe behavior. (Possibilities include trapping, performing the operation on an arbitrary element of this or any other collection etc.) Concrete collection types can specify behavior; implementations are advised to perform a trap.

An arbitrary index instance is not valid for an arbitrary collection instance.

The following points apply to all collections, defined in the library or by the user:

1. Indices obtained from a collection `C` via `C.startIndex`, `C.endIndex` and other collection-specific APIs returning indices, are valid for `C`.
2. If an index `I` is valid for a collection `C`, a copy of `I` is valid for `C`.
3. If an index `I` is valid for a collection `C`, indices obtained from `I` via `I.successor()`, `I.predecessor()`, and other index-specific APIs, are valid for `C`. FIXME: disallow startIndex.predecessor(), endIndex.successor()
4. **Indices of collections and slices freely interoperate.**

   If an index `I` is valid for a collection `C`, it is also valid for slices of `C`, provided that `I` was in the bounds that were passed to the slicing subscript.

   If an index `I` is valid for a slice obtained from a collection `C`, it is also valid for `C` itself.
5. If an index `I` is valid for a collection `C`, it is also valid for a copy of `C`.
6. If an index `I` is valid for a collection `C`, it continues to be valid after a call to a non-mutating method on `C`.
7. Calling a non-mutating method on a collection instance does not invalidate any indexes.
8. Indices behave as if they are composites of offsets in the underlying data structure. For example:

   - an index into a set backed by a hash table with open addressing is the number of the bucket where the element is stored;
   - an index into a collection backed by a tree is a sequence of integers that describe the path from the root of the tree to the leaf node;
   - an index into a lazy flatMap collection consists of a pair of indices, an index into the base collection that is being mapped, and the index into the result of mapping the element designated by the first index.

   This rule does not imply that indices should be cheap to convert to actual integers. The offsets for consecutive elements could be non-consecutive (e.g., in a hash table with open addressing), or consist of multiple offsets so that the conversion to an integer is non-trival (e.g., in a tree).

   Note that this rule, like all other rules, is an "as if" rule. As long as the resulting semantics match what the rules dictate, the actual implementation can be anything.

   Rationale and discussion:

   - This rule is mostly motivated by its consequences, in particular, being able to mutate an element of a collection without changing the collection's structure, and, thus, without invalidating indices.
   - Replacing a collection element has runtime complexity O(1) and is not considered a structural mutation. Therefore, there seems to be no reason for a collection model would need to invalidate indices from the implementation point of view.
   - Iterating over a collection and performing mutations in place is a common pattern that Swift's collection library needs to support. If replacing individual collection elements would invalidate indices, many common algorithms (like sorting) wouldn't be implementable directly with indices; the code would need to maintain its own shadow indices, for example, plain integers, that are not invalidated by mutations.

Consequences:

- The setter of `MutableCollection.subscript(_: Index)` does not invalidate any indices. Indices are composites of offsets, so replacing the value does not change the shape of the data structure and preserves offsets.
- A value type mutable linked list can not conform to `MutableCollectionType`. An index for a linked list has to be implemented as a pointer to the list node to provide O(1) element access. Mutating an element of a

non-uniquely referenced linked list will create a copy of the nodes that comprise the list. Indices obtained before the copy was made would point to the old nodes and wouldn't be valid for the copy of the list.

It is still valid to have a value type linked list conform to `CollectionType`, or to have a reference type mutable linked list conform to `MutableCollection`.

The following points apply to all collections by default, but specific collection implementations can be less strict:

1. A call to a mutating method on a collection instance, except the setter of `MutableCollection.subscript(_: Index)`, invalidates all indices for that collection instance.

Consequences:

- Passing a collection as an `inout` argument invalidates all indexes for that collection instance, unless the function explicitly documents stronger guarantees. (The function can call mutating methods on an `inout` argument or completely replace it.)
  - `Swift.swap()` does not invalidate any indexes.

1.3 Additional guarantees for `Swift.Array`, `Swift.ContiguousArray`, `Swift.ArraySlice`

Valid array indexes can be created without using Array APIs. Array indexes are plain integers. Integers that are dynamically in the range `0..<A.count` are valid indexes for the array or slice `A`. It does not matter if an index was obtained from the collection instance, or derived from input or unrelated data.

Traps are guaranteed. Using an invalid index to designate elements of an array or an array slice is guaranteed to perform a trap.

1.4 Additional guarantees for `Swift.Dictionary`

Insertion into a Dictionary invalidates indexes only on a rehash. If a `Dictionary` has enough free buckets (guaranteed by calling an initializer or reserving space), then inserting elements does not invalidate indexes.

Note: unlike C++'s `std::unordered_map`, removing elements from a `Dictionary` invalidates indexes.

CHAPTER 2

Access Control

The general guiding principle of Swift access control:

> **No entity can be defined in terms of another entity that has a lower access level.**

There are three levels of access: "private", "internal", and "public". Private entities can only be accessed from within the source file where they are defined. Internal entities can be accessed anywhere within the module they are defined. Public entities can be accessed from anywhere within the module and from any other context that imports the current module.

The names `public` and `private` have precedent in many languages; `internal` comes from C#. In the future, `public` may be used for both API and SPI, at which point we may design additional annotations to distinguish the two.

By default, most entities in a source file have `internal` access. This optimizes for the most common case—a single-target application project—while not accidentally revealing entities to clients of a framework module.

- *Rules*
  - *Globals and Members*
  - *Protocols*
  - *Structs, Enums, and Classes*
  - *Types*
- *Runtime Guarantees*
  - *Interaction with Objective-C*
- *Non-Goals: "class-only" and "protected"*
- *Potential Future Directions*

2.1 Rules

Access to a particular entity is considered relative to the current *access context*. The access context of an entity is the current file (if `private`), the current module (if `internal`), or the current program (if `public`). A reference to an entity may only be written within the entity's access context.

If a particular entity is not accessible, it does not appear in name lookup, unlike in C++. However, access control does not restrict access to members via runtime reflection (where applicable), nor does it necessarily restrict visibility of symbols in a linked binary.

2.1.1 Globals and Members

A global function, constant, or variable may have any access level less than or equal to the access level of its type. That is, a `private` constant can have `public` type, but not the other way around.

Accessors for variables have the same access level as their associated variable. The setter may be explicitly annotated with an access level less than or equal to the access level of the variable; this is written as `private(set)` or `internal(set)` before the `var` introducer.

An initializer, method, subscript, or property may have any access level less than or equal to the access level of its type (including the implicit 'Self' type), with a few additional rules:

- If the type's access level is `private`, the access level of members defaults to `private`. If the type's access level is `internal` or `public`, the access level of members defaults to `internal`.
- If a member is used to satisfy a protocol requirement, its access level must be at least as high as the protocol conformance's; see *Protocols* below.
- If an initializer is `required` by a superclass, its access level must be at least as high as the access level of the subclass itself.
- Accessors for subscripts follow the same rules as accessors for variables.
- A member may be overridden whenever it is accessible.

The implicit memberwise initializer for a struct has the minimum access level of all of the struct's stored properties, except that if all properties are `public` the initializer is `internal`. The implicit no-argument initializer for structs and classes follows the default access level for the type.

Currently, enum cases always have the same access level as the enclosing enum.

Deinitializers are only invoked by the runtime and do not nominally have access. Internally, the compiler represents them as having the same access level as the enclosing type.

2.1.2 Protocols

A protocol may have any access level less than or equal to the access levels of the protocols it refines. That is, a `private` ExtendedWidget protocol can refine an `public` Widget protocol, but not the other way around.

The access level of a requirement is the access level of the enclosing protocol, even when the protocol is `public`. Currently, requirements may not be given a lower access level than the enclosing protocol.

Swift does not currently support private protocol conformances, so for runtime consistency, the access level of the conformance of type T to protocol P is equal to the minimum of T's access level and P's access level; that is, the conformance is accessible whenever both T and P are accessible. This does not change if the protocol is conformed to in an extension. (The access level of a conformance is not currently reflected in the source, but is a useful concept for applying restrictions consistently.)

All members used to satisfy a conformance must have an access level at least as high as the conformance's. This ensures consistency between views of the type; if any member has a *lower* access level than the conformance, then the member could be accessed anyway through a generic function constrained by the protocol.

Note: This rule disallows an `internal` member of a protocol extension to satisfy a `public` requirement for a `public` type. Removing this limitation is not inherently unsafe, but (a) may be unexpected given the lack of explicit reference to the member, and (b) results in references to non-public symbols in the current representation.

A protocol may be used as a type whenever it is accessible. A nominal can conform to a protocol whenever the protocol is accessible.

2.1.3 Structs, Enums, and Classes

A struct, enum, or class may be used as a type whenever it is accessible. A struct, enum, or class may be extended whenever it is accessible.

A class may be subclassed whenever it is accessible. A class may have any access level less than or equal to the access level of its superclass.

Members in an extension have the same default access level as members declared within the extended type. However, an extension may be marked with an explicit access modifier (e.g. `private extension`), in which case the default access level of members within the extension is changed to match.

Extensions with explicit access modifiers may not add new protocol conformances, since Swift does not support private protocol conformances (see *Protocols* above).

A type may conform to a protocol with lower access than the type itself.

2.1.4 Types

A nominal type's access level is the same as the access level of the nominal declaration itself. A generic type's access level is the minimum of the access level of the base type and the access levels of all generic argument types.

A tuple type's access level is the minimum of the access levels of its elements. A function type's access level is the minimum of the access levels of its input and return types.

A typealias may have any access level up to the access level of the type it aliases. That is, a `private` typealias can refer to an `public` type, but not the other way around. This includes associated types used to satisfy protocol conformances.

2.2 Runtime Guarantees

Non-`public` members of a class or extension will not be seen by subclasses or other extensions from outside the module. Therefore, members of a subclass or extension will not conflict with or inadvertently be considered to override non-accessible members of the superclass.

Both `private` and `internal` increase opportunities for devirtualization, though it is still possible to put a subclass of a `private` class within the same file.

Most information about a non-`public` entity still has to be put into a module file for now, since we don't have resilience implemented. This can be improved later, and is no more revealing than the information currently available in the runtime for pure Objective-C classes.

2.2.1 Interaction with Objective-C

If an entity is exposed to Objective-C, most of the runtime guarantees and optimization opportunities go out the window. We have to use a particular selector for members, everything can be inspected at runtime, and even a private member can cause selector conflicts. In this case, access control is only useful for discipline purposes.

Members explicitly marked `private` are *not* exposed to Objective-C unless they are also marked `@objc` (or `@IBAction` or similar), even if declared within a class implicitly or explicitly marked `@objc`.

Any `public` entities will be included in the generated header. In an application or unit test target, `internal` entities will be exposed as well.

2.3 Non-Goals: "class-only" and "protected"

This proposal omits two forms of access control commonly found in other languages, a "class-implementation-only" access (often called "private"), and a "class and any subclasses" access (often called "protected"). We chose not to include these levels of access control because they do not add useful functionality beyond `private`, `internal`, and `public`.

"class-only" If "class-only" includes extensions of the class, it is clear that it provides no protection at all, since a class may be extended from any context where it is accessible. So a hypothetical "class-only" must already be limited with regards to extensions. Beyond that, however, a "class-only" limit forces code to be declared within the class that might otherwise naturally be a top-level helper or an extension method on another type.

`private` serves the proper use case of limiting access to the implementation details of a class (even from the rest of the module!) while not requiring that all of those implementation details be written lexically inside the class.

"protected" "protected" access provides no guarantees of information hiding, since any subclass can now access the implementation details of its superclass—and expose them publicly, if it so chooses. This interacts poorly with our future plans for resilient APIs. Additionally, it increases the complexity of the access control model for both the compiler and for developers, and like "class-only" it is not immediately clear how it interacts with extensions.

Though it is not compiler-enforced, members that might be considered "protected" are effectively publicly accessible, and thus should be marked `public` in Swift. They can still be documented as intended for overriding rather than for subclassing, but the specific details of this are best dealt with on a case-by-case basis.

2.4 Potential Future Directions

- Allowing `private` or `internal` protocol conformances, which are only accessible at compile-time from a particular access context.
- Limiting particular capabilities, such as marking something `final(public)` to restrict subclassing or overriding outside of the current module.
- Allowing the Swift parts of a mixed-source framework to access private headers.
- Revealing `internal` Swift API in a mixed-source framework in a second generated header.
- Levels of `public`, for example `public("SPI")`.
- Enum cases less accessible than the enum.
- Protocol requirements less accessible than the protocol.

CHAPTER 3

Driver Design & Internals

- *Introduction*
- *Driver Stages*
  - *Parse: Option parsing*
  - *Pipeline: Converting Args into Actions*
  - *Build: Translating Actions into Jobs using a ToolChain*
  - *Execute: Running the Jobs in a Compilation using a TaskQueue*

3.1 Introduction

This document serves to describe the high-level design of the Swift 2.0 compiler driver (which includes what the driver is intended to do, and the approach it takes to do that), as well as the internals of the driver (which is meant to provide a brief overview of and rationale for how the high-level design is implemented).

The Swift driver is not intended to be GCC/Clang compatible, as it does not need to serve as a drop-in replacement for either driver. However, the design of the driver is inspired by Clang's design.

3.2 Driver Stages

The compiler driver for Swift roughly follows the same design as Clang's compiler driver:

1. Parse: Command-line arguments are parsed into `Args`. A ToolChain is selected based on the current platform.
2. Pipeline: Based on the arguments and inputs, a tree of `Actions` is generated. These are the high-level processing steps that need to occur, such as "compile this file" or "link the output of all compilation actions".

3. Bind: The ToolChain converts the `Actions` into a set of `Jobs`. These are individual commands that need to be run, such as "ld main.o -o main". Jobs have dependencies, but are not organized into a tree structure.
4. Execute: The `Jobs` are run in a `Compilation`, which spawns off sub-processes for each job that needs execution. The `Compilation` is responsible for deciding which `Jobs` actually need to run, based on dependency information provided by the output of each sub-process. The low-level management of sub-processes is handled by a `TaskQueue`.

3.2.1 Parse: Option parsing

The command line arguments are parsed as options and inputs into Arg instances. Some miscellaneous validation and normalization is performed. Most of the implementation is provided by LLVM.

An important part of this step is selecting a ToolChain. This is the Swift driver's view of the current platform's set of compiler tools, and determines how it will attempt to accomplish tasks. More on this below.

One of the optional steps here is building an *output file map*. This allows a build system (such as Xcode) to control the location of intermediate output files. The output file map uses a simple JSON format mapping inputs to a map of output paths, keyed by file type. Entries under an input of "" refer to the top-level driver process.

FIXME

Certain capabilities, like incremental builds or compilation without linking, currently require an output file map. This should not be necessary.

3.2.2 Pipeline: Converting Args into Actions

At this stage, the driver will take the input Args and input files and establish a graph of Actions. This details the high-level tasks that need to be performed. The graph (a DAG) tracks dependencies between actions, but also manages ownership.

FIXME

Actions currently map one-to-one to sub-process invocations. This means that there are actions for things that should be implementation details, like generating dSYM output.

3.2.3 Build: Translating Actions into Jobs using a ToolChain

Once we have a graph of high-level Actions, we need to translate that into actual tasks to execute. This starts by determining the output that each Action needs to produce based on its inputs. Then we ask the ToolChain how to perform that Action on the current platform. The ToolChain produces a Job, which wraps up both the output information and the actual invocation. It also remembers which Action it came from and any Jobs it depends on. Unlike the Action graph, Jobs are owned by a single Compilation object and stored in a flat list.

When a Job represents a compile of a single file, it may also be used for dependency analysis, to determine whether it is safe to not recompile that file in the current build. This is covered by checking if the input has been modified since the last build; if it hasn't, we only need to recompile if something it depends on has changed.

3.2.4 Execute: Running the Jobs in a Compilation using a TaskQueue

A Compilation's goal is to make sure every Job in its list of Jobs is handled. If a Job needs to be run, the Compilation attempts to *schedule* it. If the Job's dependencies have all been completed (or determined to be skippable), it is added to the TaskQueue; otherwise it is marked as *blocked.*

To support Jobs compiling individual Swift files, which may or may not need to be run, the Compilation keeps track of a DependencyGraph. (If file A depends on file B and file B has changed, file A needs to be recompiled.) When a Job completes successfully, the Compilation will both re-attempt to schedule Jobs that were directly blocked on it, and check to see if any other Jobs now need to run based on the DependencyGraph. See the section on Dependency Analysis for more information.

The Compilation's TaskQueue controls the low-level aspects of managing subprocesses. Multiple Jobs may execute simultaneously, but communication with the parent process (the driver) is handled on a single thread. The level of parellelism may be controlled by a compiler flag.

If a Job does not finish successfully, the Compilation needs to record which jobs have failed, so that they get rebuilt next time the user tries to build the project.

CHAPTER 4

Parseable Driver Output

- *Introduction*
- *Message Format*
- *Message Kinds*
  - *Began Message*
  - *Finished Message*
  - *Signalled Message*
  - *Skipped Message*
- *Message Names*

4.1 Introduction

This document serves to describe the parseable output format provided by the Swift compiler driver with the "-parseable-output" flag. This output format is intended to be parsed by other programs; one such use case is to allow an IDE to construct a detailed log based on the commands the driver issued.

4.2 Message Format

The parseable output provided by the Swift driver is provided as messages encoded in JSON objects. All messages are structured like this:

```
<Message Length>\n
{
  "kind": "<Message Kind>",
```

```
  "name": "<Message Name>",
  "<key>": "<value>",
  ...
}\n
```

This allows the driver to pass as much information as it wants about the ongoing compilation, and programs which parse this data can decide what to use and what to ignore.

4.3 Message Kinds

- *Began Message*
- *Finished Message*
- *Signalled Message*
- *Skipped Message*

The driver may emit four kinds of messages: "began", "finished", "signalled", and "skipped".

4.3.1 Began Message

A "began" message indicates that a new task began. As with all task-based messages, it will include the task's PID under the "pid" key. It may specify the task's inputs as an array of paths under the "inputs" key. It may specify the task's outputs as an array of objects under the "outputs" key. An "outputs" object will have two fields, a "kind" describing the type of the output, and a "path" containing the path to the output. A "began" message will specify the command which was executed under the "command" key.

Example:

```
{
  "kind": "began",
  "name": "compile",
  "pid": 12345,
  "inputs": [ "/src/foo.swift" ],
  "outputs": [
     {
       "type": "object",
       "path": "/build/foo.o"
     },
     {
       "type": "swiftmodule",
       "path": "/build/foo.swiftmodule"
     },
     {
       "type": "diagnostics",
       "path": "/build/foo.dia"
     },
  ],
  "command": "swift -frontend -c -primary-file /src/foo.swift /src/bar.swift -emit-
↪module-path /build/foo.swiftmodule -emit-diagnostics-path /build/foo.dia"
}
```

4.3.2 Finished Message

A "finished" message indicates that a task finished execution. As with all task- based messages, it will include the task's PID under the "pid" key. It will include the exit status of the task under the "exit-status" key. It may include the stdout/stderr of the task under the "output" key; if this key is missing, no output was generated by the task.

Example:

```
{
  "kind": "finished",
  "name": "compile",
  "pid": 12345,
  "exit-status": 0
  // "output" key omitted because there was no stdout/stderr.
}
```

4.3.3 Signalled Message

A "signalled" message indicates that a task exited abnormally due to a signal. As with all task-based message, it will include the task's PID under the "pid" key. It may include an error message describing the signal under the "error-message" key. As with the "finished" message, it may include the stdout/stderr of the task under the "output" key; if this key is missing, no output was generated by the task.

Example:

```
{
  "kind": "signalled",
  "name": "compile",
  "pid": 12345,
  "error-message": "Segmentation fault: 11"
  // "output" key omitted because there was no stdout/stderr.
}
```

4.3.4 Skipped Message

A "skipped" message indicates that the driver determined a command did not need to run during the current compilation. A "skipped" message is equivalent to a "began" message, with the exception that it does not include the "pid" key.

Example:

```
{
  "kind": "skipped",
  "name": "compile",
  "inputs": [ "/src/foo.swift" ],
  "outputs": [
     {
       "type": "object",
       "path": "/build/foo.o"
     },
     {
       "type": "swiftmodule",
       "path": "/build/foo.swiftmodule"
     },
     {
```

```
      "type": "diagnostics",
      "path": "/build/foo.dia"
    },
  ],
  "command": "swift -frontend -c -primary-file /src/foo.swift /src/bar.swift -emit-
↪module-path /build/foo.swiftmodule -emit-diagnostics-path /build/foo.dia"
}
```

4.4 Message Names

The name of the message identifies the kind of command the message describes. Some valid values are:

- compile
- merge-module
- link
- generate-dsym

A "compile" message represents a regular Swift frontend command. A "merge-module" message represents an invocation of the Swift frontend which is used to merge partial swiftmodule files into a complete swiftmodule. A "link" message indicates that the driver is invoking the linker to produce an executable or a library. A "generate-dsym" message indicates that the driver is invoking dsymutil to generate a dSYM.

Parsers of this format should be resilient in the event of an unknown name, as the driver may emit messages with new names whenever it needs to execute a new kind of command.

CHAPTER 5

Error Handling in Swift 2.0

As a tentpole feature for Swift 2.0, we are introducing a new first-class error handling model. This feature provides standardized syntax and language affordances for throwing, propagating, catching, and manipulating recoverable error conditions.

Error handling is a well-trod path, with many different approaches in other languages, many of them problematic in various ways. We believe that our approach provides an elegant solution, drawing on the lessons we've learned from other languages and fixing or avoiding some of the pitfalls. The result is expressive and concise while still feeling explicit, safe, and familiar; and we believe it will work beautifully with the Cocoa APIs.

We're intentionally not using the term "exception handling", which carries a lot of connotations from its use in other languages. Our proposal has some similarities to the exceptions systems in those languages, but it also has a lot of important differences.

5.1 Kinds of Error

What exactly is an "error"? There are many possible error conditions, and they don't all make sense to handle in exactly the same way, because they arise in different circumstances and programmers have to react to them differently.

We can break errors down into four categories, in increasing order of severity:

A **simple domain error** arises from an operation that can fail in some obvious way and which is often invoked speculatively. Parsing an integer from a string is a really good example. The client doesn't need a detailed description of the error and will usually want to handle the error immediately. These errors are already well-modeled by returning an optional value; we don't need a more complex language solution for them.

A **recoverable error** arises from an operation which can fail in complex ways, but whose errors can be reasonably anticipated in advance. Examples including opening a file or reading from a network connection. These are the kinds of errors that Apple's APIs use NSError for today, but there are close analogues in many other APIs, such as `errno` in POSIX.

Ignoring this kind of error is usually a bad idea, and it can even be dangerous (e.g. by introducing a security hole). Developers should be strongly encouraged to write code that handles the error. It's common for developers to want to

handle errors from different operations in the same basic way, either by reporting the error to the user or passing the error back to their own clients.

These errors will be the focus on this proposal.

The final two classes of error are outside the scope of this proposal. A **universal error** is theoretically recoverable, but by its nature the language can't help the programmer anticipate where it will come from. A **logic failure** arises from a programmer mistake and should not be recoverable at all. In our system, these kinds of errors are reported either with Objective-C/C++ exceptions or simply by logging a message and calling `abort()`. Both kinds of error are discussed extensively in the rationale. Having considered them carefully, we believe that we can address them in a later release without significant harm.

5.2 Aspects of the Design

This approach proposed here is very similar to the error handling model manually implemented in Objective-C with the `NSError` convention. Notably, the approach preserves these advantages of this convention:

- Whether a method produces an error (or not) is an explicit part of its API contract.
- Methods default to *not* producing errors unless they are explicitly marked.
- The control flow within a function is still mostly explicit: a maintainer can tell exactly which statements can produce an error, and a simple inspection reveals how the function reacts to the error.
- Throwing an error provides similar performance to allocating an error and returning it – it isn't an expensive, table-based stack unwinding process.
- Cocoa APIs using standard `NSError` patterns can be imported into this world automatically. Other common patterns (e.g. `CFError`, `errno`) can be added to the model in future versions of Swift.

In addition, we feel that this design improves on Objective-C's error handling approach in a number of ways:

- It eliminates a lot of boilerplate control-flow code for propagating errors.
- The syntax for error handling will feel familiar to people used to exception handling in other languages.
- Defining custom error types is simple and ties in elegantly with Swift enums.

As to basic syntax, we decided to stick with the familiar language of exception handling. We considered intentionally using different terms (like `raise` / `handle`) to try to distinguish our approach from other languages. However, by and large, error propagation in this proposal works like it does in exception handling, and people are inevitably going to make the connection. Given that, we couldn't find a compelling reason to deviate from the `throw` / `catch` legacy.

This document just contains the basic proposal and will be very light on rationale. We considered many different languages and programming environments as part of making this proposal, and there's an extensive discussion of them in the separate rationale document. For example, that document explains why we don't simply allow all functions to throw, why we don't propagate errors using simply an `ErrorOr<T>` return type, and why we don't just make error propagation part of a general monad feature. We encourage you to read that rationale if you're interested in understanding why we made the decisions we did.

With that out of the way, let's get to the details of the proposal.

5.3 Typed propagation

Whether a function can throw is part of its type. This applies to all functions, whether they're global functions, methods, or closures.

By default, a function cannot throw. The compiler statically enforces this: anything the function does which can throw must appear in a context which handles all errors.

A function can be declared to throw by writing `throws` on the function declaration or type:

```
func foo() -> Int {  // This function is not permitted to throw.
func bar() throws -> Int {   // This function is permitted to throw.
```

`throws` is written before the arrow to give a sensible and consistent grammar for function types and implicit `()` result types, e.g.:

```
func baz() throws {

// Takes a 'callback' function that can throw.
// 'fred' itself can also throw.
func fred(callback: (UInt8) throws -> ()) throws {

// These are distinct types.
let a : () -> () -> ()
let b : () throws -> () -> ()
let c : () -> () throws -> ()
let d : () throws -> () throws -> ()
```

For curried functions, `throws` only applies to the innermost function. This function has type `(Int) -> (Int) throws -> Int`:

```
func jerry(i: Int)(j: Int) throws -> Int {
```

`throws` is tracked as part of the type system: a function value must also declare whether it can throw. Functions that cannot throw are a subtype of functions that can, so you can use a function that can't throw anywhere you could use a function that can:

```
func rachel() -> Int { return 12 }
func donna(generator: () throws -> Int) -> Int { ... }

donna(rachel)
```

The reverse is not true, since the caller would not be prepared to handle the error.

A call to a function which can throw within a context that is not allowed to throw is rejected by the compiler.

It isn't possible to overload functions solely based on whether the functions throw. That is, this is not legal:

```
func foo() {
func foo() throws {
```

A throwing method cannot override a non-throwing method or satisfy a non-throwing protocol requirement. However, a non-throwing method can override a throwing method or satisfy a throwing protocol requirement.

It is valuable to be able to overload higher-order functions based on whether an argument function throws, so this is allowed:

```
func foo(callback: () throws -> Bool) {
func foo(callback: () -> Bool) {
```

5.3.1 rethrows

Functions which take a throwing function argument (including as an autoclosure) can be marked as `rethrows`:

```
extension Array {
  func map<U>(fn: ElementType throws -> U) rethrows -> [U]
}
```

It is an error if a function declared `rethrows` does not include a throwing function in at least one of its parameter clauses.

`rethrows` is identical to `throws`, except that the function promises to only throw if one of its argument functions throws.

More formally, a function is *rethrowing-only* for a function *f* if:

- it is a throwing function parameter of *f*,
- it is a non-throwing function, or
- it is implemented within *f* (i.e. it is either *f* or a function or closure defined therein) and it does not throw except by either:
  - calling a function that is rethrowing-only for *f* or
  - calling a function that is `rethrows`, passing only functions that are rethrowing-only for *f*.

It is an error if a `rethrows` function is not rethrowing-only for itself.

A `rethrows` function is considered to be a throwing function. However, a direct call to a `rethrows` function is considered to not throw if it is fully applied and none of the function arguments can throw. For example:

```
// This call to map is considered not to throw because its
// argument function does not throw.
let absolutePaths = paths.map { "/" + $0 }

// This call to map is considered to throw because its
// argument function does throw.
let streams = try absolutePaths.map { try InputStream(filename: $0) }
```

For now, `rethrows` is a property of declared functions, not of function values. Binding a variable (even a constant) to a function loses the information that the function was `rethrows`, and calls to it will use the normal rules, meaning that they will be considered to throw regardless of whether a non-throwing function is passed.

For the purposes of override and conformance checking, `rethrows` lies between `throws` and non-`throws`. That is, an ordinary throwing method cannot override a `rethrows` method, which cannot override a non-throwing method; but an ordinary throwing method can be overridden by a `rethrows` method, which can be overridden by a non-throwing method. Equivalent rules apply for protocol conformance.

5.4 Throwing an error

The `throw` statement begins the propagation of an error. It always take an argument, which can be any value that conforms to the `ErrorType` protocol (described below).

```
if timeElapsed > timeThreshold {
  throw HomeworkError.Overworked
}

throw NSError(domain: "whatever", code: 42, userInfo: nil)
```

As mentioned above, attempting to throw an error out of a function not marked `throws` is a static compiler error.

5.5 Catching errors

A `catch` clause includes an optional pattern that matches the error. This pattern can use any of the standard pattern-matching tools provided by `switch` statements in Swift, including boolean `where` conditions. The pattern can be omitted; if so, a `where` condition is still permitted. If the pattern is omitted, or if it does not bind a different name to the error, the name `error` is automatically bound to the error as if with a `let` pattern.

The `try` keyword is used for other purposes which it seems to fit far better (see below), so `catch` clauses are instead attached to a generalized `do` statement:

```
// Simple do statement (without a trailing while condition),
// just provides a scope for variables defined inside of it.
do {
   let x = foo()
}

// do statement with two catch clauses.
do {
  ...

} catch HomeworkError.Overworked {
  // a conditionally-executed catch clause

} catch _ {
  // a catch-all clause.
}
```

As with `switch` statements, Swift makes an effort to understand whether catch clauses are exhaustive. If it can determine it is, then the compiler considers the error to be handled. If not, the error automatically propagates out out of scope, either to a lexically enclosing `catch` clause or out of the containing function (which must be marked `throws`).

We expect to refine the `catch` syntax with usage experience.

5.6 `ErrorType`

The Swift standard library will provide `ErrorType`, a protocol with a very small interface (which is not described in this proposal). The standard pattern should be to define the conformance of an `enum` to the type:

```
enum HomeworkError : ErrorType {
  case Overworked
  case Impossible
  case EatenByCat(Cat)
  case StopStressingMeWithYourRules
}
```

The `enum` provides a namespace of errors, a list of possible errors within that namespace, and optional values to attach to each option.

Note that this corresponds very cleanly to the `NSError` model of an error domain, an error code, and optional user data. We expect to import system error domains as enums that follow this approach and implement `ErrorType`. `NSError` and `CFError` themselves will also conform to `ErrorType`.

The physical representation (still being nailed down) will make it efficient to embed an `NSError` as an `ErrorType` and vice-versa. It should be possible to turn an arbitrary Swift `enum` that conforms to `ErrorType` into an `NSError`

by using the qualified type name as the domain key, the enumerator as the error code, and turning the payload into user data.

5.7 Automatic, marked, propagation of errors

Once an error is thrown, Swift will automatically propagate it out of scopes (that permit it), rather than relying on the programmer to manually check for errors and do their own control flow. This is just a lot less boilerplate for common error handling tasks. However, doing this naively would introduce a lot of implicit control flow, which makes it difficult to reason about the function's behavior. This is a serious maintenance problem and has traditionally been a considerable source of bugs in languages that heavily use exceptions.

Therefore, while Swift automatically propagates errors, it requires that statements and expressions that can implicitly throw be marked with the `try` keyword. For example:

```
func readStuff() throws {
  // loadFile can throw an error.  If so, it propagates out of readStuff.
  try loadFile("mystuff.txt")

  // This is a semantic error; the 'try' keyword is required
  // to indicate that it can throw.
  var y = stream.readFloat()

  // This is okay; the try covers the entire statement.
  try y += stream.readFloat()

  // This try applies to readBool().
  if try stream.readBool() {
    // This try applies to both of these calls.
    let x = try stream.readInt() + stream.readInt()
  }

  if let err = stream.getOutOfBandError() {
    // Of course, the programmer doesn't have to mark explicit throws.
    throw err
  }
}
```

Developers can choose to "scope" the `try` very tightly by writing it within parentheses or on a specific argument or list element:

```
// Ok.
let x = (try stream.readInt()) + (try stream.readInt())

// Semantic error: the try only covers the parenthesized expression.
let x2 = (try stream.readInt()) + stream.readInt()

// The try applies to the first array element.  Of course, the
// developer could cover the entire array by writing the try outside.
let array = [ try foo(), bar(), baz() ]
```

Some developers may wish to do this to make the specific throwing calls very clear. Other developers may be content with knowing that something within a statement can throw. The compiler's fixit hints will guide developers towards inserting a single `try` that covers the entire statement. This could potentially be controlled someday by a coding style flag passed to the compiler.

5.7.1 `try!`

To concisely indicate that a call is known to not actually throw at runtime, `try` can be decorated with `!`, turning the error check into a runtime assertion that the call does not throw.

For the purposes of checking that all errors are handled, a `try!` expression is considered to handle any error originating from within its operand.

`try!` is otherwise exactly like `try`: it can appear in exactly the same positions and doesn't affect the type of an expression.

5.8 Manual propagation and manipulation of errors

Taking control over the propagation of errors is important for some advanced use cases (e.g. transporting an error result across threads when synchronizing a future) and can be more convenient or natural for specific use cases (e.g. handling a specific call differently within a context that otherwise allows propagation).

As such, the Swift standard library should provide a standard Rust-like `Result<T>` enum, along with API for working with it, e.g.:

- A function to evaluate an error-producing closure and capture the result as a `Result<T>`.
- A function to unpack a `Result<T>` by either returning its value or propagating the error in the current context.

This is something that composes on top of the basic model, but that has not been designed yet and details aren't included in this proposal.

The name `Result<T>` is a stand-in and needs to be designed and reviewed, as well as the basic operations on the type.

5.9 `defer`

Swift should provide a `defer` statement that sets up an *ad hoc* clean-up action to be run when the current scope is exited. This replicates the functionality of a Java-style `finally`, but more cleanly and with less nesting.

This is an important tool for ensuring that explicitly-managed resources are released on all paths. Examples include closing a network connection and freeing memory that was manually allocated. It is convenient for all kinds of error-handling, even manual propagation and simple domain errors, but is especially nice with automatic propagation. It is also a crucial part of our long-term vision for universal errors.

`defer` may be followed by an arbitrary statement. The compiler should reject a `defer` action that might terminate early, whether by throwing or with `return`, `break`, or `continue`.

Example:

```
if exists(filename) {
  let file = open(filename, O_READ)
  defer close(file)

  while let line = try file.readline() {
    ...
  }

  // close occurs here, at the end of the formal scope.
}
```

If there are multiple defer statements in a scope, they are guaranteed to be executed in reverse order of appearance. That is:

```
let file1 = open("hello.txt")
defer close(file1)
let file2 = open("world.txt")
defer close(file2)
...
// file2 will be closed first.
```

A potential extension is to provide a convenient way to mark that a defer action should only be taken if an error is thrown. This is a convenient shorthand for controlling the action with a flag. We will evaluate whether adding complexity to handle this case is justified based on real-world usage experience.

5.10 Importing Cocoa

If possible, Swift's error-handling model should transparently work with the SDK with a minimal amount of effort from framework owners.

We believe that we can cover the vast majority of Objective-C APIs with `NSError**` out-parameters by importing them as `throws` and removing the error clause from their signature. That is, a method like this one from `NSAttributedString`:

```
- (NSData *)dataFromRange:(NSRange)range
       documentAttributes:(NSDictionary *)dict
                    error:(NSError **)error;
```

would be imported as:

```
func dataFromRange(range: NSRange,
                   documentAttributes dict: NSDictionary) throws -> NSData
```

There are a number of cases to consider, but we expect that most can be automatically imported without extra annotation in the SDK, by using a couple of simple heuristics:

- The most common pattern is a `BOOL` result, where a false value means an error occurred. This seems unambiguous.
- Also common is a pointer result, where a `nil` result usually means an error occurred. This appears to be universal in Objective-C; APIs that can return `nil` results seem to do so via out-parameters. So it seems to be safe to make a policy decision that it's okay to assume that a `nil` result is an error by default.

  If the pattern for a method is that a `nil` result means it produced an error, then the result can be imported as a non-optional type.
- A few APIs return `void`. As far as I can tell, for all of these, the caller is expected to check for a non-`nil` error.

For other sentinel cases, we can consider adding a new clang attribute to indicate to the compiler what the sentinel is:

- There are several APIs returning `NSInteger` or `NSUInteger`. At least some of these return 0 on error, but that doesn't seem like a reasonable general assumption.
- `AVFoundation` provides a couple methods returning `AVKeyValueStatus`. These produce an error if the API returned `AVKeyValueStatusFailed`, which, interestingly enough, is not the zero value.

The clang attribute would specify how to test the return value for an error. For example:

```
+ (NSInteger)writePropertyList:(id)plist
                      toStream:(NSOutputStream *)stream
                        format:(NSPropertyListFormat)format
                       options:(NSPropertyListWriteOptions)opt
                         error:(out NSError **)error
  NS_ERROR_RESULT(0);

- (AVKeyValueStatus)statusOfValueForKey:(NSString *)key
                                  error:(NSError **)
  NS_ERROR_RESULT(AVKeyValueStatusFailed);
```

We should also provide a Clang attribute which specifies that the correct way to test for an error is to check the out-parameter. Both of these attributes could potentially be used by the static analyzer, not just Swift. (For example, they could try to detect an invalid error check.)

Cases that do not match the automatically imported patterns and that lack an attribute would be left unmodified (i.e., they'd keep their NSErrorPointer argument) and considered "not awesome" in the SDK auditing tool. These will still be usable in Swift: callers will get the NSError back like they do today, and have to throw the result manually.

For initializers, importing an initializer as throwing takes precedence over importing it as failable. That is, an imported initializer with a nullable result and an error parameter would be imported as throwing. Throwing initializers have very similar constraints to failable initializers; in a way, it's just a new axis of failability.

One limitation of this approach is that we need to be able to reconstruct the selector to use when an overload of a method is introduced. For this reason, the import is likely to be limited to methods where the error parameter is the last one and the corresponding selector chunk is either `error:` or the first chunk (see below). Empirically, this seems to do the right thing for all but two sets of APIs in the public API:

- The `ISyncSessionDriverDelegate` category on `NSObject` declares half-a-dozen methods like this:

  ```
  - (BOOL)sessionDriver:(ISyncSessionDriver *)sender
          didRegisterClientAndReturnError:(NSError **)outError;
  ```

  Fortunately, these delegate methods were all deprecated in Lion, and are thus unavailable in Swift.

- `NSFileCoordinator` has half a dozen methods where the `error:` clause is second-to-last, followed by a block argument. These methods are not deprecated as far as I know.

The above translation rule would import methods like this one from `NSDocument`:

```
- (NSDocument *)duplicateAndReturnError:(NSError **)outError;
```

like so:

```
func duplicateAndReturnError() throws -> NSDocument
```

The `AndReturnError` bit is common but far from universal; consider this method from `NSManagedObject`:

```
- (BOOL)validateForDelete:(NSError **)error;
```

This would be imported as:

```
func validateForDelete() throws
```

This is a really nice import, and it's somewhat unfortunate that we can't import `duplicateAndReturnError:` as `duplicate()`.

5.11 Potential future extensions to this model

We believe that the proposal above is sufficient to provide a huge step forward in error handling in Swift programs, but there is always more to consider in the future. Some specific things we've discussed (and may come back to in the future) but don't consider to be core to the Swift 2.0 model are:

5.11.1 Higher-order polymorphism

We should make it easy to write higher-order functions that behave polymorphically with respect to whether their arguments throw. This can be done in a fairly simple way: a function can declare that it throws if any of a set of named arguments do. As an example (using strawman syntax):

```
func map<T,U>(array: [T], fn: T -> U) throwsIf(fn) -> [U] {
  ...
}
```

There's no need for a more complex logical operator than disjunction for normal higher-order stuff.

This feature is highly desired (e.g. it would allow many otherwise redundant overloads to be collapsed into a single definition), but it may or may not make it into Swift 2.0 based on schedule limitations.

5.11.2 Generic polymorphism

For similar reasons to higher-order polymorphism, we should consider making it easier to parameterize protocols on whether their operations can throw. This would allow the writing of generic algorithms, e.g. over `Sequence`, that handle both conformances that cannot throw (like `Array`) and those that can (like a hypothetical cloud-backed implementation).

However, this would be a very complex feature, yet to be designed, and it is far out-of-scope for Swift 2.0. In the meantime, most standard protocols will be written to not allow throwing conformances, so as to not burden the use of common generic algorithms with spurious error-handling code.

5.11.3 Statement-like functions

Some functions are designed to take trailing closures that feel like sub-statements. For example, `autoreleasepool` can be used this way:

```
autoreleasepool {
  foo()
}
```

The error-handling model doesn't cause major problems for this. The compiler can infer that the closure throws, and `autoreleasepool` can be overloaded on whether its argument closure throws; the overload that takes a throwing closures would itself throw.

There is one minor usability problem here, though. If the closure contains throwing expressions, those expression must be explicitly marked within the closure with `try`. However, from the compiler's perspective, the call to `autoreleasepool` is also a call that can throw, and so it must also be marked with `try`:

```
try autoreleasepool {    // 'try' is required here...
  let string = try parseString() // ...and here.
  ...
}
```

This marking feels redundant. We want functions like `autoreleasepool` to feel like statements, but marks inside builtin statements like `if` don't require the outer statement to be marked. It would be better if the compiler didn't require the outer `try`.

On the other hand, the "statement-like" story already has a number of other holes: for example, `break`, `continue`, and `return` behave differently in the argument closure than in statements. In the future, we may consider fixing that; that fix will also need to address the error-propagation problem.

5.11.4 `using`

A `using` statement would acquire a resource, holds it for a fixed period of time, optionally binds it to a name, and then releases it whenever the controlled statement exits. `using` has many similarities to `defer`. It does not subsume `defer`, which is useful for many ad-hoc and tokenless clean-ups. But it could be convenient for the common pattern of a type-directed clean-up.

5.11.5 Automatically importing CoreFoundation and C functions

CF APIs use `CFErrorRef` pretty reliably, but there are several problems here: 1) the memory management rules for CFErrors are unclear and potentially inconsistent. 2) we need to know when an error is raised.

In principle, we could import POSIX functions into Swift as throwing functions, filling in the error from `errno`. It's nearly impossible to imagine doing this with an automatic import rule, however; much more likely, we'd need to wrap them all in an overlay.

In both cases, it is possible to pull these into the Swift error handling model, but because this is likely to require massive SDK annotations it is considered out of scope for iOS 9/OSX 10.11 & Swift 2.0.

5.11.6 Unexpected and universal errors

As discussed above, we believe that we can extend our current model to support untyped propagation for universal errors. Doing this well, and in particular doing it without completely sacrificing code size and performance, will take a significant amount of planning and insight. For this reason, it is considered well out of scope for Swift 2.0.

CHAPTER 6

Error Handling Rationale and Proposal

This paper surveys the error-handling world, analyzes various ideas which have been proposed or are in practice in other languages, and ultimately proposes an error-handling scheme for Swift together with import rules for our APIs.

6.1 Fundamentals

I need to establish some terminology first.

6.1.1 Kinds of propagation

I've heard people talk about **explicit vs. implicit propagation**. I'm not going to use those terms, because they're not helpful: there are at least three different things about error-handling that can be more or less explicit, and some of the other dimensions are equally important.

The most important dimensions of variation are:

- Whether the language allows functions to be designated as producing errors or not; such a language has **typed propagation**.
- Whether, in a language with typed propagation, the default rule is that that a function can produce an error or that it can't; this is the language's **default propagation rule**.
- Whether, in a language with typed propagation, the language enforces this statically, so that a function which cannot produce an error cannot call a function which can without handling it; such a language has **statically-enforced typed propagation**. (A language could instead enforce this dynamically by automatically inserting code to assert if an error propagates out. C++ does this.)
- Whether the language requires all potential error sites to be identifiable as potential error sites; such a language has **marked propagation**.
- Whether propagation is done explicitly with the normal data-flow and control-flow tools of the language; such a language has **manual propagation**. In contrast, a language where control implicitly jumps from the original error site to the proper handler has **automatic propagation**.

6.1.2 Kinds of error

What is an error? There may be many different possible error conditions in a program, but they can be categorized into several kinds based on how programmers should be expected to react to them. Since the programmer is expected to react differently, and since the language is the tool of the programmer's reaction, it makes sense for each group to be treated differently in the language.

To be clear, in many cases the kind of error reflects a conscious decision by the author of the error-producing code, and different choices might be useful in different contexts. The example I'm going to use of a "simple domain error" could easily be instead treated as a "recoverable error" (if the author expected automatic propagation to be more useful than immediate recovery) or even a "logic failure" (if the author wished to prevent speculative use, e.g. if checking the precondition was very expensive).

In order of increasing severity and complexity:

Simple domain errors

A simple domain error is something like calling `String.toInt()` on a string that isn't an integer. The operation has an obvious precondition about its arguments, but it's useful to be able to pass other values to test whether they're okay. The client will often handle the error immediately.

Conditions like this are best modeled with an optional return value. They don't benefit from a more complex error-handling model, and using one would make common code unnecessarily awkward. For example, speculatively trying to parse a `String` as an integer in Java requires catching an exception, which is far more syntactically heavyweight (and inefficient without optimization).

Because Swift already has good support for optionals, these conditions do not need to be a focus of this proposal.

Recoverable errors

Recoverable errors include file-not-found, network timeouts, and similar conditions. The operation has a variety of possible error conditions. The client should be encouraged to recognize the possibility of the error condition and consider the right way to handle it. Often, this will be by aborting the current operation, cleaning up after itself if needed, and propagating the error to a point where it can more sensibly be handled, e.g. by reporting it to the user.

These are the error conditions that most of our APIs use `NSError` and `CFError` for today. Most libraries have some similar notion. This is the focus of this proposal.

Universal errors

The difference between universal errors and ordinary recoverable errors is less the kind of error condition and more the potential sources of the error in the language. An error is universal if it could arise from such a wealth of different circumstances that it becomes nearly impracticable for the programmer to directly deal with all the sources of the error.

Some conditions, if they are to be brought into the scope of error-handling, can only conceivably be dealt with as universal errors. These include:

- Asynchronous conditions, like the process receiving a `SIGINT`, or the current thread being cancelled by another thread. These conditions could, in principle, be delivered at an arbitrary instruction boundary, and handling them appropriately requires extraordinary care from the programmer, the compiler, and the runtime.
- Ubiquitous errors, like running out of memory or overflowing the stack, that essentially any operation can be assumed to potentially do.

But other kinds of error condition can essentially become universal errors with the introduction of abstraction. Reading the size of a collection, or reading a property of an object, is not an operation that a programmer would normally expect

to produce an error. However, if the collection is actually backed by a database, the database query might fail. If the user must write code as if any opaque abstraction might produce an error, they are stuck in the world of universal errors.

Universal errors mandate certain language approaches. Typed propagation of universal errors is impossible, other than special cases which can guarantee to not produce errors. Marked propagation would provoke a user revolt. Propagation must be automatic, and the implementation must be "zero-cost", or as near to it as possible, because checking for an error after every single operation would be prohibitive.

For these reasons, in our APIs, universal error conditions are usually implemented using Objective-C exceptions, although not all uses of Objective-C exceptions fall in this category.

This combination of requirements means that all operations must be implicitly "unwindable" starting from almost any call site it makes. For the stability of the system, this unwinding process must restore any invariants that might have been temporarily violated; but the compiler cannot assist the programmer in this. The programmer must consciously recognize that an error is possible while an invariant is broken, and they must do this proactively — that, or track it down when they inevitably forget. This requires thinking quite rigorously about one's code, both to foresee all the error sites and to recognize that an important invariant is in flux.

How much of a problem this poses depends quite a lot on the code being written. There are some styles of programming that make it pretty innocuous. For example, a highly functional program which conscientiously kept mutation and side-effects to its outermost loops would naturally have very few points where any invariants were in flux; propagating an error out of an arbitrary place within an operation would simply abandon all the work done up to that point. However, this happy state falls apart quite quickly the more that mutation and other side-effects come into play. Complex mutations cannot be trivially reversed. Packets cannot be unsent. And it would be quite amazing for us to assert that code shouldn't be written that way, understanding nothing else about it. As long as programmers do face these issues, the language has some responsibility to help them.

Therefore, in my judgement, promoting the use of universal errors is highly problematic. They undermine the easy comprehension of code, and they undermine the language's ability to help the programmer reason about errors. This design will instead focus on explicitly trackable errors of the sort that `NSError` is used for today on Apple platforms.

However, there are some important reasons not to rule out universal errors completely:

- They remain the only viable means of bringing certain error conditions into the error-handling model, as discussed above. Of these, most run into various objections; the most important remaining use case is "escaping", where an unexpected implementation of an API that was not designed to throw finds itself needing to.
- Objective-C and C++ exceptions are a legitimate interoperation problem on any conceivable platform Swift targets. Swift must have some sort of long-term answer for them.

These reasons don't override the problems with universal errors. It is inherently dangerous to implicitly volunteer functions for unwinding from an arbitrary point. We don't want to promote this model. However, it is certainly possible to write code that handles universal errors correctly; and pragmatically, unwinding through most code will generally just work. Swift could support a secondary, untyped propagation mechanism using "zero-cost" exceptions. Code can be written carefully to minimize the extent of implicit unwinding, e.g. by catching universal errors immediately after calling an "escaping" API and rethrowing them with normal typed propagation.

However, this work is outside of the scope of Swift 2.0. We can comfortably make this decision because doing so doesn't lock us out of implementing it in the future:

- We do not currently support propagating exceptions through Swift functions, so changing `catch` to catch them as well would not be a major compatibility break.
- With some admitted awkwardness, external exceptions can be reflected into an `ErrorType` - like model automatically by the catch mechanism.
- In the meanwhile, developers who must handle an Objective-C exception can always do so by writing a stub in Objective-C to explicitly "bridge" the exception into an `NSError` out parameter. This isn't ideal, but it's acceptable.

Logic failures

The final category is logic failures, including out of bounds array accesses, forced unwrap of `nil` optionals, and other kinds of assertions. The programmer has made a mistake, and the failure should be handled by fixing the code, not by attempting to recover dynamically.

High-reliability systems may need some way to limp on even after an assertion failure. Tearing down the process can be viewed as a vector for a denial-of-service attack. However, an assertion failure might indicate that the process has been corrupted and is under attack, and limping on anyway may open the system up for other, more serious forms of security breach.

The correct handling of these error conditions is an open question and is not a focus of this proposal. Should we decide to make them recoverable, they will likely follow the same implementation mechanism as universal errors, if not necessarily the same language rules.

6.2 Analysis

Let's take a deeper look into the different dimensions of error-handling I laid out above.

6.2.1 Propagation methods

At a language level, there are two basic ways an error can be propagated from an error site to something handling it.

The first is that it can be done with the normal evaluation, data flow, and control flow processes of the language; let's call this **manual propagation**. Here's a good example of manual propagation using special return values in an imperative language, C:

```
struct object *read_object(void) {
  char buffer[1024];
  ssize_t numRead = read(0, buffer, sizeof(buffer));
  if (numRead < 0) return NULL;
  ...
}
```

Here's an example of manual propagation of an error value through out-parameters in another imperative language, Objective-C:

```
- (BOOL) readKeys: (NSArray<NSString*>**) strings error: (NSError**) err {
  while (1) {
    NSString *key;
    if ([self readKey: &key error: err]) {
      return TRUE;
    }
    ...
  }
  ...
}
```

Here's an example of manual propagation using an ADT in an impure functional language, SML; it's somewhat artificial because the SML library actually uses exceptions for this:

```
fun read_next_cmd () =
  case readline(stdin) of
    NONE => NONE
  | SOME line => if ...
```

All of these excerpts explicitly test for errors using the language's standard tools for data flow and then explicitly bypass the evaluation of the remainder of the function using the language's standard tools for control flow.

The other basic way to propagate errors is in some hidden, more intrinsic way not directly reflected in the ordinary control flow rules; let's call this **automatic propagation**. Here's a good example of automatic propagation using exceptions in an imperative language, Java:

```
String next = readline();
```

If `readline` encounters an error, it throws an exception; the language then terminates scopes until it dynamically reaches a `try` statement with a matching handler. Note the lack of any code at all implying that this might be happening.

The chief disadvantages of manual propagation are that it's tedious to write and requires a lot of repetitive boilerplate. This might sound superficial, but these are serious concerns. Tedium distracts programmers and makes them careless; careless error-handling code can be worse than useless. Repetitive boilerplate makes code less readable, hurting maintainability; it occupies the programmer's time, creating opportunity costs; it discourages handling errors *well* by making it burdensome to handle them *at all*; and it encourages shortcuts (such as extensive macro use) which may undermine other advantages and goals.

The chief disadvantage of automatic propagation is that it obscures the control flow of the code. I'll talk about this more in the next section.

Note that automatic propagation needn't be intrinsic in a language. The propagation is automatic if it doesn't correspond to visible constructs in the source. This effect can be duplicated as a library with any language facility that allows restructuring of code (e.g. with macros or other term-rewriting facilities) or overloading of basic syntax (e.g. Haskell mapping its `do` notation onto monads).

Note also that multiple propagation strategies may be "in play" for any particular program. For example, Java generally uses exceptions in its standard libraries, but some specific APIs might opt to instead return `null` on error for efficiency reasons. Objective-C provides a fairly full-featured exceptions model, but the standard APIs (with a few important exceptions) reserve them solely for unrecoverable errors, preferring manual propagation with `NSError` out-parameters instead. Haskell has a large number of core library functions which return `Maybe` values to indicate success or error, but it also offers at least two features resembling traditional, automatically-propagating exceptions (the `ErrorT` monad transform and exceptions in the `IO` monad).

So, while I'm going to talk as if languages implement a single propagation strategy, it should be understood that reality will always be more complex. It is literally impossible to prevent programmers from using manual propagation if they want to. Part of the proposal will discuss using multiple strategies at once.

6.2.2 Marked propagation

Closely related to the question of whether propagation is manual or automatic is whether it is marked or unmarked. Let's say that a language uses **marked propagation** if there is something *at the call site* which indicates that propagation is possible from that point.

To a certain extent, every language using manual propagation uses marked propagation, since the manual code to propagate the error approximately marks the call which generated the error. However, it is possible for the propagation logic to get separated from the call.

Marked propagation is at odds with one other major axis of language design: a language can't solely use marked propagation if it ever performs implicit operations that can produce errors. For example, a language that wanted out-of-memory conditions to be recoverable errors would have to consider everything that could allocate memory to a source of propagation; in a high-level language, that would include a large number of implicit operations. Such a language could not claim to use marked propagation.

The reason this all matters is because unmarked propagation is a pretty nasty thing to end up with; it makes it impossible to directly see what operations can produce errors, and therefore to directly understand the control flow of a

function. This leaves you with two options as a programmer:

- You can carefully consider the actual dynamic behavior of every function called by your function.
- You can carefully arrange your function so that there are no critical sections where an universal error can leave things in an unwanted state.

There are techniques for making the second more palatable. Chiefly, they involve never writing code that relies on normal control flow to maintain invariants and clean up after an operation; for example, always using constructors and destructors in C++ to manage resources. This is compulsory in C++ with exceptions enabled because of the possibility of implicit code that can throw, but it could theoretically be used in other languages. However, it still requires a fairly careful and rigorous style of programming.

It is possible to imagine a marked form of automatic propagation, where the propagation itself is implicit except that (local) origination points have to be explicitly marked. This is part of our proposal, and I'll discuss it below.

6.2.3 Typed propagation

The next major question is whether error propagation is explicitly tracked and limited by the language. That is, is there something explicitly *in the declaration of a function* that tells the programmer whether it can produce errors? Let's call this **typed propagation**.

Typed manual propagation

Whether propagation is typed is somewhat orthogonal to whether it's manual or marked, but there are some common patterns. The most dominant forms of manual propagation are all typed, since they pass the failure out of the callee, either as a direct result or in an out-parameter.

Here's another example of an out-parameter:

```
- (instancetype)initWithContentsOfURL:(NSURL *)url encoding:(NSStringEncoding)enc␣
↪error:(NSError **)error;
```

Out-parameters have some nice advantages. First, they're a reliable source of marking; even if the actual propagation gets separated from the call, you can always detect a call that can generate errors as long as its out-parameter has a recognizable name. Second, some of the boilerplate can be shared, because you can use the same variable as an out-parameter multiple times; unfortunately, you can't use this to "cheat" and only check for an error once unless you have some conventional guarantee that later calls won't spuriously overwrite the variable.

A common alternative in functional languages is to return an `Either` type:

```
trait Writer {
  fn write_line(&mut self, s: &str) -> Result<(), IoError>;
}
```

This forces the caller to deal with the error if they want to use the result. This works well unless the call does not really have a meaningful result (as `write_line` does not); then it depends on whether language makes it easy to accidentally ignore results. It also tends to create a lot of awkward nesting:

```
fn parse_two_ints_and_add_them() {
  match parse_int() {
    Err e => Err e
    Ok x => match parse_int() {
      Err e => Err e
      Ok y => Ok (x + y)
    }
```

```
  }
}
```

Here, another level of nesting is required for every sequential computation that can fail. Overloaded evaluation syntax like Haskell's `do` notation would help with both of these problems, but only by switching to a kind of automatic propagation.

Manual propagation can be untyped if it occurs through a side channel. For example, consider an object which set a flag on itself when it encountered an error instead of directly returning it; or consider a variant of POSIX which expected you to separately check `errno` to see if a particular system call failed.

Typed automatic propagation

Languages with typed automatic propagation vary along several dimensions.

The default typing rule

The most important question is whether you opt in to producing errors or opt out of them. That is, is a function with no specific annotation able to produce errors or not?

The normal resilience guideline is that you want the lazier option to preserve more flexibility for the implementation. A function that can produce errors is definitely more flexible, since it can do more things. Contrariwise, changing a function that doesn't produce errors into a function that does clearly changes its contract in ways that callers need to respond to. Unfortunately, this has some unpleasant consequences:

- Marked propagation would become very burdensome. Every call would involve an annotation, either on the function (to say it cannot generate errors) or on the call site (to mark propagation). Users would likely rebel against this much bookkeeping.
- Most functions cannot generate recoverable errors in the way I've defined that. That is, ignoring sources of universal errors, most functions can be reasonably expected to not be able to produce errors. But if that's not the default state, that means that most functions would need annotations; again, that's a lot of tedious bookkeeping. It's also a lot of clutter in the API.
- Suppose that you notice that a function incorrectly lacks an annotation. You go to fix it, but you can't without annotating all of the functions it calls, ad infinitum; like `const` correctness in C++, the net effect is to punish conscientious users for trying to improve their code.
- A model which pretends that every function is a source of errors is likely to be overwhelming for humans. Programmers ought to think rigorously about their code, but expecting them to also make rigorous decisions about all the code their code touches is probably too much. Worse, without marked propagation, the compiler can't really help the programmer concentrate on the known-possible sources of error.
- The compiler's analysis for code generation has to assume that all sorts of things can produce errors when they really can't. This creates a lot of implicit propagation paths that are actually 100% dead, which imposes a serious code-size penalty.

The alternative is to say that, by default, functions are not being able to generate errors. This agrees with what I'm assuming is the most common case. In terms of resilience, it means expecting users to think more carefully about which functions can generate errors before publishing an API; but this is similar to how Swift already asks them to think carefully about types. Also, they'll have at least added the right set of annotations for their initial implementation. So I believe this is a reasonable alternative.

Enforcement

The next question is how to enforce the typing rules that prohibit automatic propagation. Should it be done statically or dynamically? That is, if a function claims to not generate errors, and it calls a function that generates errors without handling the error, should that be a compiler error or a runtime assertion?

The only real benefit of dynamic enforcement is that it makes it easier to use a function that's incorrectly marked as being able to produce errors. That's a real problem if all functions are assumed to produce errors by default, because the mistake could just be an error of omission. If, however, functions are assumed to not produce errors, then someone must have taken deliberate action that introduced the mistake. I feel like the vastly improved static type-checking is worth some annoyance in this case.

Meanwhile, dynamic enforcement undermines most of the benefits of typed propagation so completely that it's hardly worth considering. The only benefit that really remains is that the annotation serves as meaningful documentation. So for the rest of this paper, assume that typed propagation is statically enforced unless otherwise indicated.

Specificity

The last question is how specific the typing should be: should a function be able to state the specific classes of errors it produces, or should the annotation be merely boolean?

Experience with Java suggests that getting over-specific with exception types doesn't really work out for the best. It's useful to be able to recognize specific classes of error, but libraries generally want to reserve flexibility about the exact kind of error they produce, and so many errors just end up falling into broad buckets. Different libraries end up with their own library-specific general error classes, and exceptions list end up just restating the library's own dependencies or wrapping the underlying errors in ways that loses criticial information.

Tradeoffs of typed propagation

Typed propagation has a number of advantages and disadvantages, mostly independent of whether the propagation is automatic.

The chief advantage is that it is safer. It forces programmers to do *something* to handle or propagate errors. That comes with some downsides, which I'll talk about, but I see this as a fairly core static safety guarantee. This is especially important in an environment where shuttling operations between threads is common, since it calls out the common situation where an error needs to propagate back to the originating thread somehow.

Even if we're settled on using typed propagation, we should be aware of the disadvantages and investigate ways to ameliorate them:

- Any sort of polymorphism gets more complicated, especially higher-order functions. Functions which cannot generate errors are in principle subtypes of functions which can. But:
  - Composability suffers. A higher-order function must decide whether its function argument is allowed to generate errors. If not, the function may be significantly limiting its usability, or at least making itself much more difficult to use with error-generating functions. If so, passing a function that does not may require a conversion (an awkward explicit one if using manual propagation), and the result of the call will likely claim to be able to generate errors when, in fact, it cannot. This can be solved with overloads, but that's a lot of boilerplate and redundancy, especially for calls that take multiple functions (like the function composition operator).
  - If an implicit conversion is allowed, it may need to introduce thunks. In some cases, these thunks would be inlineable — except that, actually, it is quite useful for code to be able to reverse this conversion and dynamically detect functions that cannot actually generate errors. For example, an algorithm might be able to avoid some unnecessary bookkeeping if it knows that its function argument never fails. This poses some representation challenges.

- It tends to promote decentralized error handling instead of letting errors propagate to a level that actually knows how to handle them.
  - Some programmers will always be tempted to incorrectly pepper their code with handlers that just swallow errors instead of correctly propagating them to the right place. This is often worse than useless; it would often be better if the error just propagated silently, because the result can be a system in an inconsistent state with no record of why. Good language and library facilities for propagating errors can help avoid this, especially when moving actions between threads.
  - There are many situations where errors are not actually possible because the programmer has carefully restricted the input. For example, matching :code:`/[0-9]{4}/` and then parsing the result as an integer. It needs to be convenient to do this in a context that cannot actually propagate errors, but the facility to do this needs to be carefully designed to discourage use for swallowing real errors. It might be sufficient if the facility does not actually swallow the error, but instead causes a real failure.
  - It is possible that the ease of higher-order programming in Swift might ameliorate many of these problems by letting users writing error-handling combinators. That is, in situations where a lazy Java programmer would find themselves writing a `try/catch` to swallow an exception, Swift would allow them to do something more correct with equal convenience.

One other minor advantage of marked, statically-enforced typed propagation: it's a boon for certain kinds of refactoring. Specifically, when a refactor makes an operation error-producing when it wasn't before, the absence of any those properties makes the refactor more treacherous and increases the odds of accidentally introducing a bug. If propagation is untyped, or the typing isn't statically enforced, the compiler isn't going to help you at all to find call sites which need to have error-checking code. Even with static typed propagation, if the propagation isn't marked specifically on the call site, the compiler won't warn you about calls made from contexts that can handle or implicitly propagate the error. But if all these things are true, the compiler will force you to look at all the existing call sites individually.

6.2.4 Error Types

There are many kinds of error. It's important to be able to recognize and respond to specific error causes programmatically. Swift should support easy pattern-matching for this.

But I've never really seen a point to coarser-grained categorization than that; for example, I'm not sure how you're supposed to react to an arbitrary, unknown IO error. And if there are useful error categories, they can probably be expressed with predicates instead of public subclasses. I think we start with a uni-type here and then challenge people to come up with reasons why they need anything more.

6.2.5 Implementation design

There are several different common strategies for implementing automatic error propagation. (Manual propagation doesn't need special attention in the implementation design.)

The implementation has two basic tasks common to most languages:

- Transferring control through scopes and functions to the appropriate handler for the error.
- Performing various semantic "clean up" tasks for the scopes that were abruptly terminated:
  - tearing down local variables, like C++ variables with destructors or strong/weak references in ARC-like languages;
  - releasing heap-allocated local variables, like captured variables in Swift or `__block` variables in ObjC;
  - executing scope-specific termination code, like C#'s `using` or Java/ObjC's `synchronized` statements; and
  - executing ad hoc cleanup blocks, like `finally` blocks in Java or `defer` actions in Swift.

Any particular call frame on the stack may have clean-ups or potential handlers or both; call these **interesting frames**.

Implicit manual propagation

One strategy is to implicitly produce code to check for errors and propagate them up the stack, imitating the code that the programmer would have written under manual propagation. For example, a function call could return an optional error in a special result register; the caller would check this register and, if appropriate, unwind the stack and return the same value.

Since propagation and unwinding are explicit in the generated code, this strategy hurts runtime performance along the non-error path more than the alternatives, and more code is required to do the explicitly unwinding. Branches involved in testing for errors are usually very easy to predict, so in hot code the direct performance impact is quite small, and the total impact is dominated by decreased code locality. Code can't always be hot, however.

These penalties are suffered even by uninteresting frames unless they appear in tail position. (An actual tail call isn't necessary; there just can't be anything that error propagation would skip.) And functions must do some added setup work before returning.

The upside is that the error path suffers no significant penalties beyond the code-size impact. The code-size impact can be significant, however: there is sometimes quite a lot of duplicate code needed for propagation along the error path.

This approach is therefore relatively even-handed about the error vs. the non-error path, although it requires some care in order to minimize code-size penalties for parallel error paths.

`setjmp` / `longmp`

Another strategy to is to dynamically maintain a thread-local stack of interesting frames. A function with an interesting frame must save information about its context in a buffer, like `setjmp` would, and then register that buffer with the runtime. If the scope returns normally, the buffer is accordingly unregistered. Starting propagation involves restoring the context for the top of the interesting-frames stack; the place where execution returns is called the "landing pad".

The advantage of this is that uninteresting frames don't need to do any work; context restoration just skips over them implicitly. This is faster both for the error and non-error paths. It is also possible to optimize this strategy so that (unlike `setjmp`) the test for an error is implicitly elided: use a slightly different address for the landing pad, so that propagating errors directly restore to that location.

The downside is that saving the context and registering the frame are not free:

- Registering the frame requires an access to thread-local state, which on our platforms means a function call because we're not willing to commit to anything more specific in the ABI.
- Jumping across arbitary frames invalidates the callee-save registers, so the registering frame must save them all eagerly. In calling conventions with many callee-save registers, this can be very expensive. However, this is only necessary when it's possible to resume normal execution from the landing pad: if the landing pad only has clean-ups and therefore always restarts propagation, those registers will have been saved and restored further out.
- Languages like C++, ObjC ARC, and Swift that have non-trivial clean-ups for many local variables tend to have many functions with interesting frames. This means both that the context-saving penalties are higher and that skipping uninteresting frames is a less valuable optimization.
- By the same token, functions in those languages often have many different clean-ups and/or handlers. For example, every new non-trivial variable might introduce a new clean-up. The function must either register a new landing pad for each clean-up (very expensive!) or track its current position in a way that a function-wide landing pad can figure out what scope it was in.

This approach can be hybridized with the unwinding approach below so that the interesting-frames stack abstractly describes the clean-ups in the frame instead of just restoring control somewhere and expecting the frame to figure it out. This can decrease the code size impact significantly for the common case of frames that just need to run some clean-ups before propagating the error further. It may even completely eliminate the need for a landing pad.

The ObjC/C++ exceptions system on iOS/ARM32 is kindof like that hybrid. Propagation and clean-up code is explicit in the function, but the registered context includes the "personality" information from the unwinding tables, which makes the decision whether to land at the landing pad at all. It also uses an optimized `setjmp` implementation that both avoids some context-saving and threads the branch as described above.

The ObjC exceptions system on pre-modern runtimes (e.g. on PPC and i386) uses the standard `setjmp` / `longjmp` functions. Every protected scope saves the context separately. This is all implemented in a very unsafe way that does not behave well in the presence of inlining.

Overall, this approach requires a lot of work in the non-error path of functions with interesting frames. Given that we expects functions with interesting frames to be very common in Swift, this is not an implementation approach we would consider in the abstract. However, it is the implementation approach for C++/ObjC exceptions on iOS/ARM32, so we need to at least interoperate with that.

Table-based unwinding

The final approach is side-table stack unwinding. This relies on being able to accurately figure out how to unwind through an arbitrary function on the system, given only the return address of a call it made and the stack pointer at that point.

On our system, this proceeds as follows. From an instruction pointer, the system unwinder looks up what linked image (executable or dylib) that function was loaded from. The linked image contains a special section, a table of unwind tables indexed by their offset within the linked image. Every non-leaf function should have an entry within this table, which provides sufficient information to unwind the function from an arbitrary call site.

This lookup process is quite expensive, especially since it has to repeat all the way up the stack until something actually handles the error. This makes the error path extremely slow. However, no explicit setup code is required along the non-error path, and so this approach is sometimes known as "zero-cost". That's something of a misnomer, because it does have several costs that can affect non-error performance. First, there's a small amount of load-time work required in order to resolve relocations to symbols used by the unwind tables. Second, the error path often requires code in the function, which can decrease code locality even if never executed. Third, the error path may use information that the non-error path would otherwise discard. And finally, the unwind tables themselves can be fairly large, although this is generally only a binary-size concern because they are carefully arranged to not need to be loaded off of disk unless an exception is thrown. But overall, "zero-cost" is close enough to correct.

To unwind a frame in this sense specifically means:

- Deciding whether the function handles the error.
- Cleaning up any interesting scopes that need to be broken down (either to get to the handler or to leave the function).
- If the function is being fully unwound, restoring any callee-save registers which the function might have changed.

This is language-specific, and so the table contains language-specific "personality" information, including a reference to a function to interpret it. This mechanism means that the unwinder is extremely flexible; not only can it support arbitrary languages, but it can support different language-specific unwinding table layouts for the same language.

Our current personality records for C++ and Objective-C contain just enough information to decide (1) whether an exception is handled by the frame and (2) if not, whether a clean-up is currently active. If either is true, it restores the context of a landing pad, which manually executes the clean-ups and enters the handler. This approach generally needs as much code in the function as implicit manual propagation would. However, we could optimize this for many

common cases by causing clean-ups to be called automatically by the interpretation function. That is, instead of a landing pad that looks notionally like this:

```
void *exception = ...;
SomeCXXType::~SomeCXXType(&foo);
objc_release(bar);
objc_release(baz);
_Unwind_Resume(exception);
```

The unwind table would have a record that looks notionally like this:

```
CALL_WITH_FRAME_ADDRESS(&SomeCXXType::~SomeCXXType, FRAME_OFFSET_OF(foo))
CALL_WITH_FRAME_VALUE(&objc_release, FRAME_OFFSET_OF(bar))
CALL_WITH_FRAME_VALUE(&objc_release, FRAME_OFFSET_OF(baz))
RESUME
```

And no code would actually be needed in the function. This would generally slow the error path down, because the interpretation function would have to interpret this mini-language, but it would move all the overhead out of the function and into the error table, where it would be more compact.

This is something that would also benefit C++ code.

6.2.6 Clean-up actions

Many languages have a built-in language tool for performing arbitrary clean-up when exiting a scope. This has two benefits. The first is that, even ignoring error propagation, it acts as a "scope guard" which ensures that the clean-up is done if the scope is exited early due to a `return`, `break`, or `continue` statement; otherwise, the programmer must carefully duplicate the clean-up in all such places. The second benefit is that it makes clean-up tractable in the face of automatic propagation, which creates so many implicit paths of control flow out of the scope that expecting the programmer to cover them all with explicit catch-and-rethrow blocks would be ridiculous.

There's an inherent tension in these language features between putting explicit clean-up code in the order it will be executed and putting it near the code it's cleaning up after. The former means that a top-to-bottom read of the code tells you what actions are being performed when; you don't have to worry about code implicitly intervening at the end of a scope. The latter makes it easy to verify at the point that a clean-up is needed that it will eventually happen; you don't need to scan down to the finally block and analyze what happens there.

`finally`

Java, Objective-C, and many other languages allow `try` statements to take a `finally` clause. The clause is an ordinary scope and may take arbitrary actions. The `finally` clause is performed when the preceding controlled scopes (including any `catch` clauses) are exited in any way: whether by falling off the end, directly branching or returning out, or throwing an exception.

`finally` is a rather awkward and verbose language feature. It separates the clean-up code from the operation that required it (although this has benefits, as discussed above). It adds a lot of braces and indentation, so edits that add new clean-ups can require a lot of code to be reformatted. When the same scope needs multiple clean-ups, the programmer must either put them in the same `finally` block (and thus create problems with clean-ups that might terminate the block early) or stack them up in separate blocks (which can really obscure the otherwise simple flow of code).

`defer`

Go provides a `defer` statement that just enqueues arbitrary code to be executed when the function exits. (More details of this appear in the survey of Go.)

This allows the defer action to be written near the code it "balances", allowing the reader to immediately see that the required clean-up will be done (but this has drawbacks, as discussed above). It's very compact, which is nice as most defer actions are short. It also allow multiple actions to pile up without adding awkward nesting. However, the function-exit semantics exacerbate the problem of searching for intervening clean-up actions, and they introduce semantic and performance problems with capturing the values of local variables.

Destructors

C++ allows types to define destructor functions, which are called when a function goes out of scope.

These are often used directly to clean up the ownership or other invariants on the type's value. For example, an owning-pointer type would free its value in its destructor, whereas a hash-table type would destroy its entries and free its buffer.

But they are also often used idiomatically just for the implicit destructor call, as a "scope guard" to ensure that something is done before the current operation completes. For an example close to my own heart, a compiler might use such a guard when parsing a local scope to ensure that new declarations are removed from the scope chains even if the function exits early due to a parse error. Unfortunately, since type destructors are C++'s only tool for this kind of clean-up, introducing ad-hoc clean-up code requires defining a new type every time.

The unique advantage of destructors compared to the options above is that destructors can be tied to temporary values created during the evaluation of an expression.

Generally, a clean-up action becomes necessary as the result of some "acquire" operation that occurs during an expression. `defer` and `finally` do not take effect until the next statement is reached, which creates an atomicity problem if code can be injected after the acquire. (For `finally`, this assumes that the acquire appears *before* the `try`. If instead the acquire appears *within* the `try`, there must be something which activates the clean-up, and that has the same atomicity problem.)

In contrast, if the acquire operation always creates a temporary with a destructor that does the clean-up, the language automatically guarantees this atomicity. This pattern is called "resource acquisition is initialization", or "RAII". Under RAII, all resources that require clean-up are carefully encapsulated within types with user-defined destructors, and the act of constructing an object of that type is exactly the act of acquiring the underlying resource.

Swift does not support user-defined destructors on value types, but it does support general RAII-like programming with class types and `deinit` methods, although (at the moment) the user must take special care to keep the object alive, as Swift does not normally guarantee the destruction order of objects.

RAII is very convenient when there's a definable "resource" and somebody's already wrapped its acquisition APIs to return appropriately-destructed objects. For other tasks, where a reasonable programmer might balk at defining a new type and possibly wrapping an API for a single purpose, a more *ad hoc* approach may be warranted.

6.3 Survey

6.3.1 C

C doesn't really have a consensus error-handling scheme. There's a built-in unwinding mechanism in `setjmp` and `longjmp`, but it's disliked for a host of good reasons. The dominant idiom in practice is for a function to encode failure using some unreasonable value for its result, like a null pointer or a negative count. The bad value(s) are often function-specific, and sometimes even argument- or state-specific.

On the caller side, it is unfortunately idiomatic (in some codebases) to have a common label for propagating failure at the end of a function (hence `goto fail`); this is because there's no inherent language support for ensuring that necessary cleanup is done before propagating out of a scope.

6.3.2 C++

C++ has exceptions. Exceptions can have almost any type in the language. Propagation typing is tied only to declarations; an indirect function pointer is generally assumed to be able to throw. Propagation typing used to allow functions to be specific about the kinds of exceptions they could throw (:code:`throws (std::exception)`), but this is deprecated in favor of just indicating whether a function can throw (:code:`noexcept(false)`).

C++ aspires to making out-of-memory a recoverable condition, and so allocation can throw. Therefore, it is essentially compulsory for the language to assume that constructors might throw. Since constructors are called pervasively and implicitly, it makes sense for the default rule to be that all functions can throw. Since many error sites are implicit, there is little choice but to use automatic unmarked propagation. The only reasonable way to clean up after a scope in such a world is to allow the compiler to do it automatically. C++ programmers therefore rely idiomatically on a pattern of shifting all scope cleanup into the destructors of local variables; sometimes such local values are created solely to set up a cleanup action in this way.

Different error sites occur with a different set of cleanups active, and there are a large number of such sites. In fact, prior to C++11, compilers were forced to assume by default that destructor calls could throw, so cleanups actually created more error sites. This all adds up to a significant code-size penalty for exceptions, even in projects which don't directly use them and which have no interest in recovering from out-of-memory conditions. For this reason, many C++ projects explicitly disable exceptions and rely on other error propagation mechanisms, on which there is no widespread consensus.

6.3.3 Objective C

Objective C has a first-class exceptions mechanism which is similar in feature set to Java's: `@throw` / `@try` / `@catch` / `@finally`. Exception values must be instances of an Objective-C class. The language does a small amount of implicit frame cleanup during exception propagation: locks held by `@synchronized` are released, stack copies of `__block` variables are torn down, and ARC `__weak` variables are destroyed. However, the language does not release object pointers held in local variables, even (by default) under ARC.

Objective C exceptions used to be implemented with `setjmp`, `longjmp`, and thread-local state managed by a runtime, but the only surviving platform we support which does that is i386, and all others now use a "zero-cost" implementation that interoperates with C++ exceptions.

Objective C exceptions are *mostly* only used for unrecoverable conditions, akin to what I called "failures" above. There are a few major exceptions to this rule, where APIs that do use exceptions to report errors.

Instead, Objective C mostly relies on manual propagation, predominantly using out-parameters of type `NSError**`. Whether the call failed is usually *not* indicated by whether a non-`nil` error was written into this parameter; calls are permitted both to succeed and write an error object into the parameter (which should be ignored) and to report an error without creating an actual error object. Instead, whether the call failed is reported in the formal return value. The most common convention is for a false `BOOL` result or null object result to mean an error, but ingenious programmers have come up with many other conventions, and there do exist APIs where a null object result is valid.

CF APIs, meanwhile, have their own magnificent set of somewhat inconsistent conventions.

Therefore, we can expect that incrementally improving CF / Objective C interoperation is going to be a long and remarkably painful process.

6.3.4 Java

Java has a first-class exceptions mechanism with unmarked automatic propagation: `throw` / `try` / `catch` / `finally`. Exception values must be instances of something inheriting from `Throwable`. Propagation is generally typed with static enforcement, with the default being that a call cannot throw exceptions *except* for subclasses of `Error` and `RuntimeException`. The original intent was that these classes would be used for catastrophic runtime errors (`Error`) and programming mistakes caught by the runtime (`RuntimeException`), both of which we would

classify as unrecoverable failures in our scheme; essentially, Java attempts to promote a fully statically-enforced model where truly catastrophic problems can still be handled when necessary. Unfortunately, these motivations don't seem to have been communicated very well to developers, and the result is kindof a mess.

Java allows methods to be very specific about the kinds of exception they throw. In my experience, exceptions tend to fall into two categories:

- There are some very specific exception kinds that callers know to look for and handle on specific operations. Generally these are obvious, predictable error conditions, like a host name not resolving, or like a string not being formatted correctly.
- There are also a lot of very vague, black-box exception kinds that can't really be usefully responded to. For example, if a method throws `IOException`, there's really nothing a caller can do except propagate it and abort the current operation.

So specific typing is useful if you can exhaustively handle a small number of specific failures. As soon as the exception list includes any kind of black box type, it might as well be a completely open set.

6.3.5 C#

C#'s model is almost exactly like Java's except that it is untyped: all methods are assumed to be able to throw. For this reason, it also has a simpler type hierarchy, where all exceptions just inherit from `Exception`.

The rest of the hierarchy doesn't really make any sense to me. Many things inherit directly from `Exception`, but many other things inherit from a subclass called `SystemException`. `SystemException` doesn't seem to be any sort of logical grouping: it includes all the runtime-assertion exceptions, but it also includes every exception that's thrown anywhere in the core library, including XML and IO exceptions.

C# also has a `using` statement, which is useful for binding something over a precise scope and then automatically disposing it on all paths. It's just built on top of `try` / `finally`.

6.3.6 Haskell

Haskell provides three different common error-propagation mechanisms.

The first is that, like many other functional languages, it supports manual propagation with a `Maybe` type. A function can return `None` to indicate that it couldn't produce a more useful result. This is the most common failure method for functions in the functional subset of the library.

The `IO` monad also provides true exceptions with unmarked automatic propagation. These exceptions can only be handled as an `IO` action, but are otherwise untyped: there is no way to indicate whether an `IO` action can or cannot throw. Exceptions can be thrown either as an `IO` action or as an ordinary lazy functional computation; in the latter case, the exception is only thrown if the computation is evaluated for some reason.

The `ErrorT` monad transform provides typed automatic propagation. In an amusing twist, since the only native computation of `ErrorT` is `throwError`, and the reason to write a computation specifically in `ErrorT` is if it's throwing, and every other computation must be explicitly lifted into the monad, `ErrorT` effectively uses marked propagation by omission, since everything that *can't* throw is explicitly marked with a `lift`:

```
prettyPrintShiftJIS :: ShiftJISString -> ErrorT TranscodeError IO ()
prettyPrintShiftJIS str = do
  lift $ putChar '"'     -- lift turns an IO computation into an ErrorT computation
  case transcodeShiftJISToUTF8 str of
    Left error -> throwError error
    Right value -> lift $ putEscapedString value
  lift $ putChar '"'
```

6.3.7 Rust

Rust distinguishes between *failures* and *panics*.

A panic is an assertion, designed for what I called logic failures; there's no way to recover from one, it just immediately crashes.

A failure is just when a function doesn't produce the value you might expect, which Rust encourages you to express with either `Option<T>` (for simple cases, like what I described as simple domain errors) or `Result<T>` (which is effectively the same, except carrying an error). In either case, it's typed manual propagation, although Rust does at least offer a standard macro which wraps the common pattern-match-and-return pattern for `Result<T>`.

The error type in Rust is a very simple protocol, much like this proposal suggests.

6.3.8 Go

Go uses an error result, conventionally returned as the final result of functions that can fail. The caller is expected to manually check whether this is nil; thus, Go uses typed manual propagation.

The error type in Go is an interface named `error`, with one method that returns a string description of the error.

Go has a `defer` statement:

```
defer foo(x, y)
```

The argument has to be a call (possibly a method call, possibly a call to a closure that you made specifically to immediately call). All the operands are evaluated immediately and captured in a deferred action. Immediately after the function exits (through whatever means), all the deferred actions are executed in LIFO order. Yes, this is tied to function exit, not scope exit, so you can have a dynamic number of deferred actions as a sort of implicit undo stack. Overall, it's a nice if somewhat quirky way to do ad-hoc cleanup actions.

It is also a key part of a second, funky kind of error propagation, which is essentially untyped automatic propagation. If you call `panic` — and certain builtin operations like array accesses behave like they do — it immediately unwinds the stack, running deferred actions as it goes. If a function's deferred action calls `recover`, the panic stops, the rest of the deferred actions for the function are called, and the function returns. A deferred action can write to the named results, allowing a function to turn a panic error into a normal, final-result error. It's conventional to not panic over API boundaries unless you really mean it; recoverable errors are supposed to be done with out-results.

6.3.9 Scripting languages

Scripting languages generally all use (untyped, obviously) automatic exception propagation, probably because it would be quite error-prone to do manual propagation in an untyped language. They pretty much all fit into the standard C++/Java/C# style of `throw` / `try` / `catch`. Ruby uses different keywords for it, though.

I feel like Python uses exceptions a lot more than most other scripting languages do, though.

6.4 Proposal

6.4.1 Automatic propagation

Swift should use automatic propagation of errors, rather than relying on the programmer to manually check for them and return out. It's just a lot less boilerplate for common error handling tasks. This introduces an implicit control flow problem, but we can ameliorate that with marked propagation; see below.

There's no compelling reason to deviate from the `throw` / `catch` legacy here. There are other options, like `raise` / `handle`. In theory, switching would somewhat dissociate Swift from the legacy of exceptions; people coming from other languages have a lot of assumptions about exceptions which don't necessarily apply to Swift. However, our error model is similar enough to the standard exception model that people are inevitably going to make the connection; there's no getting around the need to explain what we're trying to do. So using different keywords just seems petty.

Therefore, Swift should provide a `throw` expression. It requires an operand of type `Error` and formally yields an arbitrary type. Its dynamic behavior is to transfer control to the innermost enclosing `catch` clause which is satisfied by the operand. A quick example:

```
if timeElapsed() > timeThreshold { throw HomeworkError.Overworked }
```

A `catch` clause includes a pattern that matches an error. We want to repurpose the `try` keyword for marked propagation, which it seems to fit far better, so `catch` clauses will instead be attached to a generalized `do` statement:

```
do {
  ...

} catch HomeworkError.Overworked {
  // a conditionally-executed catch clause

} catch _ {
  // a catch-all clause
}
```

Swift should also provide some tools for doing manual propagation. We should have a standard Rust-like :code:`Result<T>` enum in the library, as well as a rich set of tools, e.g.:

- A function to evaluate an error-producing closure and capture the result as a :code:`Result<T>`.
- A function to unpack a :code:`Result<T>` by either returning its value or propagating the error in the current context.
- A futures library that traffics in :code:`Result<T>` when applicable.
- An overload of `dispatch_sync` which takes an error-producing closure and propagates an error in the current context.
- etc.

6.4.2 Typed propagation

Swift should use statically-enforced typed propagation. By default, functions should not be able to throw. A call to a function which can throw within a context that is not allowed to throw should be rejected by the compiler.

Function types should indicate whether the function throws; this needs to be tracked even for first-class function values. Functions which do not throw are subtypes of functions that throw.

This would be written with a `throws` clause on the function declaration or type:

```
// This function is not permitted to throw.
 func foo() -> Int {
   // Therefore this is a semantic error.
   return try stream.readInt()
 }

 // This function is permitted to throw.
 func bar() throws -> Int {
   return try stream.readInt()
```

```
}

// 'throws' is written before the arrow to give a sensible and
// consistent grammar for function types and implicit () result types.
func baz() throws {
  if let byte = try stream.getOOB() where byte == PROTO_RESET {
    reset()
  }
}

// 'throws' appears in a consistent position in function types.
func fred(callback: (UInt8) throws -> ()) throws {
   while true {
     let code = try stream.readByte()
     if code == OPER_CLOSE { return }
     try callback(code)
   }
}

// It only applies to the innermost function for curried functions;
// this function has type:
//   (Int) -> (Int) throws -> Int
func jerry(i: Int)(j: Int) throws -> Int {
  // It's not an error to use 'throws' on a function that can't throw.
  return i + j
}
```

The reason to use a keyword here is that it's much nicer for function declarations, which generally outnumber function types by at least an order of magnitude. A punctuation mark would be easily lost or mistaken amidst all the other punctuation in a function declaration, especially if the punctuation mark were something like `!` that can validly appear at the end of a parameter type. It makes sense for the keyword to appear close to the return type, as it's essentially a part of the result and a programmer should be able to see both parts in the same glance. The keyword appears before the arrow for the simple reason that the arrow is optional (along with the rest of the return type) in function and initializer declarations; having the keyword appear in slightly different places based on the presence of a return type would be silly and would making adding a non-void return type feel awkward. The keyword itself should be descriptive, and it's particularly nice for it to be a form of the verb used by the throwing expression, conjugated as if performed by the function itself. Thus, `throw` becomes `throws`; if we used `raise` instead, this would be `raises`, which I personally find unappealing for reasons I'm not sure I can put a name to.

It shouldn't be possible to overload functions solely based on whether the functions throw. That is, this is not legal:

```
func foo() { ... } // called in contexts that cannot throw
func foo() throws { ... } // called in contexts that can throw
```

It is valuable to be able to overload higher-order functions based on whether an argument function throws; it is easy to imagine algorithms that can be implemented more efficiently if they do not need to worry about exceptions. (We do not, however, particularly want to encourage a pattern of duplicating This is straightforward if the primary type-checking pass is able to reliably decide whether a function value can throw.

Typed propagation checking can generally be performed in a secondary pass over a type-checked function body: if a function is not permitted to throw, walk its body and verify that there are no `throw` expressions or calls to functions that can `throw`. If all throwing calls must be marked, this can be done prior to type-checking to decide syntactically whether a function can apparently throw; of course, the later pass is still necessary, but the ability to do this dramatically simplifies the implementation of the type-checker, as discussed below. Certain type-system features may need to be curtailed in order to make this implementation possible for schedule reasons. (It's important to understand that this is *not* the motivation for marked propagation. It's just a convenient consequence that marked propagation makes this implementation possible.)

Reliably deciding whether a function value can throw is easy for higher-order uses of declared functions. The problem, as usual, is anonymous functions. We don't want to require closures to be explicitly typed as throwing or non-throwing, but the fully-accurate inference algorithm requires a type-checked function body, and we can't always type-check an anonymous function independently of its enclosing context. Therefore, we will rely on being able to do a pass prior to type-checking to syntactically infer whether a closure throws, then making a second pass after type-checking to verify the correctness of that inference. This may break certain kinds of reasonable code, but the multi-pass approach should let us heuristically unbreak targeted cases.

Typed propagation has implications for all kinds of polymorphism:

Higher-order polymorphism

We should make it easy to write higher-order functions that behave polymorphically w.r.t. whether their arguments throw. This can be done in a fairly simple way: a function can declare that it throws if any of a set of named arguments do. As an example (using strawman syntax):

```
func map<T,U>(array: [T], fn: T throws -> U) throwsIf(fn) -> [U] {
  ...
}
```

There's no need for a more complex logical operator than disjunction. You can construct really strange code where a function throws only if one of its arguments doesn't, but it'd be contrived, and it's hard to imagine how they could be type-checked without a vastly more sophisticated approach. Similarly, you can construct situations where whether a function can throw is value-dependent on some other argument, like a "should I throw an exception" flag, but it's hard to imagine such cases being at all important to get right in the language. This schema is perfectly sufficient to express normal higher-order stuff.

In fact, while the strawman syntax above allows the function to be specific about exactly which argument functions cause the callee to throw, that's already overkill in the overwhelmingly likely case of a function that throws if any of its argument functions throw (and there's probably only one). So it would probably be better to just have a single `rethrows` annotation, with vague plans to allow it to be parameterized in the future if necessary.

This sort of propagation-checking would be a straightforward extension of the general propagation checker. The normal checker sees that a function isn't allowed to propagate out and looks for propagation points. The conditional checker sees that a function has a conditional propagation clause and looks for propagation points, assuming that the listed functions don't throw (including when looking at any conditional propagation clauses). The parameter would have to be a `let`.

We probably do need to get higher-order polymorphism right in the first release, because we will need it for the short-circuiting operators.

Generic polymorphism

It would be useful to be able to parameterize protocols, and protocol conformances, on whether the operations produce errors. Lacking this feature means that protocol authors must decide to either conservatively allow throwing conformances, and thus force all generic code using the protocol to deal with probably-spurious errors, or aggressively forbid them, and thus forbid conformances by types whose operations naturally throw.

There are several different ways we could approach this problem, but after some investigation I feel confident that they're workable. Unfortunately, they are clearly out-of-scope for the first release. For now, the standard library should provide protocols that cannot throw, even though this limits some potential conformances. (It's worth noting that such conformances generally aren't legal today, since they'd need to return an error result somehow.)

A future direction for both generic and higher-order polymorphism is to consider error propagation to be one of many possible effects in a general, user-extensible effect tracking system. This would allow the type system to check that

certain specific operations are only allowed in specific contexts: for example, that a blocking operation is only allowed in a blocking context.

Error type

The Swift standard library will provide `ErrorType`, a protocol with a very small interface (which is not described in this proposal). The standard pattern should be to define the conformance of an `enum` to the type:

```
enum HomeworkError : ErrorType {
  case Overworked
  case Impossible
  case EatenByCat(Cat)
  case StopStressingMeWithYourRules
}
```

The `enum` provides a namespace of errors, a list of possible errors within that namespace, and optional values to attach to each option.

For now, the list of errors in a domain will be fixed, but permitting future extension is just ordinary enum resilience, and the standard techniques for that will work fine in the future.

Note that this corresponds very cleanly to the `NSError` model of an error domain, an error code, and optional user data. We expect to import system error domains as enums that follow this approach and implement `ErrorType`. `NSError` and `CFError` themselves will also conform to `ErrorType`.

The physical representation (still being nailed down) will make it efficient to embed an `NSError` as an `ErrorType` and vice-versa. It should be possible to turn an arbitrary Swift `enum` that conforms to `ErrorType` into an `NSError` by using the qualified type name as the domain key, the enumerator as the error code, and turning the payload into user data.

It's acceptable to allocate memory whenever an error is needed, but our representation should not inhibit the optimizer from forwarding a `throw` directly to a `catch` and removing the intermediate error object.

6.4.3 Marked propagation

Swift should use marked propagation: there should be some lightweight bit of syntax decorating anything that is known be able to throw (other than a `throw` expression itself, of course).

Our proposed syntax is to repurpose `try` as something that can be wrapped around an arbitrary expression:

```
// This try applies to readBool().
if try stream.readBool() {

  // This try applies to both of these calls.
  let x = try stream.readInt() + stream.readInt()

  // This is a semantic error; it needs a try.
  var y = stream.readFloat()

  // This is okay; the try covers the entire statement.
  try y += stream.readFloat()
}
```

Developers can "scope" the `try` very tightly by writing it within parentheses or on a specific argument or list element:

```
// Semantic error: the try only covers the parenthesized expression.
let x = (try stream.readInt()) + stream.readInt()

// The try applies to the first array element.  Of course, the
// developer could cover the entire array by writing the try outside.
let array = [ try foo(), bar(), baz() ]
```

Some developers may wish to do this to make the specific throwing calls very clear. Other developers may be content with knowing that something within a statement can throw.

We also briefly considered the possibility of putting the marker into the call arguments clause, e.g.:

```
parser.readKeys(&strings, try)
```

This works as long as the only throwing calls are written syntactically as calls; this covers calls to free functions, methods, and initializers. However, it effectively requires Swift to forbid operators and property and subscript accessors from throwing, which may not be a reasonable limitation, especially for operators. It is also somewhat unnatural, and it forces users to mark every single call site instead of allowing them to mark everything within a statement at once.

Autoclosures pose a problem for marking. For the most part, we want to pretend that the expression of an autoclosure is being evaluated in the enclosing context; we don't want to have to mark both a call within the autoclosure and the call to the function taking the autoclosure! We should teach the type-checking pass to recognize this pattern: a call to a function that `throwsIf` an autoclosure argument does.

There's a similar problem with functions that are supposed to feel like statements. We want you to be able to write:

```
autoreleasepool {
  let string = parseString(try)
  ...
}
```

without marking the call to `autoreleasepool`, because this undermines the ability to write functions that feel like statements. However, there are other important differences between these trailing-closure uses and true built-in statements, such as the behavior of `return`, `break`, and `continue`. An attribute which marks the function as being statement-like would be a necessary step towards addressing both problems. Doing this reliably in closures would be challenging, however.

Asserting markers

Typed propagation is a hypothesis-checking mechanism and so suffers from the standard problem of false positives. (Basic soundness eliminates false negatives, of course: the compiler is supposed to force programmers to deal with *every* source of error.) In this case, a false positive means a situation where an API is declared to throw but an error is actually dynamically impossible.

For example, a function to load an image from a URL would usually be designed to produce an error if the image didn't exist, the connection failed, the file data was malformed, or any of a hundred other problems arose. The programmer should be expected to deal with that error in general. But a programmer might reasonably use the same API to load an image completely under their control, e.g. from their program's private resources. We shouldn't make it too syntactically inconvenient to "turn off" error-checking for such calls.

One important point is that we don't want to make it too easy to *ignore* errors. Ignored errors usually lead to a terrible debugging experience, even if the error is logged with a meaningful stack trace; the full context of the failure is lost and can be difficult to reproduce. Ignored errors also have a way of compounding, where an error that's "harmlessly" ignored at one layer of abstraction causes another error elsewhere; and of course the second error can be ignored, etc., but only by making the program harder and harder to understand and debug, leaving behind log files that are increasingly jammed with the detritus of a hundred ignored errors. And finally, ignoring errors creates a number of

type-safety and security problems by encouraging programs to blunder onwards with meaningless data and broken invariants.

Instead, we just want to make it (comparatively) easy to turn a static problem into a dynamic one, much as assertions and the ! operator do. Of course, this needs to be an explicit operation, because otherwise we would completely lose typed propagation; and it should be call-specific, so that the programmer has to make an informed decision about individual operations. But we already have an explicit, call-site-specific annotation: the `try` operator. So the obvious solution is to allow a variant of `try` that asserts that an error is not thrown out of its operand; and the obvious choice there within our existing design language is to use the universal "be careful, this is unsafe" marker by making the keyword `try!`.

It's reasonable to ask whether `try!` is actually *too* easy to write, given that this is, after all, an unsafe operation. One quick rejoinder is that it's no worse than the ordinary `!` operator in that sense. Like `!`, it's something that a cautious programmer might want to investigate closer, and you can easily imagine codebases that expect uses of it to always be explained in comments. But more importantly, just like `!` it's only *statically* unsafe, and it will reliably fail when the programmer is wrong. Therefore, while you can easily imagine (and demonstrate) uncautious programmers flailing around with it to appease the type-checker, that's not actually a tenable position for the overall program: eventually the programmer will have to learn how to use the feature, or else their program simply won't run.

Furthermore, while `try!` does somewhat undermine error-safety in the hands of a careless programmer, it's still better to promote this kind of unsafety than to implicitly promote the alternative. A careless programmer isn't going to write good error handling just because we don't give them this feature. Instead, they'll write out a `do/catch` block, and the natural pressure there will be to silently swallow the error — after all, that takes less boilerplate than asserting or logging.

In a future release, when we add support for universal errors, we'll need to reconsider the behavior of `try!`. One possibility is that `try!` should simply start propagating its operand as a universal error; this would allow emergency recovery. Alternatively, we may want `try!` to assert that even universal errors aren't thrown out of it; this would provide a more consistent language model between the two kinds of errors. But we don't need to think too hard about this yet.

6.4.4 Other syntax

Clean-up actions

Swift should provide a statement for cleaning up with an *ad hoc* action.

Overall, I think it is better to use a Go-style `defer` than a Java-style `try ... finally`. While this makes the exact order of execution more obscure, it does make it obvious that the clean-up *will* be executed without any further analysis, which is something that readers will usually be interested in.

Unlike Go, I think this should be tied to scope-exit, not to function-exit. This makes it very easy to know the set of `defer` actions that will be executed when a scope exits: it's all the `defer` statement in exactly that scope. In contrast, in Go you have to understand the dynamic history of the function's execution. This also eliminates some semantic and performance oddities relating to variable capture, since the `defer` action occurs with everything still in scope. One downside is that it's not as good for "transactional" idioms which push an undo action for everything they do, but that style has composition problems across function boundaries anyway.

I think `defer` is a reasonable name for this, although we might also consider `finally`. I'll use `defer` in the rest of this proposal.

`defer` may be followed by an arbitrary statement. The compiler should reject an action that might terminate early, whether by throwing or with `return`, `break`, or `continue`.

Examples:

```
if exists(filename) {
  let file = open(filename, O_READ)
  defer close(file)

  while let line = try file.readline() {
    ...
  }

  // close occurs here, at the end of the formal scope.
}
```

We should consider providing a convenient way to mark that a `defer` action should only be taken if an error is thrown. This is a convenient shorthand for controlling the action with a flag that's only set to true at the end of an operation. The flag approach is often more useful, since it allows the action to be taken for *any* early exit, e.g. a `return`, not just for error propagation.

`using`

Swift should consider providing a `using` statement which acquires a resource, holds it for a fixed period of time, optionally binds it to a name, and then releases it whenever the controlled statement exits.

`using` has many similarities to `defer`. It does not subsume `defer`, which is useful for many ad-hoc and tokenless clean-ups. But it is convenient for the common pattern of a type-directed clean-up.

We do not expect this feature to be necessary in the first release.

6.4.5 C and Objective-C Interoperation

It's of paramount importance that Swift's error model interact as cleanly with Objective-C APIs as we can make it.

In general, we want to try to import APIs that produce errors as throwing; if this fails, we'll import the API as an ordinary non-throwing function. This is a safe approach only under the assumption that importing the function as throwing will require significant changes to the call. That is, if a developer writes code assuming that an API will be imported as throwing, but in fact Swift fails to import the API that way, it's important that the code doesn't compile.

Fortunately, this is true for the common pattern of an error out-parameter: if Swift cannot import the function as throwing, it will leave the out-parameter in place, and the compiler will complain if the developer fails to pass an error argument. However, it is possible to imagine APIs where the "meat" of the error is returned in a different way; consider a POSIX API that simply sets `errno`. Great care would need to be taken when such an API is only partially imported as throwing.

Let's wade into the details.

Error types

`NSError` and `CFError` should implement the `ErrorType` protocol. It should be possible to turn an arbitrary Swift `enum` that conforms to `ErrorType` into an `NSError` by using the qualified type name as the domain key, the enumerator as the error code, and turning the payload into user data.

Recognizing system enums as error domains is a matter of annotation. Most likely, Swift will just special-case a few common domains in the first release.

Objective-C method error patterns

The most common error pattern in ObjC by far is for a method to have an autoreleased `NSError**` out-parameter. We don't currently propose automatically importing anything as `throws` when it lacks such a parameter.

If any APIs take an `NSError**` and *don't* intend for it to be an error out-parameter, they will almost certainly need it to be marked.

Detecting an error

Many of these methods have some sort of significant result which is used for testing whether an error occurred:

- The most common pattern is a `BOOL` result, where a false value means an error occurred. This seems unambiguous.

  Swift should import these methods as if they'd returned `Void`.

- Also common is a pointer result, where a `nil` result usually means an error occurred.

  I've been told that there are some exceptions to this rule, where a `nil` result is valid and the caller is apparently meant to check for a non-`nil` error. I haven't been able to find any such APIs in Cocoa, though; the claimed APIs I've been referred to do have nullable results, but returned via out-parameters with a *BOOL* formal result. So it seems to be a sound policy decision for Objective-C that `nil` results are errors by default. CF might be a different story, though.

  When a `nil` result implies that an error has occurred, Swift should import the method as returning a non-optional result.

- A few CF APIs return `void`. As far as I can tell, for all of these, the caller is expected to check for a non-`nil` error.

For other sentinel cases, we can consider adding a new clang attribute to indicate to the compiler what the sentinel is:

- There are several APIs returning `NSInteger` or `NSUInteger`. At least some of these return 0 on error, but that doesn't seem like a reasonable general assumption.
- `AVFoundation` provides a couple methods returning `AVKeyValueStatus`. These produce an error if the API returned `AVKeyValueStatusFailed`, which, interestingly enough, is not the zero value.

The clang attribute would specify how to test the return value for an error. For example:

```
+ (NSInteger)writePropertyList:(id)plist
                      toStream:(NSOutputStream *)stream
                        format:(NSPropertyListFormat)format
                       options:(NSPropertyListWriteOptions)opt
                         error:(out NSError **)error
  NS_ERROR_RESULT(0)

- (AVKeyValueStatus)statusOfValueForKey:(NSString *)key
                                  error:(NSError **)
  NS_ERROR_RESULT(AVKeyValueStatusFailed);
```

We should also provide a Clang attribute which specifies that the correct way to test for an error is to check the out-parameter. Both of these attributes could potentially be used by the static analyzer, not just Swift. (For example, they could try to detect an invalid error check.)

A constant value would be sufficient for the cases I've seen, but if the argument has to generalized to a simple expression, that's still feasible.

The error parameter

The obvious import rule for Objective-C methods with `NSError**` out-parameters is to simply mark them `throws` and remove the selector clause corresponding to the out-parameter. That is, a method like this one from `NSAttributedString`:

```
- (NSData *)dataFromRange:(NSRange)range
      documentAttributes:(NSDictionary *)dict
                   error:(NSError **)error;
```

would be imported as:

```
func dataFromRange(range: NSRange,
                   documentAttributes dict: NSDictionary) throws -> NSData
```

However, applying this rule haphazardly causes problems for Objective-C interoperation, because multiple methods can be imported the same way. The model is far more comprehensible to both compiler and programmer if the original Objective-C declaration can be unambiguously reconstructed from a Swift declaration.

There are two sources of this ambiguity:

- The error parameter could have appeared at an arbitrary position in the selector; that is, both `foo:bar:error:` and `foo:error:bar:` would appear as `foo:bar:` after import.
- The error parameter could have had an arbitrary selector chunk; that is, both `foo:error:` and `foo:withError:` would appear as `foo:` after import.

To allow reconstruction, then, we should only apply the rule when the error parameter is the last parameter and the corresponding selector is either `error:` or the first chunk. Empirically, this seems to do the right thing for all but two sets of APIs in the public API:

- The `ISyncSessionDriverDelegate` category on `NSObject` declares half-a-dozen methods like this:

  ```
  - (BOOL)sessionDriver:(ISyncSessionDriver *)sender
        didRegisterClientAndReturnError:(NSError **)outError;
  ```

  Fortunately, these delegate methods were all deprecated in Lion, and Swift currently doesn't even import deprecated methods.

- `NSFileCoordinator` has half a dozen methods where the `error:` clause is second-to-last, followed by a block argument. These methods are not deprecated as far as I know.

Of course, user code could also fail to follow this rule.

I think it's acceptable for Swift to just not import these methods as `throws`, leaving the original error parameter in place exactly as if they didn't follow an intelligible pattern in the header.

This translation rule would import methods like this one from `NSDocument`:

```
- (NSDocument *)duplicateAndReturnError:(NSError **)outError;
```

like so:

```
func duplicateAndReturnError() throws -> NSDocument
```

Leaving the `AndReturnError` bit around feels unfortunate to me, but I don't see what we could do without losing the ability to automatically reconstruct the Objective-C signature. This pattern is common but hardly universal; consider this method from `NSManagedObject`:

```
- (BOOL)validateForDelete:(NSError **)error;
```

This would be imported as:

```
func validateForDelete() throws
```

This seems like a really nice import.

CoreFoundation functions

CF APIs use `CFErrorRef` pretty reliably, but there are two problems.

First, we're not as confident about the memory management rules for the error object. Is it always returned at +1?

Second, I'm not as confident about how to detect that an error has occurred:

- There are a lot of functions that return `Boolean` or `bool`. It's likely that these functions consistently use the same convention as Objective-C: false means error.
- Similarly, there are many functions that return an object reference. Again, we'd need a policy on whether to treat `nil` results as errors.
- There are a handful of APIs that return a `CFIndex`, all with apparently the same rule that a zero value means an error. (These are serialization APIs, so writing nothing seems like a reasonable error.) But just like Objective-C, that does not seem like a reasonable default assumption.
- `ColorSyncProfile` has several related functions that return `float`! These are both apparently meant to be checked by testing whether the error result was filled in.

There are also some APIs that do not use `CFErrorRef`. For example, most of the `CVDisplayLink` APIs in CoreVideo returns their own `CVReturn` enumeration, many with more than one error value. Obviously, these will not be imported as throwing unless CoreVideo writes an overlay.

Other C APIs

In principle, we could import POSIX functions into Swift as throwing functions, filling in the error from `errno`. It's nearly impossible to imagine doing this with an automatic import rule, however; much more likely, we'd need to wrap them all in an overlay.

6.4.6 Implementation design

Error propagation for the kinds of explicit, typed errors that I've been focusing on should be handled by implicit manual propagation. It would be good to bias the implementation somewhat towards the non-error path, perhaps by moving error paths to the ends of functions and so on, and perhaps even by processing cleanups with an interpretive approach instead of directly inlining that code, but we should not bias so heavily as to seriously compromise performance. In other words, we should not use table-based unwinding.

Error propagation for universal errors should be handled by table-based unwinding. `catch` handlers can catch both, mapping unwind exceptions to `ErrorType` values as necessary. With a carefully-designed interpretation function aimed to solve the specific needs of Swift, we can avoid most of the code-size impact by shifting it to the unwind tables, which needn't ever be loaded in the common case.

CHAPTER 7

Generics in Swift

7.1 Motivation

Most types and functions in code are expressed in terms of a single, concrete set of sets. Generics generalize this notion by allowing one to express types and functions in terms of an abstraction over a (typically unbounded) set of types, allowing improved code reuse. A typical example of a generic type is a linked list of values, which can be used with any type of value. In C++, this might be expressed as:

```
template<typename T>
class List {
public:
  struct Node {
    T value;
    Node *next;
  };

  Node *first;
};
```

where List<Int>, List<String>, and List<DataRecord> are all distinct types that provide a linked list storing integers, strings, and DataRecords, respectively. Given such a data structure, one also needs to be able to implement generic functions that can operate on a list of any kind of elements, such as a simple, linear search algorithm:

```
template<typename T>
typename List<T>::Node *find(const List<T>&list, const T& value) {
  for (typename List<T>::Node *result = list.first; result; result = result->next)
    if (result->value == value)
      return result;

  return 0;
}
```

Generics are important for the construction of useful libraries, because they allow the library to adapt to application-specific data types without losing type safety. This is especially important for foundational libraries containing com-

mon data structures and algorithms, since these libraries are used across nearly every interesting application.

The alternatives to generics tend to lead to poor solutions:

- Object-oriented languages tend to use "top" types (id in Objective-C, java.lang.Object in pre-generics Java, etc.) for their containers and algorithms, which gives up static type safety. Pre- generics Java forced the user to introduce run-time-checked type casts when interacting with containers (which is overly verbose), while Objective-C relies on id's unsound implicit conversion behavior to eliminate the need for casts.
- Many languages bake common data structures (arrays, dictionaries, tables) into the language itself. This is unfortunate both because it significantly increases the size of the core language and because users then tend to use this limited set of data structures for *every* problem, even when another (not-baked-in) data structure would be better.

Swift is intended to be a small, expressive language with great support for building libraries. We'll need generics to be able to build those libraries well.

7.2 Goals

- Generics should enable the development of rich generic libraries that feel similar to first-class language features
- Generics should work on any type, whether it is a value type or some kind of object type
- Generic code should be almost as easy to write as non-generic code
- Generic code should be compiled such that it can be executed with any data type without requiring a separate "instantiation" step
- Generics should interoperate cleanly with run-time polymorphism
- Types should be able to retroactively modified to meet the requirements of a generic algorithm or data structure

As important as the goals of a feature are the explicit non-goals, which we don't want or don't need to support:

- Compile-time "metaprogramming" in any form
- Expression-template tricks a la Boost.Spirit, POOMA

7.3 Polymorphism

Polymorphism allows one to use different data types with a uniform interface. Overloading already allows a form of polymorphism (ad hoc polymorphism) in Swift. For example, given:

```
func +(x : Int, y : Int) -> Int { add... }
func +(x : String, y : String) -> String { concat... }
```

we can write the expression "x + y", which will work for both integers and strings.

However, we want the ability to express an algorithm or data structure independently of mentioning any data type. To do so, we need a way to express the essential interface that algorithm or data structure requires. For example, an accumulation algorithm would need to express that for any type T, one can write the expression "x + y" (where x and y are both of type T) and it will produce another T.

7.4 Protocols

Most languages that provide some form of polymorphism also have a way to describe abstract interfaces that cover a range of types: Java and C# interfaces, C++ abstract base classes, Objective-C protocols, Scala traits, Haskell type classes, C++ concepts (briefly), and many more. All allow one to describe functions or methods that are part of the interface, and provide some way to re-use or extend a previous interface by adding to it. We'll start with that core feature, and build onto it what we need.

In Swift, I suggest that we use the term protocol for this feature, because I expect the end result to be similar enough to Objective-C protocols that our users will benefit, and (more importantly) different enough from Java/C# interfaces and C++ abstract base classes that those terms will be harmful. The term trait comes with the wrong connotation for C++ programmers, and none of our users know Scala.

In its most basic form, a protocol is a collection of function signatures:

```
protocol Document {
  func title() -> String
}
```

Document describes types that have a title() operation that accepts no arguments and returns a String. Note that there is implicitly a 'self' type, which is the type that conforms to the protocol itself. This follows how most object-oriented languages describe interfaces, but deviates from Haskell type classes and C++ concepts, which require explicit type parameters for all of the types. We'll revisit this decision later.

7.5 Protocol Inheritance

Composition of protocols is important to help programmers organize and understand a large number of protocols and the data types that conform to those protocols. For example, we could extend our Document protocol to cover documents that support versioning:

```
protocol VersionedDocument : Document {
  func version() -> Int
}
```

Multiple inheritance is permitted, allowing us to form a directed acyclic graph of protocols:

```
protocol PersistentDocument : VersionedDocument, Serializable {
  func saveToFile(filename : path)
}
```

Any type that conforms to PersistentDocument also conforms to VersionedDocument, Document, and Serializable, which gives us substitutability.

7.6 Self Types

Protocols thus far do not give us an easy way to express simple binary operations. For example, let's try to write a Comparable protocol that could be used to search for a generic find() operation:

```
protocol Comparable {
  func isEqual(other : ???) -> bool
}
```

Our options for filling in ??? are currently very poor. We could use the syntax for saying "any type" or "any type that is comparable", as one must do most OO languages, including Java, C#, and Objective-C, but that's not expressing what we want: that the type of both of the arguments be the same. This is sometimes referred to as the binary method problem (http://www.cis.upenn.edu/~bcpierce/papers/binary.ps has a discussion of this problem, including the solution I'm proposing below).

Neither C++ concepts nor Haskell type classes have this particular problem, because they don't have the notion of an implicit 'Self' type. Rather, they explicitly parameterize everything. In C++ concepts:

```
concept Comparable<typename T> {
  bool T::isEqual(T);
}
```

Java and C# programmers work around this issue by parameterizing the interface, e.g. (in Java):

```
abstract class Comparable<THIS extends Comparable<THIS>> {
  public bool isEqual(THIS other);
}
```

and then a class X that wants to be Comparable will inherit from Comparable<X>. This is ugly and has a number of pitfalls; see http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6479372 .

Scala and Strongtalk have the notion of the 'Self' type, which effectively allows one to refer to the eventual type of 'self' (which we call 'self'). 'Self' (which we call 'Self' in Swift) allows us to express the Comparable protocol in a natural way:

```
protocol Comparable {
  func isEqual(other : Self) -> bool
}
```

By expressing Comparable in this way, we know that if we have two objects of type T where T conforms to Comparable, comparison between those two objects with isEqual is well-typed. However, if we have objects of different types T and U, we cannot compare those objects with isEqual even if both T and U are Comparable.

Self types are not without their costs, particularly in the case where Self is used as a parameter type of a class method that will be subclassed. Here, the parameter type ends up being (implicitly) covariant, which tightens up type-checking but may also force us into more dynamic type checks. We can explore this separately; within protocols, type-checking for Self is more direct.

7.7 Associated Types

In addition to Self, a protocol's operations often need to refer to types that are related to the type of 'Self', such as a type of data stored in a collection, or the node and edge types of a graph. For example, this would allow us to cleanly describe a protocol for collections:

```
protocol Collection {
  typealias Element
  func forEach(callback : (value : Element) -> void)
  func add(value : Element)
}
```

It is important here that a generic function that refers to a given type T, which is known to be a collection, can access the associated types corresponding to T. For example, one could implement an "accumulate" operation for an arbitrary Collection, but doing so requires us to specify some constraints on the Value type of the collection. We'll return to this later.

7.8 Operators, Properties, and Subscripting

As previously noted, protocols can contain both function requirements (which are in effect requirements for instance methods) and associated type requirements. Protocols can also contain operators, properties, and subscript operators:

```
protocol RandomAccessContainer : Collection {
  var length : Int
  func ==(lhs : Self, rhs : Self)
  subscript (i : Int) -> Element
}
```

Operator requirements can be satisfied by operator definitions, property requirements can be satisfied by either variables or properties, and subscript requirements can be satisfied by subscript operators.

7.9 Conforming to a Protocol

Thus far, we have not actually shown how a type can meet the requirements of a protocol. The most syntactically lightweight approach is to allow implicit conformance. This is essentially duck typing, where a type is assumed to conform to a protocol if it meets the syntactic requirements of the protocol. For example, given:

```
protocol Shape {
  func draw()
}
```

One could write a Circle struct such as:

```
struct Circle {
  var center : Point
  var radius : Int

  func draw() {
    // draw it
  }
}
```

Circle provides a draw() method with the same input and result types as required by the Shape protocol. Therefore, Circle conforms to Shape.

Implicit protocol conformance is convenient, because it requires no additional typing. However, it can run into some trouble when an entity that syntactically matches a protocol doesn't provide the required semantics. For example, Cowboys also know how to "draw!":

```
struct Cowboy {
  var gun : SixShooter

  func draw() {
    // draw!
  }
}
```

It is unlikely that Cowboy is meant to conform to Shape, but the method name and signatures match, so implicit conformance deduces that Cowboy conforms to Shape. Random collisions between types are fairly rare. However, when one is using protocol inheritance with fine- grained (semantic or mostly-semantic) differences between protocols in the hierarchy, they become more common. See http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1798.html for examples of this problem as it surfaced with C++ concepts. It is not clear at this time whether we want

implicit conformance in Swift: there's no existing code to worry about, and explicit conformance (described below) provides some benefits.

7.10 Explicit Protocol Conformance

Type authors often implement types that are intended to conform to a particular protocol. For example, if we want a linked-list type to conform to Collection, we can specify that it is by adding a protocol conformance annotation to the type:

```
struct EmployeeList : Collection { // EmployeeList is a collection
  typealias Element = T
  func forEach(callback : (value : Element) -> void) { /* Implement this */ }
  func add(value : Element) { /* Implement this */ }
}
```

This explicit protocol conformance declaration forces the compiler to check that EmployeeList actually does meet the requirements of the Collection protocol. If we were missing an operation (say, forEach) or had the wrong signature, the definition of 'EmployeeList' would be ill-formed. Therefore, explicit conformance provides both documentation for the user of EmployeeList and checking for the author and future maintainers of EmployeeList.

Any nominal type (such as an enum, struct, or class) can be specified to conform to one or more protocols in this manner. Additionally, a typealias can be specified to conform to one or more protocols, e.g.,:

```
typealias NSInteger : Numeric = Int
```

While not technically necessary due to retroactive modeling (below), this can be used to document and check that a particular type alias does in fact meet some basic, important requirements. Moreover, it falls out of the syntax that places requirements on associated types.

7.11 Retroactive Modeling

When using a set of libraries, it's fairly common that one library defines a protocol (and useful generic entities requiring that protocol) while another library provides a data type that provides similar functionality to that protocol, but under a different name. Retroactive modeling is the process by which the type is retrofitted (without changing the type) to meet the requirements of the protocol.

In Swift, we provide support for retroactive modeling by allowing extensions, e.g.,:

```
extension String : Collection {
  typealias Element = char
  func forEach(callback : (value : Element) -> void) { /* use existing String␣
↪routines to enumerate characters */ }
  func add(value : Element) { self += value /* append character */ }
}
```

Once an extension is defined, the extension now conforms to the Collection protocol, and can be used anywhere a Collection is expected.

7.12 Default Implementations

The functions declared within a protocol are requirements that any type must meet if it wants to conform to the protocol. There is a natural tension here, then, between larger protocols that make it easier to write generic algorithms, and

smaller protocols that make it easier to write conforming types. For example, should a Numeric protocol implement all operations, e.g.,:

```
protocol Numeric {
  func +(lhs : Self, rhs : Self) -> Self
  func -(lhs : Self, rhs : Self) -> Self
  func +(x : Self) -> Self
  func -(x : Self) -> Self
}
```

which would make it easy to write general numeric algorithms, but requires the author of some BigInt class to implement a lot of functionality, or should the numeric protocol implement just the core operations:

```
protocol Numeric {
  func +(lhs : Self, rhs : Self) -> Self
  func -(x : Self) -> Self
}
```

to make it easier to adopt the protocol (but harder to write numeric algorithms)? Both of the protocols express the same thing (semantically), because one can use the core operations (binary +, unary -) to implement the other algorithms. However, it's far easier to allow the protocol itself to provide default implementations:

```
protocol Numeric {
  func +(lhs : Self, rhs : Self) -> Self
  func -(lhs : Self, rhs : Self) -> Self { return lhs + -rhs }
  func +(x : Self) -> Self { return x }
  func -(x : Self) -> Self
}
```

This makes it easier both to implement generic algorithms (which can use the most natural syntax) and to make a new type conform to the protocol. For example, if we were to define only the core algorithms in our BigNum type:

```
struct BigNum : Numeric {
  func +(lhs : BigNum, rhs : BigNum) -> BigNum { ... }
  func -(x : BigNum) -> BigNum { ... }
}
```

the compiler will automatically synthesize the other operations needed for the protocol. Moreover, these operations will be available to uses of the BigNum class as if they had been written in the type itself (or in an extension of the type, if that feature is used), which means that protocol conformance actually makes it easier to define types that conform to protocols, rather than just providing additional checking.

7.13 Subtype Polymorphism

Subtype polymorphism is based on the notion of substitutability. If a type S is a subtype of a type T, then a value of type S can safely be used where a value of type T is expected. Object-oriented languages typically use subtype polymorphism, where the subtype relationship is based on inheritance: if the class Dog inherits from the class Animal, then Dog is a subtype of Animal. Subtype polymorphism is generally dynamic, in the sense that the substitution occurs at run-time, even if it is statically type-checked.

In Swift, we consider protocols to be types. A value of protocol type has an existential type, meaning that we don't know the concrete type until run-time (and even then it varies), but we know that the type conforms to the given protocol. Thus, a variable can be declared with type "Serializable", e.g.,:

```
var x : Serializable = // value of any Serializable type
x.serialize() // okay: serialize() is part of the Serializable protocol
```

Naturally, such polymorphism is dynamic, and will require boxing of value types to implement. We can now see how Self types interact with subtype polymorphism. For example, say we have two values of type Comparable, and we try to compare them:

```
var x : Comparable = ...
var y : Comparable = ...
if x.isEqual(y) { // well-typed?
}
```

Whether x.isEqual(y) is well-typed is not statically determinable, because the dynamic type of x may different from the dynamic type of y, even if they are both comparable (e.g., one is an Int and the other a String). It can be implemented by the compiler as a dynamic type check, with some general failure mode (aborting, throwing an exception, etc.) if the dynamic type check fails.

To express types that meet the requirements of several protocols, one can just create a new protocol aggregating those protocols:

```
protocol SerializableDocument : Document, Serializable { }
var doc : SerializableDocument
print(doc.title()) // okay: title() is part of the Document protocol, so we can call␣
↪it
doc.serialize(stout) // okay: serialize() is part of the Serializable protocol
```

However, this only makes sense when the resulting protocol is a useful abstraction. A SerializableDocument may or may not be a useful abstraction. When it is not useful, one can instead use protocol<> types to compose different protocols, e.g.,:

```
var doc : protocol<Document, Serializable>
```

Here, doc has an existential type that is known to conform to both the Document and Serializable protocols. This gives rise to a natural "top" type, such that every type in the language is a subtype of "top". Java has java.lang.Object, C# has object, Objective-C has "id" (although "id" is weird, because it is also convertible to everything; it's best not to use it as a model). In Swift, the "top" type is simply an empty protocol composition:

```
typealias Any = protocol<>

var value : Any = 17 // an any can hold an integer
value = "hello" // or a String
value = (42, "hello", Red) // or anything else
```

7.14 Bounded Parametric Polymorphism

Parametric polymorphism is based on the idea of providing type parameters for a generic function or type. When using that function or type, one substitutes concrete types for the type parameters. Strictly speaking, parametric polymorphism allows *any* type to be substituted for a type parameter, but it's useless in practice because that means that generic functions or types cannot do anything to the type parameters: they must instead rely on first-class functions passed into the generic function or type to perform any meaningful work.

Far more useful (and prevalent) is bounded parametric polymorphism, which allows the generic function or type to specify constraints (bounds) on the type parameters. By specifying these bounds, it becomes far easier to write and use these generic functions and types. Haskell type classes, Java and C# generics, C++ concepts, and many other language features support bounded parametric polymorphism.

Protocols provide a natural way to express the constraints of a generic function in Swift. For example, one could define a generic linked list as:

```
struct ListNode<T> {
  var Value : T
  enum NextNode { case Node : ListNode<T>, End }
  var Next : NextNode
}

struct List<T > {
  var First : ListNode<T>::NextNode
}
```

This list works on any type T. One could then add a generic function that inserts at the beginning of the list:

```
func insertAtBeginning<T>(list : List<T>, value : T) {
  list.First = ListNode<T>(value, list.First)
}
```

7.15 Expressing Constraints

Within the type parameter list of a generic type or function (e.g., the <T> in ListNode<T>), the 'T' introduces a new type parameter and the (optional) ": type" names a protocol (or protocol composition) to which 'T' must conform. Within the body of the generic type or function, any of the functions or types described by the constraints are available. For example, let's implement a find() operation on lists:

```
func find<T : Comparable>(list : List<T>, value : T) -> Int {
  var index = 0
  var current
  for (current = list.First; current is Node; current = current.Next) {
    if current.Value.isEqual(value) { // okay: T is Comparable
      return index
    }
    index = index + 1
  }
  return -1
}
```

In addition to providing constraints on the type parameters, we also need to be able to constrain associated types. To do so, we introduce the notion of a "where" clause, which follows the signature of the generic type or function. For example, let's generalize our find algorithm to work on any ordered collection:

```
protocol OrderedCollection : Collection {
  func size() -> Int
  func getAt(index : Int) -> Element // Element is an associated type
}

func find<C : OrderedCollection where C.Element : Comparable>(
       collection : C, value : C.Element) -> Int
{
  for index in 0...collection.size() {
    if (collection.getAt(index) == value) { // okay: we know that C.Element is␣
↪Comparable
      return index
    }
```

```
  }
  return -1
}
```

The where clause is actually the more general way of expressing constraints, and the constraints expressed in the angle brackets (e.g., <C : OrderedCollection>) are just sugar for a where clause. For example, the above find() signature is equivalent to:

```
func find<C where C : OrderedCollection, C.Element : Comparable>(
       collection : C, value : C.Element)-> Int
```

Note that find<C> is shorthand for (and equivalent to) find<C : Any>, since every type conforms to the Any protocol composition.

There are two other important kinds of constraints that need to be expressible. Before we get to those, consider a simple "Enumerator" protocol that lets us describe an iteration of values of some given value type:

```
protocol Enumerator {
  typealias Element
  func isEmpty() -> Bool
  func next() -> Element
}
```

Now, we want to express the notion of an enumerable collection, which provides a iteration, which we do by adding requirements into the protocol:

```
protocol EnumerableCollection : Collection {
  typealias EnumeratorType : Enumerator
  where EnumeratorType.Element == Element
  func getEnumeratorType() -> EnumeratorType
}
```

Here, we are specifying constraints on an associated type (EnumeratorType must conform to the Enumerator protocol), by adding a conformance clause (: Enumerator) to the associated type definition. We also use a separate where clause to require that the type of values produced by querying the enumerator is the same as the type of values stored in the container. This is important, for example, for use with the Comparable protocol (and any protocol using Self types), because it maintains type identity within the generic function or type.

7.16 Constraint Inference

Generic types often constrain their type parameters. For example, a SortedDictionary, which provides dictionary functionality using some kind of balanced binary tree (as in C++'s std::map), would require that its key type be Comparable:

```
class SortedDictionary<Key : Comparable, Value> {
  // ...
}
```

Naturally, one any generic operation on a SortedDictionary<K,V> would also require that K be Comparable, e.g.,:

```
func forEachKey<Key : Comparable, Value>(c : SortedDictionary<Key, Value>,
                                         f : (Key) -> Void) { /* ... */ }
```

However, explicitly requiring that Key conform to Comparable is redundant: one could not provide an argument for 'c' without the Key type of the SortedDictionary conforming to Comparable, because the SortedDictionary type

itself could not be formed. Constraint inference infers these additional constraints within a generic function from the parameter and return types of the function, simplifying the specification of forEachKey:

```
func forEachKey<Key, Value>(c : SortedDictionary<Key, Value>,
                            f : (Key) -> Void) { /* ... */ }
```

7.17 Type Parameter Deduction

As noted above, type arguments will be deduced from the call arguments to a generic function:

```
var values : list<Int>
insertAtBeginning(values, 17) // deduces T = Int
```

Since Swift already has top-down type inference (as well as the C++-like bottom-up inference), we can also deduce type arguments from the result type:

```
func cast<T, U>(value : T) -> U { ... }
var x : Any
var y : Int = cast(x) // deduces T = Any, U = Int
```

We require that all type parameters for a generic function be deducible. We introduce this restriction so that we can avoid introducing a syntax for explicitly specifying type arguments to a generic function, e.g.,:

```
var y : Int = cast<Int>(x) // not permitted: < is the less-than operator
```

This syntax is horribly ambiguous in C++, and with good type argument deduction, should not be necessary in Swift.

7.18 Implementation Model

Because generics are constrained, a well-typed generic function or type can be translated into object code that uses dynamic dispatch to perform each of its operations on type parameters. This is in stark contrast to the instantiation model of C++ templates, where each new set of template arguments requires the generic function or type to be compiled again. This model is important for scalability of builds, so that the time to perform type-checking and code generation scales with the amount of code written rather than the amount of code instantiated. Moreover, it can lead to smaller binaries and a more flexible language (generic functions can be "virtual").

The translation model is fairly simple. Consider the generic find() we implemented for lists, above:

```
func find<T : Comparable>(list : List<T>, value : T) -> Int {
  var index = 0
  var current = list.First
  while current is ListNode<T> { // now I'm just making stuff up
    if current.value.isEqual(value) { // okay: T is Comparable
      return index
    }
    current = current.Next
    index = index + 1
  }
  return -1
}
```

to translate this into executable code, we form a vtable for each of the constraints on the generic function. In this case, we'll have a vtable for Comparable T. Every operation within the body of this generic function type-checks to either

an operation on some concrete type (e.g., the operations on Int), to an operation within a protocol (which requires indirection through the corresponding vtable), or to an operation on a generic type definition, all of which can be emitted as object code.

7.19 Specialization

This implementation model lends itself to optimization when we know the specific argument types that will be used when invoking the generic function. In this case, some or all of the vtables provided for the constraints will effectively be constants. By specializing the generic function (at compile-time, link-time, or (if we have a JIT) run-time) for these types, we can eliminate the cost of the virtual dispatch, inline calls when appropriate, and eliminate the overhead of the generic system. Such optimizations can be performed based on heuristics, user direction, or profile-guided optimization.

7.20 Existential Types and Generics

Both existential types and generics depend on dynamic dispatching based on protocols. A value of an existential type (say, Comparable) is a pair (value, vtable). 'value' stores the current value either directly (if it fits in the 3 words allocated to the value) or as a pointer to the boxed representation (if the actual representation is larger than 3 words). By itself, this value cannot be interpreted, because it's type is not known statically, and may change due to assignment. The vtable provides the means to manipulate the value, because it provides a mapping between the protocols to which the existential type conforms (which is known statically) to the functions that implementation that functionality for the type of the value. The value, therefore, can only be safely manipulated through the functions in this vtable.

A value of some generic type T uses a similar implementation model. However, the (value, vtable) pair is split apart: values of type T contain only the value part (the 3 words of data), while the vtable is maintained as a separate value that can be shared among all T's within that generic function.

7.21 Overloading

Generic functions can be overloaded based entirely on constraints. For example, consider a binary search algorithm:

```
func binarySearch<
   C : EnumerableCollection where C.Element : Comparable
>(collection : C, value : C.Element)
  -> C.EnumeratorType
{
  // We can perform log(N) comparisons, but EnumerableCollection
  // only supports linear walks, so this is linear time
}

protocol RandomAccessEnumerator : Enumerator {
  // splits a range in half, returning both halves
  func split() -> (Enumerator, Enumerator)
}

func binarySearch<
   C : EnumerableCollection
    where C.Element : Comparable,
              C.EnumeratorType: RandomAccessEnumerator
>(collection : C, value : C.Element)
  -> C.EnumeratorType
```

```
{
  // We can perform log(N) comparisons and log(N) range splits,
  // so this is logarithmic time
}
```

If binarySearch is called with a sequence whose range type conforms to RandomAccessEnumerator, both of the generic functions match. However, the second function is more specialized, because its constraints are a superset of the constraints of the first function. In such a case, overloading should pick the more specialized function.

There is a question as to when this overloading occurs. For example, binarySearch might be called as a subroutine of another generic function with minimal requirements:

```
func doSomethingWithSearch<
  C : EnumerableCollection where C.Element : Ordered
>(
  collection : C, value : C.Element
) -> C.EnumeratorType
{
  binarySearch(collection, value)
}
```

At the time when the generic definition of doSomethingWithSearch is type-checked, only the first binarySearch() function applies, since we don't know that C.EnumeratorType conforms to RandomAccessEnumerator. However, when doSomethingWithSearch is actually invoked, C.EnumeratorType might conform to the RandomAccessEnumerator, in which case we'd be better off picking the second binarySearch. This amounts to run-time overload resolution, which may be desirable, but also has downsides, such as the potential for run-time failures due to ambiguities and the cost of performing such an expensive operation at these call sites. Of course, that cost could be mitigated in hot generic functions via the specialization mentioned above.

Our current proposal for this is to decide statically which function is called (based on similar partial-ordering rules as used in C++), and avoid run-time overload resolution. If this proves onerous, we can revisit the decision later.

7.22 Parsing Issues

The use of angle brackets to supply arguments to a generic type, while familiar to C++/C#/Java programmers, cause some parsing problems. The problem stems from the fact that '<', '>', and '>>' (the latter of which will show up in generic types such as Array<Array<Int>>) match the 'operator' terminal in the grammar, and we wish to continue using this as operators.

When we're in the type grammar, this is a minor inconvenience for the parser, because code like this:

```
var x : Array<Int>
```

will essentially parse the type as:

```
identifier operator Int operator
```

and verify that the operators are '<' and '>', respectively. Cases involving <> are more interesting, because the type of:

```
var y : Array<Array<Int>>
```

is effectively parsed as:

```
identifier operator identifier operator identifier operator operator
```

by splitting the '>>' operator token into two '>' operator tokens.

However, this is manageable, and is already implemented for protocol composition (protocol<>). The larger problem occurs at expression context, where the parser cannot disambiguate the tokens:

```
Matrix<Double>(10, 10)
```

i.e.,:

```
identifier operator identifier operator unspaced_lparen integer- literal comma␣
↪integer-literal rparen
```

which can be interpreted as either:

```
(greater_than
  (less_than
    (declref Matrix)
    (declref Double)
  (tuple
    (integer_literal 10)
    (integer_literal 10)))
```

or:

```
(constructor Matrix<Double>
  (tuple
    (integer_literal 10)
    (integer_literal 10)))
```

Both Java and C# have this ambiguity. C# resolves the ambiguity by looking at the token after the closing '>' to decide which way to go; Java seems to do the same. We have a few options:

1. Follow C# and Java and implement the infinite lookahead needed to make this work. Note that we have true ambiguities, because one could make either of the above parse trees well-formed.
2. Introduce some kind of special rule for '<' like we have for '(', such as: an identifier followed by an unspaced '<' is a type, while an identifier followed by spacing and then '<' is an expression, or
3. Pick some syntax other than angle brackets, which is not ambiguous. Note that neither '(' nor '[' work, because they too have expression forms.
4. Disambiguate between the two parses semantically.

We're going to try a variant of #1, using a variation of the disambiguation rule used in C#. Essentially, when we see:

```
identifier <
```

we look ahead, trying to parse a type parameter list, until parsing the type parameter list fails or we find a closing '>'. We then look ahead an additional token to see if the closing '>' is followed by a '(', '.', or closing bracketing token (since types are most commonly followed by a constructor call or static member access). If parsing the type parameter list succeeds, and the closing angle bracket is followed by a '(', '.', or closing bracket token, then the '<...>' sequence is parsed as a generic parameter list; otherwise, the '<' is parsed as an operator.

CHAPTER 8

Logical Objects

Universal across programming languages, we have this concept of a value, which is just some amount of fixed data. A value might be the int 5, or a pair of the bool true and the int -20, or an NSRect with the component values (0, 0, 400, 600), or whatever.

In imperative languages we have this concept of an object. It's an unfortunately overloaded term; here I'm using it like the standards do, which is to say that it's a thing that holds a value, but which can be altered at any time to hold a different value. It's tempting to use the word variable instead, and a variable is indeed an object, but "variable" implies all this extra stuff, like being its own independent, self-contained thing, whereas we want a word that also covers fields and array elements and what-have-you. So let's just suck it up and go with "object".

You might also call these "r-value" and "l-value". These have their own connotations that I don't want to get into. Stick with "value" and "object".

In C and C++, every object is physical. It's actually a place in memory somewhere. It's not necessarily easily addressable (because of bitfields), and its lifetime may be tightly constrained (because of temporaries or deallocation), but it's always memory.

In Objective-C, properties and subscripting add an idea of a logical object. The only way you can manipulate it is by calling a function (with unrestricted extra arguments) to either fetch the current value or update it with a new value. The logical object doesn't promise to behave like a physical object, either: for example, you can set it, then immediately get it, and the result might not match the value you set.

Swift has logical objects as well. We have them in a few new places (global objects can be logical), and sometimes we treat objects that are really physical as logical (because resilience prevents us from assuming physicality), and we're considering making some restrictions on how different a logical object can be from a physical object (although set-and-get would still be opaque), but otherwise they're pretty much just like they are in Objective-C.

That said, they do interact with some other language features in interesting ways.

For example, methods on value types have a this parameter. Usually parameters are values, but this is actually an object: if I call a method on an object, and the method modifies the value of this, I expect it to modify the object I called the method on. This is the high-level perspective of what [inout] really means: that what we're really passing as a parameter is an object, not a value. With one exception, everything that follows applies to any sort of [inout] parameter, not just this on value types. More on that exception later.

How do you actually pass an object, though, given that even physical objects might not be addressable, but especially given that an object might be logical?

Well, we can always treat a physical object like a logical object. It's possible to come up with ways to implement passing a logical object (pass a pointer to a struct, the first value of which is a getter, the second value of which is a setter, and the rest of which is opaque to the callee; the struct must be passed to the getter and setter functions). Unfortunately, the performance implications would be terrible: accessing multiple fields would involve multiple calls to the getter, each returning tons of extra information. And getter and setter calls might be very expensive.

We could pass a hybrid format, using direct accesses when possible and a getter/setter when not. Unfortunately, that's a lot of code bloat in every single method implementation.

Or we can always pass a physical, addressable object. This avoids penalizing the fast case where the object is really physical, which is great. For the case where the object is logical, we just have to make it physical somehow. That means materialization: calling the getter, storing the result into temporary memory, passing the temporary, and then calling the setter with the new value in the temporary when the method call is done. This last step is called writeback.

(About that one exception to this all applying equally to [inout]: in addition to all this stuff about calling methods on different kinds of object, we also want to support calling a method on a value. This is also implemented with a form of materialization, which looks just like the logical-object kind except without writeback, because there's nothing to write back to. This is a special rule that only applies to passing this, because we assume that most types will have lots of useful methods that don't involve writing to this, whereas we assume that a function with an explicit [inout] parameter is almost certain to want to write to it.)

CHAPTER 9

Object Initialization

Warning: This document is incomplete and not up-to-date; it currently describes the initialization model from Swift 1.0.

Contents

- *Object Initialization*
  - *Introduction*
  - *Initializers*
    - *Designated Initializers*
    - *Convenience Initializers*
    - *Initializer Inheritance*
    - *Synthesized Initializers*
    - *Required Initializers*
    - *Initializers in Protocols*
    - *De-initializers*
    - *Methods Returning `Self`*
  - *Memory Safety*
    - *Three-Phase Initialization*
    - *Initializer Inheritance Model*
  - *Objective-C Interoperability*
    - *Initializers and Init Methods*

* *Designated and Convenience Initializers*
* *Allocation and Deallocation*
* *Dynamic Subclassing*

9.1 Introduction

Object initialization is the process by which a new object is allocated, its stored properties initialized, and any additional setup tasks are performed, including allowing its superclass's to perform their own initialization. *Object teardown* is the reverse process, performing teardown tasks, destroying stored properties, and eventually deallocating the object.

9.2 Initializers

An initializer is responsible for the initialization of an object. Initializers are introduced with the `init` keyword. For example:

```
class A {
  var i: Int
  var s: String

  init int(i: Int) string(s: String) {
    self.i = i
    self.s = s
    completeInit()
  }

  func completeInit() { /* ... */ }
}
```

Here, the class `A` has an initializer that accepts an `Int` and a `String`, and uses them to initialize its two stored properties, then calls another method to perform other initialization tasks. The initializer can be invoked by constructing a new `A` object:

```
var a = A(int: 17, string: "Seventeen")
```

The allocation of the new `A` object is implicit in the construction syntax, and cannot be separated from the call to the initializer.

Within an initializer, all of the stored properties must be initialized (via an assignment) before `self` can be used in any way. For example, the following would produce a compiler error:

```
init int(i: Int) string(s: String) {
  completeInit() // error: variable 'self.i' used before being initialized
  self.i = i
  self.s = s
}
```

A stored property with an initial value declared within the class is considered to be initialized at the beginning of the initializer. For example, the following is a valid initializer:

```
class A2 {
  var i: Int = 17
  var s: String = "Seventeen"

  init int(i: Int) string(s: String) {
    // okay: i and s are both initialized in the class
    completeInit()
  }

  func completeInit() { /* ... */ }
}
```

After all stored properties have been initialized, one is free to use `self` in any manner.

9.2.1 Designated Initializers

There are two kinds of initializers in Swift: designated initializers and convenience initializers. A *designated initializer* is responsible for the primary initialization of an object, including the initialization of any stored properties, *chaining* to one of its superclass's designated initializers via a `super.init` call (if there is a superclass), and performing any other initialization tasks, in that order. For example, consider a subclass `B` of `A`:

```
class B : A {
  var d: Double

  init int(i: Int) string(s: String) {
    self.d = Double(i)            // initialize stored properties
    super.init(int: i, string: s) // chain to superclass
    completeInitForB()            // perform other tasks
  }

  func completeInitForB() { /* ... */ }
}
```

Consider the following construction of an object of type `B`:

```
var b = B(int: 17, string: "Seventeen")
```

Note

Swift differs from many other languages in that it requires one to initialize stored properties *before* chaining to the superclass initializer. This is part of Swift's memory safety guarantee, and is discussed further in the section on *Three-Phase Initialization*.

Initialization proceeds in several steps:

1. An object of type `B` is allocated by the runtime.
2. `B`'s initializer initializes the stored property `d` to `17.0`.
3. `B`'s initializer chains to `A`'s initializer.
4. `A`'s initializer initialize's the stored properties `i` and `s`'.
5. `A`'s initializer calls `completeInit()`, then returns.
6. `B`'s initializer calls `completeInitForB()`, then returns.

A class generally has a small number of designated initializers, which act as funnel points through which the object will be initialized. All of the designated initializers for a class must be written within the class definition itself, rather than in an extension, because the complete set of designated initializers is part of the interface contract with subclasses of a class.

The other, non-designted initializers of a class are called convenience initializers, which tend to provide additional initialization capabilities that are often more convenient for common tasks.

9.2.2 Convenience Initializers

A *convenience initializer* is an initializer that provides an alternative interface to the designated initializers of a class. A convenience initializer is denoted by the return type `Self` in the definition. Unlike designated initializers, convenience initializers can be defined either in the class definition itself or within an extension of the class. For example:

```
extension A {
  init() -> Self {
    self.init(int: 17, string: "Seventeen")
  }
}
```

A convenience initializer cannot initialize the stored properties of the class directly, nor can it invoke a superclass initializer via `super.init`. Rather, it must *dispatch* to another initializer using `self.init`, which is then responsible for initializing the object. A convenience initializer is not permitted to access `self` (or anything that depends on `self`, such as one of its properties) prior to the `self.init` call, although it may freely access `self` after `self.init`.

Convenience initializers and designated initializers can both be used to construct objects, using the same syntax. For example, the `A` initializer above can be used to build a new `A` object without any arguments:

```
var a2 = A() // uses convenience initializer
```

9.2.3 Initializer Inheritance

One of the primary benefits of convenience initializers is that they can be inherited by subclasses. Initializer inheritance eliminates the need to repeat common initialization code—such as initial values of stored properties not easily written in the class itself, or common registration tasks that occur during initialization—while using the same initialization syntax. For example, this allows a `B` object to be constructed with no arguments by using the inherited convenience initializer defined in the previous section:

```
var b2 = B()
```

Initialization proceeds as follows:

1. A `B` object is allocated by the runtime.
2. `A`'s convenience initializer `init()` is invoked.
3. `A`'s convenience initializer dispatches to `init int:string:` via the `self.init` call. This call dynamically resolves to `B`'s designated initializer.
4. `B`'s designated initializer initializes the stored property `d` to `17.0`.
5. `B`'s designated initializer chains to `A`'s designated initializer.
6. `A`'s designated initializer initialize's the stored properties `i` and `s`'.
7. `A`'s designated initializer calls `completeInit()`, then returns.

8. `B`'s designated initializer calls `completeInitForB()`, then returns.
9. `A`'s convenience initializer returns.

Convenience initializers are only inherited under certain circumstances. Specifically, for a given subclass to inherit the convenience initializers of its superclass, the subclass must override each of the designated initializers of its superclass. For example `B` provides the initializer `init int:string:`, which overrides `A`'s designated initializer `init int:string:` because the initializer name and parameters are the same. If we had some other subclass `OtherB` of `A` that did not provide such an override, it would not inherit `A`'s convenience initializers:

```
class OtherB : A {
  var d: Double

  init int(i: Int) string(s: String) double(d: Double) {
    self.d = d                    // initialize stored properties
    super.init(int: i, string: s) // chain to superclass
  }
}

var ob = OtherB()   // error: A's convenience initializer init() not inherited
```

Note

The requirement that a subclass override all of the designated initializers of its superclass to enable initializer inheritance is crucial to Swift's memory safety model. See *Initializer Inheritance Model* for more information.

Note that a subclass may have different designated initializers from its superclass. This can occur in a number of ways. For example, the subclass might override one of its superclass's designated initializers with a convenience initializer:

```
class YetAnotherB : A {
  var d: Double

  init int(i: Int) string(s: String) -> Self {
    self.init(int: i, string: s, double: Double(i)) // dispatch
  }

  init int(i: Int) string(s: String) double(d: Double) {
    self.d = d                    // initialize stored properties
    super.init(int: i, string: s) // chain to superclass
  }
}

var yab = YetAnotherB()   // okay: YetAnotherB overrides all of A's designated␣
↪initializers
```

In other cases, it's possible that the convenience initializers of the superclass simply can't be made to work, because the subclass initializers require additional information provided via a parameter that isn't present in the convenience initializers of the superclass:

```
class PickyB : A {
  var notEasy: NoEasyDefault

  init int(i: Int) string(s: String) notEasy(NoEasyDefault) {
    self.notEasy = notEasy
    super.init(int: i, string: s) // chain to superclass
  }
}
```

Here, PickyB has a stored property of a type NoEasyDefault that can't easily be given a default value: it has to be provided as a parameter to one of PickyB's initializers. Therefore, PickyB takes over responsibility for its own initialization, and none of A's convenience initializers will be inherited into PickyB.

9.2.4 Synthesized Initializers

When a particular class does not specify any designated initializers, the implementation will synthesize initializers for the class when all of the class's stored properties have initial values in the class. The form of the synthesized initializers depends on the superclass (if present).

When a superclass is present, the compiler synthesizes a new designated initializer in the subclass for each designated initializer of the superclass. For example, consider the following class C:

```
class C : B {
  var title: String = "Default Title"
}
```

The superclass B has a single designated initializer,:

```
init int(i: Int) string(s: String)
```

Therefore, the compiler synthesizes the following designated initializer in C, which chains to the corresponding designated initializer in the superclass:

```
init int(i: Int) string(s: String) {
  // title is already initialized in the class C
  super.init(int: i, string: s)
}
```

The result of this synthesis is that all designated initializers of the superclass are (automatically) overridden in the subclass, becoming designated initializers of the subclass as well. Therefore, any convenience initializers in the superclass are also inherited, allowing the subclass (C) to be constructed with the same initializers as the superclass (B):

```
var c1 = C(int: 17, string: "Seventeen")
var c2 = C()
```

When the class has no superclass, a default initializer (with no parameters) is implicitly defined:

```
class D {
  var title = "Default Title"

  /* implicitly defined */
  init() { }
}

var d = D() // uses implicitly-defined default initializer
```

9.2.5 Required Initializers

Objects are generally constructed with the construction syntax T(...) used in all of the examples above, where T is the name of the type. However, it is occasionally useful to construct an object for which the actual type is not

known until runtime. For example, one might have a `View` class that expects to be initialized with a specific set of coordinates:

```
struct Rect {
  var origin: (Int, Int)
  var dimensions: (Int, Int)
}

class View {
  init frame(Rect) { /* initialize view */ }
}
```

The actual initialization of a subclass of `View` would then be performed at runtime, with the actual subclass being determined via some external file that describes the user interface. The actual instantiation of the object would use a type value:

```
func createView(viewClass: View.Type, frame: Rect) -> View {
  return viewClass(frame: frame) // error: 'init frame:' is not 'required'
}
```

The code above is invalid because there is no guarantee that a given subclass of `View` will have an initializer `init frame:`, because the subclass might have taken over its own initialization (as with `PickyB`, above). To require that all subclasses provide a particular initializer, use the `required` attribute as follows:

```
class View {
  @required init frame(Rect) {
    /* initialize view */
  }
}

func createView(viewClass: View.Type, frame: Rect) -> View {
  return viewClass(frame: frame) // okay
}
```

The `required` attribute allows the initializer to be used to construct an object of a dynamically-determined subclass, as in the `createView` method. It places the (transitive) requirement on all subclasses of `View` to provide an initializer `init frame:`. For example, the following `Button` subclass would produce an error:

```
class Button : View {
  // error: 'Button' does not provide required initializer 'init frame:'.
}
```

The fix is to implement the required initializer in `Button`:

```
class Button : View {
  @required init frame(Rect) { // okay: satisfies requirement
    super.init(frame: frame)
  }
}
```

9.2.6 Initializers in Protocols

Initializers may be declared within a protocol. For example:

```
protocol DefaultInitializable {
  init()
}
```

Note

Initializers in protocols have not yet been implemented. Stay tuned.

A class can satisfy this requirement by providing a required initializer. For example, only the first of the two following classes conforms to its protocol:

```
class DefInit : DefaultInitializable {
  @required init() { }
}

class AlmostDefInit : DefaultInitializable {
  init() { } // error: initializer used for protocol conformance must be 'required'
}
```

The `required` requirement ensures that all subclasses of the class declaring conformance to the protocol will also have the initializer, so they too will conform to the protocol. This allows one to construct objects given type values of protocol type:

```
func createAnyDefInit(typeVal: DefaultInitializable.Type) -> DefaultInitializable {
  return typeVal()
}
```

9.2.7 De-initializers

While initializers are responsible for setting up an object's state, *de-initializers* are responsible for tearing down that state. Most classes don't require a de-initializer, because Swift automatically releases all stored properties and calls to the superclass's de-initializer. However, if your class has allocated a resource that is not an object (say, a Unix file descriptor) or has registered itself during initialization, one can write a de-initializer using `deinit`:

```
class FileHandle {
  var fd: Int32

  init withFileDescriptor(fd: Int32) {
    self.fd = fd
  }

  deinit {
    close(fd)
  }
}
```

The statements within a de-initializer (here, the call to `close`) execute first, then the superclass's de-initializer is called. Finally, stored properties are released and the object is deallocated.

9.2.8 Methods Returning `Self`

A class method can have the special return type `Self`, which refers to the dynamic type of `self`. Such a method guarantees that it will return an object with the same dynamic type as `self`. One of the primary uses of the `Self`

return type is for factory methods:

```
extension View {
  class func createView(frame: Rect) -> Self {
    return self(frame: frame)
  }
}
```

Note

The return type `Self` fulfills the same role as Objective-C's `instancetype`, although Swift provides stronger type checking for these methods.

Within the body of this class method, the implicit parameter `self` is a value with type `View.Type`, i.e., it's a type value for the class `View` or one of its subclasses. Therefore, the restrictions are the same as for any value of type `View.Type`: one can call other class methods and construct new objects using required initializers of the class, among other things. The result returned from such a method must be derived from the type of `Self`. For example, it cannot return a value of type `View`, because `self` might refer to some subclass of `View`.

Instance methods can also return `Self`. This is typically used to allow chaining of method calls by returning `Self` from each method, as in the builder pattern:

```
class DialogBuilder {
  func setTitle(title: String) -> Self {
    // set the title
    return self;
  }

  func setBounds(frame: Rect) -> Self {
    // set the bounds
    return self;
  }
}

var builder = DialogBuilder()
                .setTitle("Hello, World!")
                .setBounds(Rect(0, 0, 640, 480))
```

9.3 Memory Safety

Swift aims to provide memory safety by default, and much of the design of Swift's object initialization scheme is in service of that goal. This section describes the rationale for the design based on the memory-safety goals of the language.

9.3.1 Three-Phase Initialization

The three-phase initialization model used by Swift's initializers ensures that all stored properties get initialized before any code can make use of `self`. This is important uses of `self`—say, calling a method on `self`—could end up referring to stored properties before they are initialized. Consider the following Objective-C code, where instance variables are initialized *after* the call to the the superclass initializer:

```
@interface A : NSObject
- (instancetype)init;
- (void)finishInit;
@end

@implementation A
- (instancetype)init {
  self = [super init];
  if (self) {
    [self finishInit];
  }
  return self;
}
@end

@interface B : A
@end

@implementation B {
  NSString *ivar;
}

- (instancetype)init {
  self = [super init];
  if (self) {
    self->ivar = @"Default name";
  }
  return self;
}

- (void) finishInit {
  NSLog(@"ivar has the value %@\n", self->ivar);
}
@end
```

Notes

In Objective-C, `+alloc` zero-initializes all of the instance variables, which gives them predictable behavior before the init method gets to initialize them. Given that Objective-C is fairly resilient to `nil` objects, this default behavior eliminates (or hides) many such initialization bugs. In Swift, however, the zero-initialized state is less likely to be valid, and the memory safety goals are stronger, so zero-initialization does not suffice.

When initializing a `B` object, the `NSLog` statement will print:

```
ivar has the value (null)
```

because `-[B finishInit]` executes before `B` has had a chance to initialize its instance variables. Swift initializers avoid this issue by splitting each initializer into three phases:

1. Initialize stored properties. In this phase, the compiler verifies that `self` is not used except when writing to the stored properties of the current class (not its superclasses!). Additionally, this initialization directly writes to the storage of the stored properties, and does not call any setter or `willSet`/`didSet` method. In this phase, it is not possible to read any of the stored properties.

2. Call to superclass initializer, if any. As with the first step, `self` cannot be accessed at all.

3. Perform any additional initialization tasks, which may call methods on `self`, access properties, and so on.

Note that, with this scheme, `self` cannot be used until the original class and all of its superclasses have initialized their stored properties, closing the memory safety hole.

9.3.2 Initializer Inheritance Model

FIXME: To be written

9.4 Objective-C Interoperability

9.4.1 Initializers and Init Methods

9.4.2 Designated and Convenience Initializers

9.4.3 Allocation and Deallocation

9.4.4 Dynamic Subclassing

CHAPTER 10

Pattern Matching

Warning: This document was used in designing the pattern-matching features of Swift 1.0. It has not been kept to date and does not describe the current or planned behavior of Swift.

10.1 Elimination rules

When type theorists consider a programming language, we break it down like this:

- What are the kinds of fundamental and derived types in the language?
- For each type, what are its introduction rules, i.e. how do you get values of that type?
- For each type, what are its elimination rules, i.e. how do you use values of that type?

Swift has a pretty small set of types right now:

- Fundamental types: currently i1, i8, i16, i32, and i64; float and double; eventually maybe others.
- Function types.
- Tuples. Heterogenous fixed-length products. Swift's system provides two basic kinds of element: positional and labelled.
- Arrays. Homogenous fixed-length aggregates.
- Algebraic data types (ADTs), introduce by enum. Nominal closed disjoint unions of heterogenous types.
- Struct types. Nominal heterogenous fixed-length products.
- Class types. Nominal, subtypeable heterogenous fixed-length products with identity.
- Protocol and protocol-composition types.

In addition, each of the nominal types can be made generic; this doesn't affect the overall introduction/elimination design because an "unapplied" generic type isn't first-class (intentionally), and an "applied" generic type behaves essentially like a non-generic type (also intentionally).

The point is that adding any other kind of type (e.g. SIMD vectors) means that we need to consider its intro/elim rules.

For most of these, intro rules are just a question of picking syntax, and we don't really need a document for that. So let's talk elimination. Generally, an elimination rule is a way at getting back to the information the intro rule(s) wrote into the value. So what are the specific elimination rules for these types? How do we use them, other than in type-generic ways like passing them as arguments to calls?

Functions are used by calling them. This is something of a special case: some values of function type may carry data, there isn't really a useful model for directly accessing it. Values of function type are basically completely opaque, except that we do provide thin vs. thick function types, which is potentially something we could pattern-match on, although many things can introduce thunks and so the result would not be reliable.

Scalars are used by feeding them to primitive binary operators. This is also something of a special case, because there's no useful way in which scalars can be decomposed into separate values.

Tuples, **structs**, and **classes** are used by projecting out their elements. Classes may also be turned into an object of a supertype (which is always a class).

Arrays are used by projecting out slices and elements.

Existentials are used by performing one of the operations that the type is known to support.

ADTs are used by projecting out elements of the current alternative, but how we determine the current alternative?

10.2 Alternatives for alternatives

I know of three basic designs for determining the current alternative of an ADT:

- Visitor pattern: there's some way of declaring a method on the full ADT and then implementing it for each individual alternative. You do this in OO languages mostly because there's no direct language support for closed disjoint unions (as opposed to open disjoint unions, which subclassing lets you achieve at some performance cost).
  - plus: doesn't require language support
  - plus: easy to "overload" and provide different kinds of pattern matching on the same type
  - plus: straightforward to add interesting ADT-specific logic, like matching a CallExpr instead of each of its N syntactic forms
  - plus: simple form of exhaustiveness checking
  - minus: cases are separate functions, so data and control flow is awkward
  - minus: lots of boilerplate to enable
  - minus: lots of boilerplate to use
  - minus: nested pattern matching is awful
- Query functions: dynamic_cast, dyn_cast, isa, instanceof
  - plus: easy to order and mix with other custom conditions
  - plus: low syntactic overhead for testing the alternative if you don't need to actually decompose
  - minus: higher syntactic overhead for decomposition
    - isa/instanceof pattern requires either a separate cast or unsafe operations later
    - dyn_cast pattern needs a fresh variable declaration, which is very awkward in complex conditions
  - minus: exhaustiveness checking is basically out the window

  - minus: some amount of boilerplate to enable

- Pattern matching
  - plus: no boilerplate to enable
  - plus: hugely reduced syntax to use if you want a full decomposition
  - plus: compiler-supported exhaustiveness checking
  - plus: nested matching is natural
  - plus: with pattern guards, natural mixing of custom conditions
  - minus: syntactic overkill to just test for a specific alternative (e.g. to filter it out)
  - minus: needs boilerplate to project out a common member across multiple/all alternatives
  - minus: awkward to group alternatives (fallthrough is a simple option but has issues)
  - minus: traditionally completely autogenerated by compiler and thus not very flexible
  - minus: usually a new grammar production that's very ambiguous with the expression grammar
  - minus: somewhat fragile against adding extra data to an alternative

I feel that this strongly points towards using pattern matching as the basic way of consuming ADTs, maybe with special dispensations for querying the alternative and projecting out common members.

Pattern matching was probably a foregone conclusion, but I wanted to spell out that having ADTs in the language is what really forces our hand because the alternatives are so bad. Once we need pattern-matching, it makes sense to provide patterns for the other kinds of types as well.

10.3 Selection statement

This is the main way we expect users to employ non-obvious pattern- matching. We obviously need something with statement children, so this has to be a statement. That's also fine because this kind of full pattern match is very syntactically heavyweight, and nobody would want to embed it in the middle of an expression. We also want a low-weight matching expression, though, for relatively simple ADTs:

```
stmt             ::= stmt-switch
stmt-switch      ::= 'switch' expr '{' switch-group+ '}'
switch-group     ::= case-introducer+ stmt-brace-item+
case-introducer ::= 'case' match-pattern-list case-guard? ':'
case-introducer ::= 'default' case-guard? ':'
case-guard       ::= 'where' expr
match-pattern-list ::= match-pattern
match-pattern-list ::= match-pattern-list ',' match-pattern
```

We can get away with using "switch" here because we're going to unify both values and patterns under match-pattern. The works chiefly by making decompositional binding a bit more awkward, but has the major upside of reducing the likelihood of dumb mistakes (rebinding 'true', for example), and it means that C-looking switches actually match our semantics quite closely. The latter is something of a priority: a C switch over an enum is actually pretty elegant — well, except for all the explicit scoping and 'break' statements, but the switching side of it feels clean.

10.3.1 Default

I keep going back and forth about having a "default" case-introducer. On the one hand, I kindof want to encourage total matches. On the other hand, (1) having it is consistent with C, (2) it's not an unnatural style, and (3) there are cases

where exhaustive switching isn't going to be possible. We can certainly recommend complete matches in switches, though.

If we do have a 'default', I think it makes the most sense for it to be semantically a complete match and therefore require it to be positioned at the end (on pain of later matches being irrelevant). First, this gives more sensible behavior to 'default where x.isPurple()', which really doesn't seem like it should get reordered with the surrounding cases; and second, it makes the matching story very straightforward. And users who like to put 'default:' at the top won't accidentally get unexpected behavior because coverage checking will immediately complain about the fact that every case after an unguarded 'default' is obviously dead.

10.3.2 Case groups

A case-group lets you do the same thing for multiple cases without an extra syntactic overhead (like a 'fallthrough' after every case). For some types (e.g. classic functional linked lists) this is basically pointless, but for a lot of other types (Int, enums, etc.) it's pervasive.

The most important semantic design point here is about bound variables in a grouped case, e.g. (using 'var' as a "bind this variable" introducer; see the pattern grammar):

```
switch (pair) {
case (var x, 0):
case (0, var y):
  return 1
case (var x, var y)
  return foo(x-1,y) + foo(x,y-1)
}
```

It's tempting to just say that an unsound name binding (i.e. a name not bound in all cases or bound to values of different types) is just always an error, but I think that's probably not the way to go. There are two things I have in mind here: first, these variables can be useful in pattern guards even if they're not used in the case block itself, and second, a well-chosen name can make a pattern much more self-documenting. So I think it should only be an error to *refer* to an unsound name binding.

The most important syntactic design point here is whether to require (or even allow) the 'case' keyword to be repeated for each case. In many cases, it can be much more compact to allow a comma-separated list of patterns after 'case':

```
switch (day) {
case .Terrible, .Horrible, .NoGood, .VeryBad:
  abort()
case .ActuallyPrettyReasonableWhenYouLookBackOnIt:
  continue
}
```

or even more so:

```
case 0...2, 5...10, 14...18, 22...:
  flagConditionallyAcceptableAge()
```

On the other hand, if this list gets really long, the wrapping gets a little weird:

```
case .Terrible, .Horrible, .NoGood, .VeryBad,
     .Awful, .Dreadful, .Appalling, .Horrendous,
     .Deplorable, .Unpleasant, .Ghastly, .Dire:
  abort()
```

And while I think pattern guards should be able to apply to multiple cases, it would be nice to allow different cases in a group to have different pattern guards:

```
case .None:
case .Some(var c) where c.isSpace() || c.isASCIIControl():
  skipToEOL()
```

So really I think we should permit multiple 'case' introducers:

```
case .Terrible, .Horrible, .NoGood, .VeryBad:
case .Awful, .Dreadful, .Appalling, .Horrendous:
case .Deplorable, .Unpleasant, .Ghastly, .Dire:
  abort()
```

With the rule that a pattern guard can only use bindings that are sound across its guarded patterns (those within the same 'case'), and the statement itself can only use bindings that are sound across all of the cases. A reference that refers to an unsound binding is an error; lookup doesn't just ignore the binding.

10.3.3 Scoping

Despite the lack of grouping braces, the semantics are that the statements in each case-group form their own scope, and falling off the end causes control to resume at the end of the switch statement — i.e. "implicit break", not "implicit fallthrough".

Chris seems motivated to eventually add an explicit 'fallthrough' statement. If we did this, my preference would be to generalize it by allowing the match to be reperformed with a new value, e.g. `fallthrough(something)`, at least optionally. I think having local functions removes a lot of the impetus, but not so much as to render the feature worthless.

Syntactically, braces and the choice of case keywords are all bound together. The thinking goes as follows. In Swift, statement scopes are always grouped by braces. It's natural to group the cases with braces as well. Doing both lets us avoid a 'case' keyword, but otherwise it leads to ugly style, because either the last case ends in two braces on the same line or cases have to further indented. Okay, it's easy enough to not require braces on the match, with the grammar saying that cases are just greedily consumed — there's no ambiguity here because the switch statement is necessarily within braces. But that leaves the code without a definitive end to the cases, and the closing braces end up causing a lot of unnecessary vertical whitespace, like so:

```
switch (x)
case .foo {
  ...
}
case .bar {
  ...
}
```

So instead, let's require the switch statement to have braces, and we'll allow the cases to be written without them:

```
switch (x) {
case .foo:
  ...
case .bar:
  ...
}
```

That's really a lot prettier, except it breaks the rule about always grouping scopes with braces (we *definitely* want different cases to establish different scopes). Something has to give, though.

We require the trailing colon because it's a huge cue for separating things, really making single-line cases visually appealing, and the fact that it doesn't suggest closing punctuation is a huge boon. It's also directly precedented in C,

and it's even roughly the right grammatical function.

10.3.4 Case selection semantics

The semantics of a switch statement are to first evaluate the value operand, then proceed down the list of case-introducers and execute the statements for the switch-group that had the first satisfied introducer.

It is an error if a case-pattern can never trigger because earlier cases are exhaustive. Some kinds of pattern (like 'default' cases and '_') are obviously exhaustive by themselves, but other patterns (like patterns on properties) can be much harder to reason about exhaustiveness for, and of course pattern guards can make this outright undecidable. It may be easiest to apply very straightforward rules (like "ignore guarded patterns") for the purposes of deciding whether the program is actually ill-formed; anything else that we can prove is unreachable would only merit a warning. We'll probably also want a way to say explicitly that a case can never occur (with semantics like llvm_unreachable, i.e. a reliable runtime failure unless that kind of runtime safety checking is disabled at compile-time).

A 'default' is satisfied if it has no guard or if the guard evaluates to true.

A 'case' is satisfied if the pattern is satisfied and, if there's a guard, the guard evaluates to true after binding variables. The guard is not evaluated if the pattern is not fully satisfied. We'll talk about satisfying a pattern later.

10.3.5 Non-exhaustive switches

Since falling out of a statement is reasonable behavior in an imperative language — in contrast to, say, a functional language where you're in an expression and you need to produce a value — there's a colorable argument that non-exhaustive matches should be okay. I dislike this, however, and propose that it should be an error to make an non-exhaustive switch; people who want non-exhaustive matches can explicitly put in default cases. Exhaustiveness actually isn't that difficult to check, at least over ADTs. It's also really the behavior that I would expect from the syntax, or at least implicitly falling out seems dangerous in a way that nonexhaustive checking doesn't. The complications with checking exhaustiveness are pattern guards and matching expressions. The obvious conservatively-safe rule is to say "ignore cases with pattern guards or matching expressions during exhaustiveness checking", but some people really want to write "where x < 10" and "where x >= 10", and I can see their point. At the same time, we really don't want to go down that road.

10.4 Other uses of patterns

Patterns come up (or could potentially come up) in a few other places in the grammar:

10.4.1 Var bindings

Variable bindings only have a single pattern, which has to be exhaustive, which also means there's no point in supporting guards here. I think we just get this:

```
decl-var ::= 'var' attribute-list? pattern-exhaustive value-specifier
```

10.4.2 Function parameters

The functional languages all permit you to directly pattern-match in the function declaration, like this example from SML:

```
fun length nil = 0
  | length (a::b) = 1 + length b
```

This is really convenient, but there's probably no reasonable analogue in Swift. One specific reason: we want functions to be callable with keyword arguments, but if you don't give all the parameters their own names, that won't work.

The current Swift approximation is:

```
func length(list : List) : Int {
  switch list {
    case .nil: return 0
    case .cons(_,var tail): return 1 + length(tail)
  }
}
```

That's quite a bit more syntax, but it's mostly the extra braces from the function body. We could remove those with something like this:

```
func length(list : List) : Int = switch list {
  case .nil: return 0
  case .cons(_,var tail): return 1 + length(tail)
}
```

Anyway, that's easy to add later if we see the need.

10.4.3 Assignment

This is a bit iffy. It's a lot like var bindings, but it doesn't have a keyword, so it's really kindof ambiguous given the pattern grammar.

Also, l-value patterns are weird. I can come up with semantics for this, but I don't know what the neighbors will think:

```
var perimeter : double
.feet(x) += yard.dimensions.height // returns Feet, which has one constructor, :feet.
.feet(x) += yard.dimensions.width
```

It's probably better to just have l-value tuple expressions and not try to work in arbitrary patterns.

10.4.4 Pattern-match expression

This is an attempt to provide that dispensation for query functions we were talking about.

I think this should bind looser than any binary operators except assignments; effectively we should have:

```
expr-binary ::= # most of the current expr grammar

expr ::= expr-binary
expr ::= expr-binary 'is' expr-primary pattern-guard?
```

The semantics are that this evaluates to true if the pattern and pattern-guard are satisfied.

'is' or 'isa'

Perl and Ruby use '=~' as the regexp pattern-matching operator, which is both obscure and really looks like an assignment operator, so I'm stealing Joe's 'is' operator, which is currently used for dynamic type-checks. I'm of two

minds about this: I like 'is' a lot for value-matching, but not for dynamic type-checks.

One possibility would be to use 'is' as the generic pattern-matching operator but use a different spelling (like 'isa') for dynamic type-checks, including the 'is' pattern. This would give us "x isa NSObject" as an expression and "case isa NSObject:" as a case selector, both of which I feel read much better. But in this proposal, we just use a single operator.

Other alternatives to 'is' include 'matches' (reads very naturally but is somewhat verbose) or some sort of novel operator like '~~'.

Note that this impacts a discussion in the section below about expression patterns.

Dominance

I think that this feature is far more powerful if the name bindings, type-refinements, etc. from patterns are available in code for which a trivial analysis would reveal that the result of the expression is true. For example:

```
if s is Window where x.isVisible {
  // can use Window methods on x here
}
```

Taken a bit further, we can remove the need for 'where' in the expression form:

```
if x is Window && x.isVisible { ... }
```

That might be problematic without hard-coding the common control-flow operators, though. (As well as hardcoding some assumptions about Bool.convertToLogicValue...)

10.5 Pattern grammar

The usual syntax rule from functional languages is that the pattern grammar mirrors the introduction-rule expression grammar, but parses a pattern wherever you would otherwise put an expression. This means that, for example, if we add array literal expressions, we should also add a corresponding array literal pattern. I think that principle is very natural and worth sticking to wherever possible.

10.5.1 Two kinds of pattern

We're blurring the distinction between patterns and expressions a lot here. My current thinking is that this simplifies things for the programmer — the user concept becomes basically "check whether we're equal to this expression, but allow some holes and some more complex 'matcher' values". But it's possible that it instead might be really badly confusing. We'll see! It'll be fun!

This kindof forces us to have parallel pattern grammars for the two major clients:

- Match patterns are used in `switch` and `matches`, where we're decomposing something with a real possibility of failing. This means that expressions are okay in leaf positions, but that name-bindings need to be explicitly advertised in some way to reasonably disambiguate them from expressions.
- Exhaustive patterns are used in `var` declarations and function signatures. They're not allowed to be non-exhaustive, so having a match expression doesn't make any sense. Name bindings are common and so shouldn't be penalized.

You might think that having a "pattern" as basic as `foo` mean something different in two different contexts would be confusing, but actually I don't think people will generally think of these as the same production — you might if you were in a functional language where you really can decompose in a function signature, but we don't allow that, and I think that will serve to divide them in programmers' minds. So we can get away with some things. :)

10.5.2 Binding patterns

In general, a lot of these productions are the same, so I'm going to talk about *-patterns, with some specific special rules that only apply to specific pattern kinds.

```
*-pattern ::= '_'
```

A single-underscore identifier is always an "ignore" pattern. It matches anything, but does not bind it to a variable.

```
exhaustive-pattern ::= identifier
match-pattern ::= '?' identifier
```

Any more complicated identifier is a variable-binding pattern. It is illegal to bind the same identifier multiple times within a pattern. However, the variable does come into scope immediately, so in a match pattern you can have a latter expression which refers to an already-bound variable. I'm comfortable with constraining this to only work "conveniently" left-to-right and requiring more complicated matches to use guard expressions.

In a match pattern, variable bindings must be prefixed with a ? to disambiguate them from an expression consisting of a variable reference. I considered using 'var' instead, but using punctuation means we don't need a space, which means this is much more compact in practice.

10.5.3 Annotation patterns

```
exhaustive-pattern ::= exhaustive-pattern ':' type
```

In an exhaustive pattern, you can annotate an arbitrary sub-pattern with a type. This is useful in an exhaustive pattern: the type of a variable isn't always inferrable (or correctly inferrable), and types in function signatures are generally outright required. It's not as useful in a match pattern, and the colon can be grammatically awkward there, so we disallow it.

10.5.4 'is' patterns

```
match-pattern ::= 'is' type
```

This pattern is satisfied if the dynamic type of the matched value "satisfies" the named type:

- if the named type is an Objective-C class type, the dynamic type must be a class type, and an 'isKindOf:' check is performed;
- if the named type is a Swift class type, the dynamic type must be a class type, and a subtype check is performed;
- if the named type is a metatype, the dynamic type must be a metatype, and the object type of the dynamic type must satisfy the object type of the named type;
- otherwise the named type must equal the dynamic type.

This inquiry is about dynamic types; archetypes and existentials are looked through.

The pattern is ill-formed if it provably cannot be satisfied.

In a 'switch' statement, this would typically appear like this:

```
case is NSObject:
```

It can, however, appear in recursive positions:

```
case (is NSObject, is NSObject):
```

Ambiguity with type value matching

There is a potential point of confusion here with dynamic type checking (done by an 'is' pattern) vs. value equality on type objects (done by an expression pattern where the expression is of metatype type. This is resolved by the proposal (currently outstanding but generally accepted, I think) to disallow naked references to type constants and instead require them to be somehow decorated.

That is, this pattern requires the user to write something like this:

```
case is NSObject:
```

It is quite likely that users will often accidentally write something like this:

```
case NSObject:
```

It would be very bad if that were actually accepted as a valid expression but with the very different semantics of testing equality of type objects. For the most part, type-checking would reject that as invalid, but a switch on (say) a value of archetype type would generally work around that.

However, we have an outstanding proposal to generally forbid 'NSObject' from appearing as a general expression; the user would have to decorate it like the following, which would let us eliminate the common mistake:

```
case NSObject.type:
```

Type refinement

If the value matched is immediately the value of a local variable, I think it would be really useful if this pattern could introduce a type refinement within its case, so that the local variable would have the refined type within that scope. However, making this kind of type refinement sound would require us to prevent there from being any sort of mutable alias of the local variable under an unrefined type. That's usually going to be fine in Swift because we usually don't permit the address of a local to escape in a way that crosses statement boundaries. However, closures are a major problem for this model. If we had immutable local bindings — and, better yet, if they were the default — this problem would largely go away.

This sort of type refinement could also be a problem with code like:

```
while expr is ParenExpr {
  expr = expr.getSubExpr()
}
```

It's tricky.

10.5.5 "Call" patterns

```
match-pattern ::= match-pattern-identifier match-pattern-tuple?
match-pattern-identifier ::= '.' identifier
match-pattern-identifier ::= match-pattern-identifier-tower
match-pattern-identifier-tower ::= identifier
match-pattern-identifier-tower ::= identifier
match-pattern-identifier-tower ::= match-pattern-identifier-tower '.' identifier
```

A match pattern can resemble a global name or a call to a global name. The global name is resolved as normal, and then the pattern is interpreted according to what is found:

- If the name resolves to a type, then the dynamic type of the matched value must match the named type (according to the rules below for 'is' patterns). It is okay for this to be trivially true.

  In addition, there must be an non-empty arguments clause, and each element in the clause must have an identifier. For each element, the identifier must correspond to a known property of the named type, and the value of that property must satisfy the element pattern.

- If the name resolves to a enum element, then the dynamic type of the matched value must match the enum type as discussed above, and the value must be of the specified element. There must be an arguments clause if and only if the element has a value type. If so, the value of the element is matched against the clause pattern.

- Otherwise, the argument clause (if present) must also be syntactically valid as an expression, and the entire pattern is reinterpreted as an expression.

This is all a bit lookup-sensitive, which makes me uncomfortable, but otherwise I think it makes for attractive syntax. I'm also a little worried about the way that, say, `f(x)` is always an expression but `A(x)` is a pattern. Requiring property names when matching properties goes some way towards making that okay.

I'm not totally sold on not allowing positional matching against struct elements; that seems unfortunate in cases where positionality is conventionally unambiguous, like with a point.

Matching against struct types requires arguments because this is intended to be used for structure decomposition, not dynamic type testing. For the latter, an 'is' pattern should be used.

10.5.6 Expression patterns

```
match-pattern ::= expression
```

When ambiguous, match patterns are interpreted using a pattern-specific production. I believe it should be true that, in general, match patterns for a production accept a strict superset of valid expressions, so that (e.g.) we do not need to disambiguate whether an open paren starts a tuple expression or a tuple pattern, but can instead just aggressively parse as a pattern. Note that binary operators can mean that, using this strategy, we sometimes have to retroactively rewrite a pattern as an expression.

It's always possible to disambiguate something as an expression by doing something not allowing in patterns, like using a unary operator or calling an identity function; those seem like unfortunate language solutions, though.

10.5.7 Satisfying an expression pattern

A value satisfies an expression pattern if the match operation succeeds. I think it would be natural for this match operation to be spelled the same way as that match-expression operator, so e.g. a member function called 'matches' or a global binary operator called '~' or whatever.

The lookup of this operation poses some interesting questions. In general, the operation itself is likely to be associated with the intended type of the expression pattern, but that type will often require refinement from the type of the matched value.

For example, consider a pattern like this:

```
case 0...10:
```

We should be able to use this pattern when switching on a value which is not an Int, but if we type-check the expression on its own, we will assign it the type Range<Int>, which will not necessarily permit us to match (say) a UInt8.

10.5.8 Order of evaluation of patterns

I'd like to keep the order of evaluation and testing of expressions within a pattern unspecified if I can; I imagine that there should be a lot of cases where we can rule out a case using a cheap test instead of a more expensive one, and it would suck to have to run the expensive one just to have cleaner formal semantics. Specifically, I'm worried about cases like `case [foo(), 0]:`; if we can test against 0 before calling `foo()`, that would be great. Also, if a name is bound and then used directly as an expression later on, it would be nice to have some flexibility about which value is actually copied into the variable, but this is less critical.

```
*-pattern ::= *-pattern-tuple
*-pattern-tuple ::= '(' *-pattern-tuple-element-list? '...'? ')'
*-pattern-tuple-element-list ::= *-pattern-tuple-element
*-pattern-tuple-element-list ::= *-pattern-tuple-element ',' pattern-tuple-element-
↪list
*-pattern-tuple-element ::= *-pattern
*-pattern-tuple-element ::= identifier '=' *-pattern
```

Tuples are interesting because of the labelled / non-labelled distinction. Especially with labelled elements, it is really nice to be able to ignore all the elements you don't care about. This grammar permits some prefix or set of labels to be matched and the rest to be ignored.

10.6 Miscellaneous

It would be interesting to allow overloading / customization of pattern-matching. We may find ourselves needing to do something like this to support non-fragile pattern matching anyway (if there's some set of restrictions that make it reasonable to permit that). The obvious idea of compiling into the visitor pattern is a bit compelling, although control flow would be tricky — we'd probably need the generated code to throw an exception. Alternatively, we could let the non-fragile type convert itself into a fragile type for purposes of pattern matching.

If we ever allow infix ADT constructors, we'll need to allow them in patterns as well.

Eventually, we will build regular expressions into the language, and we will allow them directly as patterns and even bind grouping expressions into user variables.

John.

CHAPTER 11

Stored and Computed Variables

Warning: This document has not been updated since the initial design in Swift 1.0.

Variables are declared using the `var` keyword. These declarations are valid at the top level, within types, and within code bodies, and are respectively known as *global variables, member variables,* and *local variables.* Member variables are commonly referred to as *properties.*

Every variable declaration can be classified as either *stored* or *computed.* Member variables inherited from a superclass obey slightly different rules.

- *Stored Variables*
- *Computed Variables*
- *Observing Accessors*
- *Overriding Read-Only Variables*
- *Overriding Read-Write Variables*

11.1 Stored Variables

The simplest form of a variable declaration provides only a type:

```
var count : Int
```

This form of `var` declares a *stored variable.* Stored variables cause storage to be allocated in their containing context:

- a new global symbol for a global variable
- a slot in an object for a member variable

- space on the stack for a local variable

(Note that this storage may still be optimized away if determined unnecessary.)

Stored variables must be initialized before use. As such, an initial value can be provided at the declaration site. This is mandatory for global variables, since it cannot be proven who accesses the variable first.

```
var count : Int = 10
```

If the type of the variable can be inferred from the initial value expression, it may be omitted in the declaration:

```
var count = 10
```

Variables formed during pattern matching are also considered stored variables.

```
switch optVal {
case .Some(var actualVal):
  // do something
case .None:
  // do something else
}
```

11.2 Computed Variables

A *computed variable* behaves syntactically like a variable, but does not actually require storage. Instead, accesses to the variable go through "accessors" known as the *getter* and the *setter*. Thus, a computed variable is declared as a variable with a custom getter:

```
struct Rect {
  // Stored member variables
  var x, y, width, height : Int

  // A computed member variable
  var maxX : Int {
    get {
      return x + width
    }
    set(newMax) {
      x = newMax - width
    }
  }

// myRect.maxX = 40
```

In this example, no storage is provided for `maxX`.

If the setter's argument is omitted, it is assumed to be named `value`:

```
var maxY : Int {
  get {
    return y + height
  }
  set {
    y = value - height
  }
}
```

Finally, if a computed variable has a getter but no setter, it becomes a *read-only variable*. In this case the `get` label may be omitted. Attempting to set a read-only variable is a compile-time error:

```
  var area : Int {
    return self.width * self.height
  }
}
```

Note that because this is a member variable, the implicit parameter `self` is available for use within the accessors.

It is illegal for a variable to have a setter but no getter.

11.3 Observing Accessors

Occasionally it is useful to provide custom behavior when changing a variable's value that goes beyond simply modifying the underlying storage. One way to do this is to pair a stored variable with a computed variable:

```
var _backgroundColor : Color
var backgroundColor : Color {
  get {
    return _backgroundColor
  }
  set {
    _backgroundColor = value
    refresh()
  }
}
```

However, this contains a fair amount of boilerplate. For cases where a stored property provides the correct storage semantics, you can add custom behavior before or after the underlying assignment using "observing accessors" `willSet` and `didSet`:

```
var backgroundColor : Color {
  didSet {
    refresh()
  }
}

var currentURL : URL {
  willSet(newValue) {
    if newValue != currentURL {
      cancelCurrentRequest()
    }
  }
  didSet {
    sendNewRequest(currentURL)
  }
}
```

A stored property may have either observing accessor, or both. Like `set`, the argument for `willSet` may be omitted, in which case it is provided as "value":

```
var accountName : String {
  willSet {
    assert(value != "root")
```

```
  }
}
```

Observing accessors provide the same behavior as the two-variable example, with two important exceptions:

- A variable with observing accessors is still a stored variable, which means it must still be initialized before use. Initialization does not run the code in the observing accessors.
- All assignments to the variable will trigger the observing accessors with the following exceptions: assignments in the init and destructor function for the enclosing type, and those from within the accessors themselves. In this context, assignments directly store to the underlying storage.

Computed properties may not have observing accessors. That is, a property may have a custom getter or observing accessors, but not both.

11.4 Overriding Read-Only Variables

If a member variable within a class is a read-only computed variable, it may be overridden by subclasses. In this case, the subclass may choose to replace that computed variable with a stored variable by declaring the stored variable in the usual way:

```
class Base {
  var color : Color {
    return .Black
  }
}

class Colorful : Base {
  var color : Color
}

var object = Colorful(.Red)
object.color = .Blue
```

The new stored variable may have observing accessors:

```
class MemoryColorful : Base {
  var oldColors : Array<Color> = []

  var color : Color {
    willSet {
      oldColors.append(color)
    }
  }
}
```

A computed variable may also be overridden with another computed variable:

```
class MaybeColorful : Base {
  var color : Color {
    get {
      if randomBooleanValue() {
        return .Green
      } else {
        return super.color
      }
```

```
    }
    set {
      print("Sorry, we choose our own colors here.")
    }
  }
}
```

11.5 Overriding Read-Write Variables

If a member variable within a class as a read-write variable, it is not generally possible to know if it is a computed variable or stored variable. A subclass may override the superclass's variable with a new computed variable:

```
class ColorBase {
  var color : Color {
    didSet {
      print("I've been painted \(color)!")
    }
  }
}

class BrightlyColored : ColorBase {
  var color : Color {
    get {
      return super.color
    }
    set(newColor) {
      // Prefer whichever color is brighter.
      if newColor.luminance > super.color.luminance {
        super.color = newColor
      } else {
        // Keep the old color.
      }
    }
  }
}
```

In this case, because the superclass's `didSet` is part of the generated setter, it is only called when the subclass actually invokes setter through its superclass. On the `else` branch, the superclass's `didSet` is skipped.

A subclass may also use observing accessors to add behavior to an inherited member variable:

```
class TrackingColored : ColorBase {
  var prevColor : Color?

  var color : Color {
    willSet {
      prevColor = color
    }
  }
}
```

In this case, the `willSet` accessor in the subclass is called first, then the setter for `color` in the superclass. Critically, this is *not* declaring a new stored variable, and the subclass will *not* need to initialize `color` as a separate member variable.

Because observing accessors add behavior to an inherited member variable, a superclass's variable may not be overridden with a new stored variable, even if no observing accessors are specified. In the rare case where this is desired, the two-variable pattern shown *above* can be used.

CHAPTER 12

Swift Intermediate Language (SIL)

Contents

- *Swift Intermediate Language (SIL)*
  - *Abstract*
  - *SIL in the Swift Compiler*
    - *SILGen*
    - *Guaranteed Optimization and Diagnostic Passes*
    - *General Optimization Passes*
  - *Syntax*
    - *SIL Stage*
    - *SIL Types*
      - *Type Lowering*
      - *Abstraction Difference*
      - *Legal SIL Types*
      - *Address Types*
      - *Local Storage Types*
      - *Box Types*
      - *Function Types*
      - *Properties of Types*
      - *Layout Compatible Types*
    - *Values and Operands*

        * *Functions*
        * *Basic Blocks*
        * *Declaration References*
        * *Linkage*
            - *Definition of the* linked *relation*
            - *Requirements on linked objects*
            - *Summary*
        * *VTables*
        * *Witness Tables*
        * *Global Variables*
    - *Dataflow Errors*
        * *Definitive Initialization*
        * *Unreachable Control Flow*
    - *Runtime Failure*
    - *Undefined Behavior*
    - *Calling Convention*
        * *Swift Calling Convention @cc(swift)*
            - *Reference Counts*
            - *Address-Only Types*
            - *Variadic Arguments*
            - *Function Currying*
            - *@inout Arguments*
        * *Swift Method Calling Convention @cc(method)*
        * *Witness Method Calling Convention @cc(witness_method)*
        * *C Calling Convention @cc(cdecl)*
        * *Objective-C Calling Convention @cc(objc_method)*
            - *Reference Counts*
            - *Method Currying*
    - *Type Based Alias Analysis*
        * *Class TBAA*
        * *Typed Access TBAA*
    - *Value Dependence*
    - *Instruction Set*
        * *Allocation and Deallocation*
            - *alloc_stack*

        - *alloc_ref*
        - *alloc_ref_dynamic*
        - *alloc_box*
        - *alloc_value_buffer*
        - *dealloc_stack*
        - *dealloc_box*
        - *project_box*
        - *dealloc_ref*
        - *dealloc_partial_ref*
        - *dealloc_value_buffer*
        - *project_value_buffer*
    * *Debug Information*
        - *debug_value*
        - *debug_value_addr*
    * *Accessing Memory*
        - *load*
        - *store*
        - *assign*
        - *mark_uninitialized*
        - *mark_function_escape*
        - *copy_addr*
        - *destroy_addr*
        - *index_addr*
        - *index_raw_pointer*
    * *Reference Counting*
        - *strong_retain*
        - *strong_retain_autoreleased*
        - *strong_release*
        - *strong_retain_unowned*
        - *unowned_retain*
        - *unowned_release*
        - *load_weak*
        - *store_weak*
        - *fix_lifetime*
        - *mark_dependence*

        - *is_unique*
        - *is_unique_or_pinned*
        - *copy_block*
    * *Literals*
        - *function_ref*
        - *global_addr*
        - *integer_literal*
        - *float_literal*
        - *string_literal*
    * *Dynamic Dispatch*
        - *class_method*
        - *super_method*
        - *witness_method*
        - *dynamic_method*
    * *Function Application*
        - *apply*
        - *partial_apply*
        - *builtin*
    * *Metatypes*
        - *metatype*
        - *value_metatype*
        - *existential_metatype*
        - *objc_protocol*
    * *Aggregate Types*
        - *retain_value*
        - *release_value*
        - *autorelease_value*
        - *tuple*
        - *tuple_extract*
        - *tuple_element_addr*
        - *struct*
        - *struct_extract*
        - *struct_element_addr*
        - *ref_element_addr*
    * *Enums*

- *enum*
- *unchecked_enum_data*
- *init_enum_data_addr*
- *inject_enum_addr*
- *unchecked_take_enum_data_addr*
- *select_enum*
- *select_enum_addr*

* *Protocol and Protocol Composition Types*
  - *init_existential_addr*
  - *deinit_existential_addr*
  - *open_existential_addr*
  - *init_existential_ref*
  - *open_existential_ref*
  - *init_existential_metatype*
  - *open_existential_metatype*
  - *alloc_existential_box*
  - *open_existential_box*
  - *dealloc_existential_box*
* *Blocks*
  - *project_block_storage*
  - *init_block_storage_header*
* *Unchecked Conversions*
  - *upcast*
  - *address_to_pointer*
  - *pointer_to_address*
  - *unchecked_ref_cast*
  - *unchecked_ref_cast_addr*
  - *unchecked_addr_cast*
  - *unchecked_trivial_bit_cast*
  - *unchecked_bitwise_cast*
  - *ref_to_raw_pointer*
  - *raw_pointer_to_ref*
  - *ref_to_unowned*
  - *unowned_to_ref*
  - *ref_to_unmanaged*
  - *unmanaged_to_ref*

        - *convert_function*
        - *thin_function_to_pointer*
        - *pointer_to_thin_function*
        - *ref_to_bridge_object*
        - *bridge_object_to_ref*
        - *bridge_object_to_word*
        - *thin_to_thick_function*
        - *thick_to_objc_metatype*
        - *objc_to_thick_metatype*
        - *objc_metatype_to_object*
        - *objc_existential_metatype_to_object*
        - *is_nonnull*
    * *Checked Conversions*
        - *unconditional_checked_cast*
        - *unconditional_checked_cast_addr*
    * *Runtime Failures*
        - *cond_fail*
    * *Terminators*
        - *unreachable*
        - *return*
        - *autorelease_return*
        - *throw*
        - *br*
        - *cond_br*
        - *switch_value*
        - *select_value*
        - *switch_enum*
        - *switch_enum_addr*
        - *dynamic_method_br*
        - *checked_cast_br*
        - *checked_cast_addr_br*
        - *try_apply*
    * *Assertion configuration*

12.1 Abstract

SIL is an SSA-form IR with high-level semantic information designed to implement the Swift programming language. SIL accommodates the following use cases:

- A set of guaranteed high-level optimizations that provide a predictable baseline for runtime and diagnostic behavior.
- Diagnostic dataflow analysis passes that enforce Swift language requirements, such as definitive initialization of variables and constructors, code reachability, switch coverage.
- High-level optimization passes, including retain/release optimization, dynamic method devirtualization, closure inlining, memory allocation promotion, and generic function instantiation.
- A stable distribution format that can be used to distribute "fragile" inlineable or generic code with Swift library modules, to be optimized into client binaries.

In contrast to LLVM IR, SIL is a generally target-independent format representation that can be used for code distribution, but it can also express target-specific concepts as well as LLVM can.

12.2 SIL in the Swift Compiler

At a high level, the Swift compiler follows a strict pipeline architecture:

- The *Parse* module constructs an AST from Swift source code.
- The *Sema* module type-checks the AST and annotates it with type information.
- The *SILGen* module generates *raw SIL* from an AST.
- A series of *Guaranteed Optimization Passes* and *Diagnostic Passes* are run over the raw SIL both to perform optimizations and to emit language-specific diagnostics. These are always run, even at -Onone, and produce *canonical SIL*.
- General SIL *Optimization Passes* optionally run over the canonical SIL to improve performance of the resulting executable. These are enabled and controlled by the optimization level and are not run at -Onone.
- *IRGen* lowers canonical SIL to LLVM IR.
- The LLVM backend (optionally) applies LLVM optimizations, runs the LLVM code generator and emits binary code.

The stages pertaining to SIL processing in particular are as follows:

12.2.1 SILGen

SILGen produces *raw SIL* by walking a type-checked Swift AST. The form of SIL emitted by SILGen has the following properties:

- Variables are represented by loading and storing mutable memory locations instead of being in strict SSA form. This is similar to the initial `alloca`-heavy LLVM IR emitted by frontends such as Clang. However, Swift represents variables as reference-counted "boxes" in the most general case, which can be retained, released, and captured into closures.
- Dataflow requirements, such as definitive assignment, function returns, switch coverage (TBD), etc. have not yet been enforced.
- `transparent` function optimization has not yet been honored.

These properties are addressed by subsequent guaranteed optimization and diagnostic passes which are always run against the raw SIL.

12.2.2 Guaranteed Optimization and Diagnostic Passes

After SILGen, a deterministic sequence of optimization passes is run over the raw SIL. We do not want the diagnostics produced by the compiler to change as the compiler evolves, so these passes are intended to be simple and predictable.

- **Mandatory inlining** inlines calls to "transparent" functions.
- **Memory promotion** is implemented as two optimization phases, the first of which performs capture analysis to promote `alloc_box` instructions to `alloc_stack`, and the second of which promotes non-address-exposed `alloc_stack` instructions to SSA registers.
- **Constant propagation** folds constant expressions and propagates the constant values. If an arithmetic overflow occurs during the constant expression computation, a diagnostic is issued.
- **Return analysis** verifies that each function returns a value on every code path and doesn't "fall of the end" of its definition, which is an error. It also issues an error when a `noreturn` function returns.
- **Critical edge splitting** splits all critical edges from terminators that don't support arbitrary basic block arguments (all non cond_branch terminators).

If all diagnostic passes succeed, the final result is the *canonical SIL* for the program.

TODO:

- Generic specialization
- Basic ARC optimization for acceptable performance at -Onone.

12.2.3 General Optimization Passes

SIL captures language-specific type information, making it possible to perform high-level optimizations that are difficult to perform on LLVM IR.

- **Generic Specialization** analyzes specialized calls to generic functions and generates new specialized version of the functions. Then it rewrites all specialized usages of the gener ic to a direct call of the appropriate specialized function.
- **Witness and VTable Devirtualization** for a given type looks up the associated method from a class's vtable or a types witness table and replaces the indirect virtual call with a call to the mapped function.
- **Performance Inlining**
- **Reference Counting Optimizations**
- **Memory Promotion/Optimizations**
- **High-level domain specific optimizations** The swift compiler implements high-level optimizations on basic Swift containers such as Array or String. Domain specific optimizations require a defined interface between the standard library and the optimizer. More details can be found here: HighLevelSILOptimizations

12.3 Syntax

SIL is reliant on Swift's type system and declarations, so SIL syntax is an extension of Swift's. A `.sil` file is a Swift source file with added SIL definitions. The Swift source is parsed only for its declarations; Swift `func` bodies (except

for nested declarations) and top-level code are ignored by the SIL parser. In a `.sil` file, there are no implicit imports; the `swift` and/or `Builtin` standard modules must be imported explicitly if used.

Here is an example of a `.sil` file:

```
sil_stage canonical

import Swift

// Define types used by the SIL function.

struct Point {
  var x : Double
  var y : Double
}

class Button {
  func onClick()
  func onMouseDown()
  func onMouseUp()
}

// Declare a Swift function. The body is ignored by SIL.
func taxicabNorm(a:Point) -> Double {
  return a.x + a.y
}

// Define a SIL function.
// The name @_T5norms11taxicabNormfT1aV5norms5Point_Sd is the mangled name
// of the taxicabNorm Swift function.
sil @_T5norms11taxicabNormfT1aV5norms5Point_Sd : $(Point) -> Double {
bb0(%0 : $Point):
  // func Swift.+(Double, Double) -> Double
  %1 = function_ref @_Tsoi1pfTSdSd_Sd
  %2 = struct_extract %0 : $Point, #Point.x
  %3 = struct_extract %0 : $Point, #Point.y
  %4 = apply %1(%2, %3) : $(Double, Double) -> Double
  %5 = return %4 : Double
}

// Define a SIL vtable. This matches dynamically-dispatched method
// identifiers to their implementations for a known static class type.
sil_vtable Button {
  #Button.onClick!1: @_TC5norms6Button7onClickfS0_FT_T_
  #Button.onMouseDown!1: @_TC5norms6Button11onMouseDownfS0_FT_T_
  #Button.onMouseUp!1: @_TC5norms6Button9onMouseUpfS0_FT_T_
}
```

12.3.1 SIL Stage

```
decl ::= sil-stage-decl
sil-stage-decl ::= 'sil_stage' sil-stage

sil-stage ::= 'raw'
sil-stage ::= 'canonical'
```

There are different invariants on SIL depending on what stage of processing has been applied to it.

- **Raw SIL** is the form produced by SILGen that has not been run through guaranteed optimizations or diagnostic passes. Raw SIL may not have a fully-constructed SSA graph. It may contain dataflow errors. Some instructions may be represented in non-canonical forms, such as `assign` and `destroy_addr` for non-address-only values. Raw SIL should not be used for native code generation or distribution.
- **Canonical SIL** is SIL as it exists after guaranteed optimizations and diagnostics. Dataflow errors must be eliminated, and certain instructions must be canonicalized to simpler forms. Performance optimization and native code generation are derived from this form, and a module can be distributed containing SIL in this (or later) forms.

SIL files declare the processing stage of the included SIL with one of the declarations `sil_stage raw` or `sil_stage canonical` at top level. Only one such declaration may appear in a file.

12.3.2 SIL Types

```
sil-type ::= '$' '*'? generic-parameter-list? type
```

SIL types are introduced with the `$` sigil. SIL's type system is closely related to Swift's, and so the type after the `$` is parsed largely according to Swift's type grammar.

Type Lowering

A *formal type* is the type of a value in Swift, such as an expression result. Swift's formal type system intentionally abstracts over a large number of representational issues like ownership transfer conventions and directness of arguments. However, SIL aims to represent most such implementation details, and so these differences deserve to be reflected in the SIL type system. *Type lowering* is the process of turning a formal type into its *lowered type*.

It is important to be aware that the lowered type of a declaration need not be the lowered type of the formal type of that declaration. For example, the lowered type of a declaration reference:

- will usually be thin,
- will frequently be uncurried,
- may have a non-Swift calling convention,
- may use bridged types in its interface, and
- may use ownership conventions that differ from Swift's default conventions.

Abstraction Difference

Generic functions working with values of unconstrained type must generally work with them indirectly, e.g. by allocating sufficient memory for them and then passing around pointers to that memory. Consider a generic function like this:

```
func generateArray<T>(n : Int, generator : () -> T) -> T[]
```

The function `generator` will be expected to store its result indirectly into an address passed in an implicit parameter. There's really just no reasonable alternative when working with a value of arbitrary type:

- We don't want to generate a different copy of `generateArray` for every type `T`.
- We don't want to give every type in the language a common representation.
- We don't want to dynamically construct a call to `generator` depending on the type `T`.

But we also don't want the existence of the generic system to force inefficiencies on non-generic code. For example, we'd like a function of type `() -> Int` to be able to return its result directly; and yet, `() -> Int` is a valid substitution of `() -> T`, and a caller of `generateArray<Int>` should be able to pass an arbitrary `() -> Int` in as the generator.

Therefore, the representation of a formal type in a generic context may differ from the representation of a substitution of that formal type. We call such differences *abstraction differences*.

SIL's type system is designed to make abstraction differences always result in differences between SIL types. The goal is that a properly- abstracted value should be correctly usable at any level of substitution.

In order to achieve this, the formal type of a generic entity should always be lowered using the abstraction pattern of its unsubstituted formal type. For example, consider the following generic type:

```
struct Generator<T> {
  var fn : () -> T
}
var intGen : Generator<Int>
```

`intGen.fn` has the substituted formal type `() -> Int`, which would normally lower to the type `@callee_owned () -> Int`, i.e. returning its result directly. But if that type is properly lowered with the pattern of its unsubstituted type `() -> T`, it becomes `@callee_owned (@out Int) -> ()`.

When a type is lowered using the abstraction pattern of an unrestricted type, it is lowered as if the pattern were replaced with a type sharing the same structure but replacing all materializable types with fresh type variables.

For example, if `g` has type `Generator<(Int,Int) -> Float>`, `g.fn` is lowered using the pattern `() -> T`, which eventually causes `(Int,Int) -> Float` to be lowered using the pattern `T`, which is the same as lowering it with the pattern `U -> V`; the result is that `g.fn` has the following lowered type:

```
@callee_owned () -> @owned @callee_owned (@out Float, @in (Int,Int)) -> ()``.
```

As another example, suppose that `h` has type `Generator<(Int, @inout Int) -> Float>`. Neither `(Int, @inout Int)` nor `@inout Int` are potential results of substitution because they aren't materializable, so `h.fn` has the following lowered type:

```
@callee_owned () -> @owned @callee_owned (@out Float, @in Int, @inout Int)
```

This system has the property that abstraction patterns are preserved through repeated substitutions. That is, you can consider a lowered type to encode an abstraction pattern; lowering `T` by `R` is equivalent to lowering `T` by (`S` lowered by `R`).

SILGen has procedures for converting values between abstraction patterns.

At present, only function and tuple types are changed by abstraction differences.

Legal SIL Types

The type of a value in SIL shall be:

- a loadable legal SIL type, `$T`,
- the address of a legal SIL type, `$*T`, or
- the address of local storage of a legal SIL type, `$*@local_storage T`.

A type `T` is a *legal SIL type* if:

- it is a function type which satisfies the constraints (below) on function types in SIL,
- it is a tuple type whose element types are legal SIL types,

- it is a legal Swift type that is not a function, tuple, or l-value type, or
- it is a `@box` containing a legal SIL type.

Note that types in other recursive positions in the type grammar are still formal types. For example, the instance type of a metatype or the type arguments of a generic type are still formal Swift types, not lowered SIL types.

Address Types

The *address of T* `$*T` is a pointer to memory containing a value of any reference or value type `$T`. This can be an internal pointer into a data structure. Addresses of loadable types can be loaded and stored to access values of those types.

Addresses of address-only types (see below) can only be used with instructions that manipulate their operands indirectly by address, such as `copy_addr` or `destroy_addr`, or as arguments to functions. It is illegal to have a value of type `$T` if `T` is address-only.

Addresses are not reference-counted pointers like class values are. They cannot be retained or released.

Address types are not *first-class*: they cannot appear in recursive positions in type expressions. For example, the type `$**T` is not a legal type.

The address of an address cannot be directly taken. `$**T` is not a representable type. Values of address type thus cannot be allocated, loaded, or stored (though addresses can of course be loaded from and stored to).

Addresses can be passed as arguments to functions if the corresponding parameter is indirect. They cannot be returned.

Local Storage Types

The *address of local storage for T* `$*@local_storage T` is a handle to a stack allocation of a variable of type `$T`.

For many types, the handle for a stack allocation is simply the allocated address itself. However, if a type is runtime-sized, the compiler must emit code to potentially dynamically allocate memory. SIL abstracts over such differences by using values of local-storage type as the first result of `alloc_stack` and the operand of `dealloc_stack`.

Local-storage address types are not *first-class* in the same sense that address types are not first-class.

Box Types

Captured local variables and the payloads of `indirect` value types are stored on the heap. The type `@box T` is a reference-counted type that references a box containing a mutable value of type `T`. Boxes always use Swift-native reference counting, so they can be queried for uniqueness and cast to the `Builtin.NativeObject` type.

Function Types

Function types in SIL are different from function types in Swift in a number of ways:

- A SIL function type may be generic. For example, accessing a generic function with `function_ref` will give a value of generic function type.
- A SIL function type declares its conventional treatment of its context value:
  - If it is `@thin`, the function requires no context value.
  - If it is `@callee_owned`, the context value is treated as an owned direct parameter.
  - If it is `@callee_guaranteed`, the context value is treated as a guaranteed direct parameter.
  - Otherwise, the context value is treated as an unowned direct parameter.

- A SIL function type declares the conventions for its parameters, including any implicit out-parameters. The parameters are written as an unlabelled tuple; the elements of that tuple must be legal SIL types, optionally decorated with one of the following convention attributes.

  The value of an indirect parameter has type `*T`; the value of a direct parameter has type `T`.

  - An `@in` parameter is indirect. The address must be of an initialized object; the function is responsible for destroying the value held there.
  - An `@inout` parameter is indirect. The address must be of an initialized object, and the function must leave an initialized object there upon exit.
  - An `@out` parameter is indirect. The address must be of an uninitialized object; the function is responsible for initializing a value there. If there is an `@out` parameter, it must be the first parameter, and the direct result must be `()`.
  - An `@owned` parameter is an owned direct parameter.
  - A `@guaranteed` parameter is a guaranteed direct parameter.
  - An `@in_guaranteed` parameter is indirect. The address must be of an initialized object; both the caller and callee promise not to mutate the pointee, allowing the callee to read it.
  - Otherwise, the parameter is an unowned direct parameter.
- A SIL function type declares the convention for its direct result. The result must be a legal SIL type.
  - An `@owned` result is an owned direct result.
  - An `@autoreleased` result is an autoreleased direct result.
  - Otherwise, the parameter is an unowned direct result.

A direct parameter or result of trivial type must always be unowned.

An owned direct parameter or result is transferred to the recipient, which becomes responsible for destroying the value. This means that the value is passed at +1.

An unowned direct parameter or result is instantaneously valid at the point of transfer. The recipient does not need to worry about race conditions immediately destroying the value, but should copy it (e.g. by `strong_retain`ing an object pointer) if the value will be needed sooner rather than later.

A guaranteed direct parameter is like an unowned direct parameter value, except that it is guaranteed by the caller to remain valid throughout the execution of the call. This means that any `strong_retain`, `strong_release` pairs in the callee on the argument can be eliminated.

An autoreleased direct result must have a type with a retainable pointer representation. It may have been autoreleased, and the caller should take action to reclaim that autorelease with `strong_retain_autoreleased`.

- The @noescape declaration attribute on Swift parameters (which is valid only on parameters of function type, and is implied by the @autoclosure attribute) is turned into a @noescape type attribute on SIL arguments. @noescape indicates that the lifetime of the closure parameter will not be extended by the callee (e.g. the pointer will not be stored in a global variable). It corresponds to the LLVM "nocapture" attribute in terms of semantics (but is limited to only work with parameters of function type in Swift).
- SIL function types may provide an optional error result, written by placing `@error` on a result. An error result is always implicitly `@owned`. Only functions with a native calling convention may have an error result.

  A function with an error result cannot be called with `apply`. It must be called with `try_apply`. There is one exception to this rule: a function with an error result can be called with `apply [nothrow]` if the compiler can prove that the function does not actually throw.

  `return` produces a normal result of the function. To return an error result, use `throw`.

  Type lowering lowers the `throws` annotation on formal function types into more concrete error propagation:

  - For native Swift functions, `throws` is turned into an error result.
  - For non-native Swift functions, `throws` is turned in an explicit error-handling mechanism based on the imported API. The importer only imports non-native methods and types as `throws` when it is possible to do this automatically.

Properties of Types

SIL classifies types into additional subgroups based on ABI stability and generic constraints:

- *Loadable types* are types with a fully exposed concrete representation:
  - Reference types
  - Builtin value types
  - Fragile struct types in which all element types are loadable
  - Tuple types in which all element types are loadable
  - Class protocol types
  - Archetypes constrained by a class protocol

  A *loadable aggregate type* is a tuple or struct type that is loadable.

  A *trivial type* is a loadable type with trivial value semantics. Values of trivial type can be loaded and stored without any retain or release operations and do not need to be destroyed.
- *Runtime-sized types* are restricted value types for which the compiler does not know the size of the type statically:
  - Resilient value types
  - Fragile struct or tuple types that contain resilient types as elements at any depth
  - Archetypes not constrained by a class protocol
- *Address-only types* are restricted value types which cannot be loaded or otherwise worked with as SSA values:
  - Runtime-sized types
  - Non-class protocol types
  - @weak types

  Values of address-only type ("address-only values") must reside in memory and can only be referenced in SIL by address. Addresses of address-only values cannot be loaded from or stored to. SIL provides special instructions for indirectly manipulating address-only values, such as `copy_addr` and `destroy_addr`.

Some additional meaningful categories of type:

- A *heap object reference* type is a type whose representation consists of a single strong-reference-counted pointer. This includes all class types, the `Builtin.ObjectPointer` and `Builtin.ObjCPointer` types, and archetypes that conform to one or more class protocols.
- A *reference type* is more general in that its low-level representation may include additional global pointers alongside a strong-reference-counted pointer. This includes all heap object reference types and adds thick function types and protocol/protocol composition types that conform to one or more class protocols. All reference types can be `retain`-ed and `release`-d. Reference types also have *ownership semantics* for their referenced heap object; see *Reference Counting* below.
- A type with *retainable pointer representation* is guaranteed to be compatible (in the C sense) with the Objective-C `id` type. The value at runtime may be `nil`. This includes classes, class metatypes, block functions, and class-bounded existentials with only Objective-C-compatible protocol constraints, as well as one level of `Optional`

or `ImplicitlyUnwrappedOptional` applied to any of the above. Types with retainable pointer representation can be returned via the `@autoreleased` return convention.

SILGen does not always map Swift function types one-to-one to SIL function types. Function types are transformed in order to encode additional attributes:

- The **convention** of the function, indicated by the

  `@convention(`*convention*`)`

  attribute. This is similar to the language-level `@convention` attribute, though SIL extends the set of supported conventions with additional distinctions not exposed at the language level:

  - `@convention(thin)` indicates a "thin" function reference, which uses the Swift calling convention with no special "self" or "context" parameters.
  - `@convention(thick)` indicates a "thick" function reference, which uses the Swift calling convention and carries a reference-counted context object used to represent captures or other state required by the function.
  - `@convention(block)` indicates an Objective-C compatible block reference. The function value is represented as a reference to the block object, which is an `id`-compatible Objective-C object that embeds its invocation function within the object. The invocation function uses the C calling convention.
  - `@convention(c)` indicates a C function reference. The function value carries no context and uses the C calling convention.
  - `@convention(objc_method)` indicates an Objective-C method implementation. The function uses the C calling convention, with the SIL-level `self` parameter (by SIL convention mapped to the final formal parameter) mapped to the `self` and `_cmd` arguments of the implementation.
  - `@convention(method)` indicates a Swift instance method implementation. The function uses the Swift calling convention, using the special `self` parameter.
  - `@convention(witness_method)` indicates a Swift protocol method implementation. The function's polymorphic convention is emitted in such a way as to guarantee that it is polymorphic across all possible implementors of the protocol.

- The **fully uncurried representation** of the function type, with all of the curried argument clauses flattened into a single argument clause. For instance, a curried function `func foo(x:A)(y:B) -> C` might be emitted as a function of type `((y:B), (x:A)) -> C`. The exact representation depends on the function's *calling convention*, which determines the exact ordering of currying clauses. Methods are treated as a form of curried function.

Layout Compatible Types

(This section applies only to Swift 1.0 and will hopefully be obviated in future releases.)

SIL tries to be ignorant of the details of type layout, and low-level bit-banging operations such as pointer casts are generally undefined. However, as a concession to implementation convenience, some types are allowed to be considered **layout compatible**. Type `T` is *layout compatible* with type `U` iff:

- an address of type `$*U` can be cast by `address_to_pointer`/`pointer_to_address` to `$*T` and a valid value of type `T` can be loaded out (or indirectly used, if `T` is address- only),
- if `T` is a nontrivial type, then `retain_value`/`release_value` of the loaded `T` value is equivalent to `retain_value`/`release_value` of the original `U` value.

This is not always a commutative relationship; `T` can be layout-compatible with `U` whereas `U` is not layout-compatible with `T`. If the layout compatible relationship does extend both ways, `T` and `U` are **commutatively layout compatible**.

It is however always transitive; if `T` is layout-compatible with `U` and `U` is layout-compatible with `V`, then `T` is layout-compatible with `V`. All types are layout-compatible with themselves.

The following types are considered layout-compatible:

- `Builtin.RawPointer` is commutatively layout compatible with all heap object reference types, and `Optional` of heap object reference types. (Note that `RawPointer` is a trivial type, so does not have ownership semantics.)
- `Builtin.RawPointer` is commutatively layout compatible with `Builtin.Word`.
- Structs containing a single stored property are commutatively layout compatible with the type of that property.
- A heap object reference is commutatively layout compatible with any type that can correctly reference the heap object. For instance, given a class `B` and a derived class `D` inheriting from `B`, a value of type `B` referencing an instance of type `D` is layout compatible with both `B` and `D`, as well as `Builtin.NativeObject` and `Builtin.UnknownObject`. It is not layout compatible with an unrelated class type `E`.
- For payloaded enums, the payload type of the first payloaded case is layout-compatible with the enum (*not* commutatively).

12.3.3 Values and Operands

```
sil-identifier ::= [A-Za-z_0-9]+
sil-value-name ::= '%' sil-identifier
sil-value ::= sil-value-name ('#' [0-9]+)?
sil-value ::= 'undef'
sil-operand ::= sil-value ':' sil-type
```

SIL values are introduced with the `%` sigil and named by an alphanumeric identifier, which references the instruction or basic block argument that produces the value. SIL values may also refer to the keyword 'undef', which is a value of undefined contents. In SIL, a single instruction may produce multiple values. Operands that refer to multiple-value instructions choose the value by following the `%name` with `#` and the index of the value. For example:

```
// alloc_box produces two values--the refcounted pointer %box#0, and the
// value address %box#1
%box = alloc_box $Int64
// Refer to the refcounted pointer
strong_retain %box#0 : $@box Int64
// Refer to the address
store %value to %box#1 : $*Int64
```

Unlike LLVM IR, SIL instructions that take value operands *only* accept value operands. References to literal constants, functions, global variables, or other entities require specialized instructions such as `integer_literal`, `function_ref`, `global_addr`, etc.

12.3.4 Functions

```
decl ::= sil-function
sil-function ::= 'sil' sil-linkage? sil-function-name ':' sil-type
                 '{' sil-basic-block+ '}'
sil-function-name ::= '@' [A-Za-z_0-9]+
```

SIL functions are defined with the `sil` keyword. SIL function names are introduced with the `@` sigil and named by an alphanumeric identifier. This name will become the LLVM IR name for the function, and is usually the mangled

name of the originating Swift declaration. The `sil` syntax declares the function's name and SIL type, and defines the body of the function inside braces. The declared type must be a function type, which may be generic.

12.3.5 Basic Blocks

```
sil-basic-block ::= sil-label sil-instruction-def* sil-terminator
sil-label ::= sil-identifier ('(' sil-argument (',' sil-argument)* ')')? ':'
sil-argument ::= sil-value-name ':' sil-type

sil-instruction-def ::= (sil-value-name '=')? sil-instruction
```

A function body consists of one or more basic blocks that correspond to the nodes of the function's control flow graph. Each basic block contains one or more instructions and ends with a terminator instruction. The function's entry point is always the first basic block in its body.

In SIL, basic blocks take arguments, which are used as an alternative to LLVM's phi nodes. Basic block arguments are bound by the branch from the predecessor block:

```
sil @iif : $(Builtin.Int1, Builtin.Int64, Builtin.Int64) -> Builtin.Int64 {
bb0(%cond : $Builtin.Int1, %ifTrue : $Builtin.Int64, %ifFalse : $Builtin.Int64):
  cond_br %cond : $Builtin.Int1, then, else
then:
  br finish(%ifTrue : $Builtin.Int64)
else:
  br finish(%ifFalse : $Builtin.Int64)
finish(%result : $Builtin.Int64):
  return %result : $Builtin.Int64
}
```

Arguments to the entry point basic block, which has no predecessor, are bound by the function's caller:

```
sil @foo : $(Int) -> Int {
bb0(%x : $Int):
  %1 = return %x : $Int
}

sil @bar : $(Int, Int) -> () {
bb0(%x : $Int, %y : $Int):
  %foo = function_ref @foo
  %1 = apply %foo(%x) : $(Int) -> Int
  %2 = apply %foo(%y) : $(Int) -> Int
  %3 = tuple ()
  %4 = return %3 : $()
}
```

12.3.6 Declaration References

```
sil-decl-ref ::= '#' sil-identifier ('.' sil-identifier)* sil-decl-subref?
sil-decl-subref ::= '!' sil-decl-subref-part ('.' sil-decl-uncurry-level)? ('.' sil-
↪decl-lang)?
sil-decl-subref ::= '!' sil-decl-uncurry-level ('.' sil-decl-lang)?
sil-decl-subref ::= '!' sil-decl-lang
sil-decl-subref-part ::= 'getter'
sil-decl-subref-part ::= 'setter'
sil-decl-subref-part ::= 'allocator'
```

```
sil-decl-subref-part ::= 'initializer'
sil-decl-subref-part ::= 'enumelt'
sil-decl-subref-part ::= 'destroyer'
sil-decl-subref-part ::= 'deallocator'
sil-decl-subref-part ::= 'globalaccessor'
sil-decl-subref-part ::= 'ivardestroyer'
sil-decl-subref-part ::= 'ivarinitializer'
sil-decl-subref-part ::= 'defaultarg' '.' [0-9]+
sil-decl-uncurry-level ::= [0-9]+
sil-decl-lang ::= 'foreign'
```

Some SIL instructions need to reference Swift declarations directly. These references are introduced with the `#` sigil followed by the fully qualified name of the Swift declaration. Some Swift declarations are decomposed into multiple entities at the SIL level. These are distinguished by following the qualified name with `!` and one or more `.`-separated component entity discriminators:

- `getter`: the getter function for a `var` declaration
- `setter`: the setter function for a `var` declaration
- `allocator`: a `struct` or `enum` constructor, or a `class`'s *allocating constructor*
- `initializer`: a `class`'s *initializing constructor*
- `enumelt`: a member of a `enum` type.
- `destroyer`: a class's destroying destructor
- `deallocator`: a class's deallocating destructor
- `globalaccessor`: the addressor function for a global variable
- `ivardestroyer`: a class's ivar destroyer
- `ivarinitializer`: a class's ivar initializer
- `defaultarg.`*n*: the default argument-generating function for the *n*-th argument of a Swift `func`
- `foreign`: a specific entry point for C/objective-C interoperability

Methods and curried function definitions in Swift also have multiple "uncurry levels" in SIL, representing the function at each possible partial application level. For a curried function declaration:

```
// Module example
func foo(x:A)(y:B)(z:C) -> D
```

The declaration references and types for the different uncurry levels are as follows:

```
#example.foo!0 : $@thin (x:A) -> (y:B) -> (z:C) -> D
#example.foo!1 : $@thin ((y:B), (x:A)) -> (z:C) -> D
#example.foo!2 : $@thin ((z:C), (y:B), (x:A)) -> D
```

The deepest uncurry level is referred to as the **natural uncurry level**. In this specific example, the reference at the natural uncurry level is `#example.foo!2`. Note that the uncurried argument clauses are composed right-to-left, as specified in the *calling convention*. For uncurry levels less than the uncurry level, the entry point itself is `@thin` but returns a thick function value carrying the partially applied arguments for its context.

Dynamic dispatch instructions such as `class method` require their method declaration reference to be uncurried to at least uncurry level 1 (which applies both the "self" argument and the method arguments), because uncurry level zero represents the application of the method to its "self" argument, as in `foo.method`, which is where the dynamic dispatch semantically occurs in Swift.

12.3.7 Linkage

```
sil-linkage ::= 'public'
sil-linkage ::= 'hidden'
sil-linkage ::= 'shared'
sil-linkage ::= 'private'
sil-linkage ::= 'public_external'
sil-linkage ::= 'hidden_external'
```

A linkage specifier controls the situations in which two objects in different SIL modules are *linked*, i.e. treated as the same object.

A linkage is *external* if it ends with the suffix `external`. An object must be a definition if its linkage is not external.

All functions, global variables, and witness tables have linkage. The default linkage of a definition is `public`. The default linkage of a declaration is `public_external`. (These may eventually change to `hidden` and `hidden_external`, respectively.)

On a global variable, an external linkage is what indicates that the variable is not a definition. A variable lacking an explicit linkage specifier is presumed a definition (and thus gets the default linkage for definitions, `public`.)

Definition of the *linked* relation

Two objects are linked if they have the same name and are mutually visible:

- An object with `public` or `public_external` linkage is always visible.
- An object with `hidden`, `hidden_external`, or `shared` linkage is visible only to objects in the same Swift module.
- An object with `private` linkage is visible only to objects in the same SIL module.

Note that the *linked* relationship is an equivalence relation: it is reflexive, symmetric, and transitive.

Requirements on linked objects

If two objects are linked, they must have the same type.

If two objects are linked, they must have the same linkage, except:

- A `public` object may be linked to a `public_external` object.
- A `hidden` object may be linked to a `hidden_external` object.

If two objects are linked, at most one may be a definition, unless:

- both objects have `shared` linkage or
- at least one of the objects has an external linkage.

If two objects are linked, and both are definitions, then the definitions must be semantically equivalent. This equivalence may exist only on the level of user-visible semantics of well-defined code; it should not be taken to guarantee that the linked definitions are exactly operationally equivalent. For example, one definition of a function might copy a value out of an address parameter, while another may have had an analysis applied to prove that said value is not needed.

If an object has any uses, then it must be linked to a definition with non-external linkage.

Summary

- `public` definitions are unique and visible everywhere in the program. In LLVM IR, they will be emitted with `external` linkage and `default` visibility.
- `hidden` definitions are unique and visible only within the current Swift module. In LLVM IR, they will be emitted with `external` linkage and `hidden` visibility.
- `private` definitions are unique and visible only within the current SIL module. In LLVM IR, they will be emitted with `private` linkage.
- `shared` definitions are visible only within the current Swift module. They can be linked only with other `shared` definitions, which must be equivalent; therefore, they only need to be emitted if actually used. In LLVM IR, they will be emitted with `linkonce_odr` linkage and `hidden` visibility.
- `public_external` and `hidden_external` objects always have visible definitions somewhere else. If this object nonetheless has a definition, it's only for the benefit of optimization or analysis. In LLVM IR, declarations will have `external` linkage and definitions (if actually emitted as definitions) will have `available_externally` linkage.

12.3.8 VTables

```
decl ::= sil-vtable
sil-vtable ::= 'sil_vtable' identifier '{' sil-vtable-entry* '}'

sil-vtable-entry ::= sil-decl-ref ':' sil-function-name
```

SIL represents dynamic dispatch for class methods using the *class_method*, *super_method*, and *dynamic_method* instructions. The potential destinations for these dispatch operations are tracked in `sil_vtable` declarations for every class type. The declaration contains a mapping from every method of the class (including those inherited from its base class) to the SIL function that implements the method for that class:

```
class A {
  func foo()
  func bar()
  func bas()
}

sil @A_foo : $@thin (@owned A) -> ()
sil @A_bar : $@thin (@owned A) -> ()
sil @A_bas : $@thin (@owned A) -> ()

sil_vtable A {
  #A.foo!1: @A_foo
  #A.bar!1: @A_bar
  #A.bas!1: @A_bas
}

class B : A {
  func bar()
}

sil @B_bar : $@thin (@owned B) -> ()

sil_vtable B {
  #A.foo!1: @A_foo
  #A.bar!1: @B_bar
```

```
  #A.bas!1: @A_bas
}

class C : B {
  func bas()
}

sil @C_bas : $@thin (@owned C) -> ()

sil_vtable C {
  #A.foo!1: @A_foo
  #A.bar!1: @B_bar
  #A.bas!1: @C_bas
}
```

Note that the declaration reference in the vtable is to the least-derived method visible through that class (in the example above, `B`'s vtable references `A.bar` and not `B.bar`, and `C`'s vtable references `A.bas` and not `C.bas`). The Swift AST maintains override relationships between declarations that can be used to look up overridden methods in the SIL vtable for a derived class (such as `C.bas` in `C`'s vtable).

12.3.9 Witness Tables

```
decl ::= sil-witness-table
sil-witness-table ::= 'sil_witness_table' sil-linkage?
                      normal-protocol-conformance '{' sil-witness-entry* '}'
```

SIL encodes the information needed for dynamic dispatch of generic types into witness tables. This information is used to produce runtime dispatch tables when generating binary code. It can also be used by SIL optimizations to specialize generic functions. A witness table is emitted for every declared explicit conformance. Generic types share one generic witness table for all of their instances. Derived classes inherit the witness tables of their base class.

```
protocol-conformance ::= normal-protocol-conformance
protocol-conformance ::= 'inherit' '(' protocol-conformance ')'
protocol-conformance ::= 'specialize' '<' substitution* '>'
                         '(' protocol-conformance ')'
protocol-conformance ::= 'dependent'
normal-protocol-conformance ::= identifier ':' identifier 'module' identifier
```

Witness tables are keyed by *protocol conformance*, which is a unique identifier for a concrete type's conformance to a protocol.

- A *normal protocol conformance* names a (potentially unbound generic) type, the protocol it conforms to, and the module in which the type or extension declaration that provides the conformance appears. These correspond 1:1 to protocol conformance declarations in the source code.
- If a derived class conforms to a protocol through inheritance from its base class, this is represented by an *inherited protocol conformance*, which simply references the protocol conformance for the base class.
- If an instance of a generic type conforms to a protocol, it does so with a *specialized conformance*, which provides the generic parameter bindings to the normal conformance, which should be for a generic type.

Witness tables are only directly associated with normal conformances. Inherited and specialized conformances indirectly reference the witness table of the underlying normal conformance.

```
sil-witness-entry ::= 'base_protocol' identifier ':' protocol-conformance
sil-witness-entry ::= 'method' sil-decl-ref ':' sil-function-name
sil-witness-entry ::= 'associated_type' identifier
```

```
sil-witness-entry ::= 'associated_type_protocol'
                      '(' identifier ':' identifier ')' ':' protocol-conformance
```

Witness tables consist of the following entries:

- *Base protocol entries* provide references to the protocol conformances that satisfy the witnessed protocols' inherited protocols.
- *Method entries* map a method requirement of the protocol to a SIL function that implements that method for the witness type. One method entry must exist for every required method of the witnessed protocol.
- *Associated type entries* map an associated type requirement of the protocol to the type that satisfies that requirement for the witness type. Note that the witness type is a source-level Swift type and not a SIL type. One associated type entry must exist for every required associated type of the witnessed protocol.
- *Associated type protocol entries* map a protocol requirement on an associated type to the protocol conformance that satisfies that requirement for the associated type.

12.3.10 Global Variables

```
decl ::= sil-global-variable
sil-global-variable ::= 'sil_global' sil-linkage identifier ':' sil-type
```

SIL representation of a global variable.

FIXME: to be written.

12.4 Dataflow Errors

Dataflow errors may exist in raw SIL. Swift's semantics defines these conditions as errors, so they must be diagnosed by diagnostic passes and must not exist in canonical SIL.

12.4.1 Definitive Initialization

Swift requires that all local variables be initialized before use. In constructors, all instance variables of a struct, enum, or class type must be initialized before the object is used and before the constructor is returned from.

12.4.2 Unreachable Control Flow

The `unreachable` terminator is emitted in raw SIL to mark incorrect control flow, such as a non-`Void` function failing to `return` a value, or a `switch` statement failing to cover all possible values of its subject. The guaranteed dead code elimination pass can eliminate truly unreachable basic blocks, or `unreachable` instructions may be dominated by applications of `@noreturn` functions. An `unreachable` instruction that survives guaranteed DCE and is not immediately preceded by a `@noreturn` application is a dataflow error.

12.5 Runtime Failure

Some operations, such as failed unconditional *checked conversions* or the `Builtin.trap` compiler builtin, cause a *runtime failure*, which unconditionally terminates the current actor. If it can be proven that a runtime failure will occur or did occur, runtime failures may be reordered so long as they remain well-ordered relative to operations external to

the actor or the program as a whole. For instance, with overflow checking on integer arithmetic enabled, a simple `for` loop that reads inputs in from one or more arrays and writes outputs to another array, all local to the current actor, may cause runtime failure in the update operations:

```
// Given unknown start and end values, this loop may overflow
for var i = unknownStartValue; i != unknownEndValue; ++i {
  ...
}
```

It is permitted to hoist the overflow check and associated runtime failure out of the loop itself and check the bounds of the loop prior to entering it, so long as the loop body has no observable effect outside of the current actor.

12.6 Undefined Behavior

Incorrect use of some operations is *undefined behavior*, such as invalid unchecked casts involving `Builtin.RawPointer` types, or use of compiler builtins that lower to LLVM instructions with undefined behavior at the LLVM level. A SIL program with undefined behavior is meaningless, much like undefined behavior in C, and has no predictable semantics. Undefined behavior should not be triggered by valid SIL emitted by a correct Swift program using a correct standard library, but cannot in all cases be diagnosed or verified at the SIL level.

12.7 Calling Convention

This section describes how Swift functions are emitted in SIL.

12.7.1 Swift Calling Convention @cc(swift)

The Swift calling convention is the one used by default for native Swift functions.

Tuples in the input type of the function are recursively destructured into separate arguments, both in the entry point basic block of the callee, and in the `apply` instructions used by callers:

```
func foo(x:Int, y:Int)

sil @foo : $(x:Int, y:Int) -> () {
entry(%x : $Int, %y : $Int):
  ...
}

func bar(x:Int, y:(Int, Int))

sil @bar : $(x:Int, y:(Int, Int)) -> () {
entry(%x : $Int, %y0 : $Int, %y1 : $Int):
  ...
}

func call_foo_and_bar() {
  foo(1, 2)
  bar(4, (5, 6))
}

sil @call_foo_and_bar : $() -> () {
entry:
  ...
```

```
  %foo = function_ref @foo : $(x:Int, y:Int) -> ()
  %foo_result = apply %foo(%1, %2) : $(x:Int, y:Int) -> ()
  ...
  %bar = function_ref @bar : $(x:Int, y:(Int, Int)) -> ()
  %bar_result = apply %bar(%4, %5, %6) : $(x:Int, y:(Int, Int)) -> ()
}
```

Calling a function with trivial value types as inputs and outputs simply passes the arguments by value. This Swift function:

```
func foo(x:Int, y:Float) -> UnicodeScalar

foo(x, y)
```

gets called in SIL as:

```
%foo = constant_ref $(Int, Float) -> UnicodeScalar, @foo
%z = apply %foo(%x, %y) : $(Int, Float) -> UnicodeScalar
```

Reference Counts

NOTE This section only is speaking in terms of rules of thumb. The actual behavior of arguments with respect to arguments is defined by the argument's convention attribute (e.g. `@owned`), not the calling convention itself.

Reference type arguments are passed in at +1 retain count and consumed by the callee. A reference type return value is returned at +1 and consumed by the caller. Value types with reference type components have their reference type components each retained and released the same way. This Swift function:

```
class A {}

func bar(x:A) -> (Int, A) { ... }

bar(x)
```

gets called in SIL as:

```
%bar = function_ref @bar : $(A) -> (Int, A)
strong_retain %x : $A
%z = apply %bar(%x) : $(A) -> (Int, A)
// ... use %z ...
%z_1 = tuple_extract %z : $(Int, A), 1
strong_release %z_1
```

When applying a thick function value as a callee, the function value is also consumed at +1 retain count.

Address-Only Types

For address-only arguments, the caller allocates a copy and passes the address of the copy to the callee. The callee takes ownership of the copy and is responsible for destroying or consuming the value, though the caller must still deallocate the memory. For address-only return values, the caller allocates an uninitialized buffer and passes its address as the first argument to the callee. The callee must initialize this buffer before returning. This Swift function:

```
 @API struct A {}

func bas(x:A, y:Int) -> A { return x }
```

```
var z = bas(x, y)
// ... use z ...
```

gets called in SIL as:

```
%bas = function_ref @bas : $(A, Int) -> A
%z = alloc_stack $A
%x_arg = alloc_stack $A
copy_addr %x to [initialize] %x_arg : $*A
apply %bas(%z, %x_arg, %y) : $(A, Int) -> A
dealloc_stack %x_arg : $*A // callee consumes %x.arg, caller deallocs
// ... use %z ...
destroy_addr %z : $*A
dealloc_stack stack %z : $*A
```

The implementation of `@bas` is then responsible for consuming `%x_arg` and initializing `%z`.

Tuple arguments are destructured regardless of the address-only-ness of the tuple type. The destructured fields are passed individually according to the above convention. This Swift function:

```
@API struct A {}

func zim(x:Int, y:A, (z:Int, w:(A, Int)))

zim(x, y, (z, w))
```

gets called in SIL as:

```
%zim = function_ref @zim : $(x:Int, y:A, (z:Int, w:(A, Int))) -> ()
%y_arg = alloc_stack $A
copy_addr %y to [initialize] %y_arg : $*A
%w_0_addr = element_addr %w : $*(A, Int), 0
%w_0_arg = alloc_stack $A
copy_addr %w_0_addr to [initialize] %w_0_arg : $*A
%w_1_addr = element_addr %w : $*(A, Int), 1
%w_1 = load %w_1_addr : $*Int
apply %zim(%x, %y_arg, %z, %w_0_arg, %w_1) : $(x:Int, y:A, (z:Int, w:(A, Int))) -> ()
dealloc_stack %w_0_arg
dealloc_stack %y_arg
```

Variadic Arguments

Variadic arguments and tuple elements are packaged into an array and passed as a single array argument. This Swift function:

```
func zang(x:Int, (y:Int, z:Int...), v:Int, w:Int...)

zang(x, (y, z0, z1), v, w0, w1, w2)
```

gets called in SIL as:

```
%zang = function_ref @zang : $(x:Int, (y:Int, z:Int...), v:Int, w:Int...) -> ()
%zs = <<make array from %z1, %z2>>
%ws = <<make array from %w0, %w1, %w2>>
apply %zang(%x, %y, %zs, %v, %ws)  : $(x:Int, (y:Int, z:Int...), v:Int, w:Int...) ->␣
↪()
```

Function Currying

Curried function definitions in Swift emit multiple SIL entry points, one for each "uncurry level" of the function. When a function is uncurried, its outermost argument clauses are combined into a tuple in right-to-left order. For the following declaration:

```
func curried(x:A)(y:B)(z:C)(w:D) -> Int {}
```

The types of the SIL entry points are as follows:

```
sil @curried_0 : $(x:A) -> (y:B) -> (z:C) -> (w:D) -> Int { ... }
sil @curried_1 : $((y:B), (x:A)) -> (z:C) -> (w:D) -> Int { ... }
sil @curried_2 : $((z:C), (y:B), (x:A)) -> (w:D) -> Int { ... }
sil @curried_3 : $((w:D), (z:C), (y:B), (x:A)) -> Int { ... }
```

@inout Arguments

`@inout` arguments are passed into the entry point by address. The callee does not take ownership of the referenced memory. The referenced memory must be initialized upon function entry and exit. If the `@inout` argument refers to a fragile physical variable, then the argument is the address of that variable. If the `@inout` argument refers to a logical property, then the argument is the address of a caller-owned writeback buffer. It is the caller's responsibility to initialize the buffer by storing the result of the property getter prior to calling the function and to write back to the property on return by loading from the buffer and invoking the setter with the final value. This Swift function:

```
func inout(x:@inout Int) {
  x = 1
}
```

gets lowered to SIL as:

```
sil @inout : $(@inout Int) -> () {
entry(%x : $*Int):
  %1 = integer_literal 1 : $Int
  store %1 to %x
  return
}
```

12.7.2 Swift Method Calling Convention @cc(method)

The method calling convention is currently identical to the freestanding function convention. Methods are considered to be curried functions, taking the "self" argument as their outer argument clause, and the method arguments as the inner argument clause(s). When uncurried, the "self" argument is thus passed last:

```
struct Foo {
  func method(x:Int) -> Int {}
}

sil @Foo_method_1 : $((x : Int), @inout Foo) -> Int { ... }
```

12.7.3 Witness Method Calling Convention @cc(witness_method)

The witness method calling convention is used by protocol witness methods in *witness tables*. It is identical to the `method` calling convention except that its handling of generic type parameters. For non-witness methods, the machine-level convention for passing type parameter metadata may be arbitrarily dependent on static aspects of the function signature, but because witnesses must be polymorphically dispatchable on their `Self` type, the `Self`-related metadata for a witness must be passed in a maximally abstracted manner.

12.7.4 C Calling Convention @cc(cdecl)

In Swift's C module importer, C types are always mapped to Swift types considered trivial by SIL. SIL does not concern itself with platform ABI requirements for indirect return, register vs. stack passing, etc.; C function arguments and returns in SIL are always by value regardless of the platform calling convention.

SIL (and therefore Swift) cannot currently invoke variadic C functions.

12.7.5 Objective-C Calling Convention @cc(objc_method)

Reference Counts

Objective-C methods use the same argument and return value ownership rules as ARC Objective-C. Selector families and the `ns_consumed`, `ns_returns_retained`, etc. attributes from imported Objective-C definitions are honored.

Applying a `@convention(block)` value does not consume the block.

Method Currying

In SIL, the "self" argument of an Objective-C method is uncurried to the last argument of the uncurried type, just like a native Swift method:

```
@objc class NSString {
  func stringByPaddingToLength(Int) withString(NSString) startingAtIndex(Int)
}

sil @NSString_stringByPaddingToLength_withString_startingAtIndex \
  : $((Int, NSString, Int), NSString)
```

That `self` is passed as the first argument at the IR level is abstracted away in SIL, as is the existence of the `_cmd` selector argument.

12.8 Type Based Alias Analysis

SIL supports two types of Type Based Alias Analysis (TBAA): Class TBAA and Typed Access TBAA.

12.8.1 Class TBAA

Class instances and other *heap object references* are pointers at the implementation level, but unlike SIL addresses, they are first class values and can be `capture`-d and alias. Swift, however, is memory-safe and statically typed, so aliasing of classes is constrained by the type system as follows:

- A `Builtin.NativeObject` may alias any native Swift heap object, including a Swift class instance, a box allocated by `alloc_box`, or a thick function's closure context. It may not alias natively Objective-C class instances.
- A `Builtin.UnknownObject` may alias any class instance, whether Swift or Objective-C, but may not alias non-class-instance heap objects.
- Two values of the same class type `$C` may alias. Two values of related class type `$B` and `$D`, where there is a subclass relationship between `$B` and `$D`, may alias. Two values of unrelated class types may not alias. This includes different instantiations of a generic class type, such as `$C<Int>` and `$C<Float>`, which currently may never alias.
- Without whole-program visibility, values of archetype or protocol type must be assumed to potentially alias any class instance. Even if it is locally apparent that a class does not conform to that protocol, another component may introduce a conformance by an extension. Similarly, a generic class instance, such as `$C<T>` for archetype `T`, must be assumed to potentially alias concrete instances of the generic type, such as `$C<Int>`, because `Int` is a potential substitution for `T`.

12.8.2 Typed Access TBAA

Define a *typed access* of an address or reference as one of the following:

- Any instruction that performs a typed read or write operation upon the memory at the given location (e.x. `load`, `store`).
- Any instruction that yields a typed offset of the pointer by performing a typed projection operation (e.x. `ref_element_addr`, `tuple_element_addr`).

It is undefined behavior to perform a typed access to an address or reference if the stored object or referent is not an allocated object of the relevant type.

This allows the optimizer to assume that two addresses cannot alias if there does not exist a substitution of archetypes that could cause one of the types to be the type of a subobject of the other. Additionally, this applies to the types of the values from which the addresses were derived, ignoring "blessed" alias-introducing operations such as `pointer_to_address`, the `bitcast` intrinsic, and the `inttoptr` intrinsic.

12.9 Value Dependence

In general, analyses can assume that independent values are independently assured of validity. For example, a class method may return a class reference:

```
bb0(%0 : $MyClass):
  %1 = class_method %0 : $MyClass, #MyClass.foo!1
  %2 = apply %1(%0) : $@cc(method) @thin (@guaranteed MyClass) -> @owned MyOtherClass
  // use of %2 goes here; no use of %1
  strong_release %2 : $MyOtherClass
  strong_release %1 : $MyClass
```

The optimizer is free to move the release of `%1` to immediately after the call here, because `%2` can be assumed to be an independently-managed value, and because Swift generally permits the reordering of destructors.

However, some instructions do create values that are intrinsically dependent on their operands. For example, the result of `ref_element_addr` will become a dangling pointer if the base is released too soon. This is captured by the concept of *value dependence*, and any transformation which can reorder of destruction of a value around another operation must remain conscious of it.

A value `%1` is said to be *value-dependent* on a value `%0` if:

- `%1` is the result and `%0` is the first operand of one of the following instructions:
  - `ref_element_addr`
  - `struct_element_addr`
  - `tuple_element_addr`
  - `unchecked_take_enum_data_addr`
  - `pointer_to_address`
  - `address_to_pointer`
  - `index_addr`
  - `index_raw_pointer`
  - possibly some other conversions
- `%1` is the result of `mark_dependence` and `%0` is either of the operands.
- `%1` is the value address of an allocation instruction of which `%0` is the local storage token or box reference.
- `%1` is the result of a `struct`, `tuple`, or `enum` instruction and `%0` is an operand.
- `%1` is the result of projecting out a subobject of `%0` with `tuple_extract`, `struct_extract`, `unchecked_enum_data`, `select_enum`, or `select_enum_addr`.
- `%1` is the result of `select_value` and `%0` is one of the cases.
- `%1` is a basic block parameter and `%0` is the corresponding argument from a branch to that block.
- `%1` is the result of a `load` from `%0`. However, the value dependence is cut after the first attempt to manage the value of `%1`, e.g. by retaining it.
- Transitivity: there exists a value `%2` which `%1` depends on and which depends on `%0`. However, transitivity does not apply to different subobjects of a struct, tuple, or enum.

Note, however, that an analysis is not required to track dependence through memory. Nor is it required to consider the possibility of dependence being established "behind the scenes" by opaque code, such as by a method returning an unsafe pointer to a class property. The dependence is required to be locally obvious in a function's SIL instructions. Precautions must be taken against this either by SIL generators (by using `mark_dependence` appropriately) or by the user (by using the appropriate intrinsics and attributes with unsafe language or library features).

Only certain types of SIL value can carry value-dependence:

- SIL address types
- unmanaged pointer types:
  - `@sil_unmanaged` types
  - `Builtin.RawPointer`
  - aggregates containing such a type, such as `UnsafePointer`, possibly recursively
- non-trivial types (but they can be independently managed)

This rule means that casting a pointer to an integer type breaks value-dependence. This restriction is necessary so that reading an `Int` from a class doesn't force the class to be kept around! A class holding an unsafe reference to an object must use some sort of unmanaged pointer type to do so.

This rule does not include generic or resilient value types which might contain unmanaged pointer types. Analyses are free to assume that e.g. a `copy_addr` of a generic or resilient value type yields an independently-managed value. The extension of value dependence to types containing obvious unmanaged pointer types is an affordance to make

the use of such types more convenient; it does not shift the ultimate responsibility for assuring the safety of unsafe language/library features away from the user.

12.10 Instruction Set

12.10.1 Allocation and Deallocation

These instructions allocate and deallocate memory.

alloc_stack

```
sil-instruction ::= 'alloc_stack' sil-type

%1 = alloc_stack $T
// %1#0 has type $*@local_storage T
// %1#1 has type $*T
```

Allocates uninitialized memory that is sufficiently aligned on the stack to contain a value of type `T`. The first result of the instruction is a local-storage handle suitable for passing to `dealloc_stack`. The second result of the instruction is the address of the allocated memory.

`alloc_stack` marks the start of the lifetime of the value; the allocation must be balanced with a `dealloc_stack` instruction to mark the end of its lifetime. All `alloc_stack` allocations must be deallocated prior to returning from a function. If a block has multiple predecessors, the stack height and order of allocations must be consistent coming from all predecessor blocks. `alloc_stack` allocations must be deallocated in last-in, first-out stack order.

The memory is not retainable. To allocate a retainable box for a value type, use `alloc_box`.

alloc_ref

```
sil-instruction ::= 'alloc_ref' ('[' 'objc' ']')? ('[' 'stack' ']')? sil-type

%1 = alloc_ref [stack] $T
// $T must be a reference type
// %1 has type $T
```

Allocates an object of reference type `T`. The object will be initialized with retain count 1; its state will be otherwise uninitialized. The optional `objc` attribute indicates that the object should be allocated using Objective-C's allocation methods (`+allocWithZone:`). The optional `stack` attribute indicates that the object can be allocated on the stack instead on the heap. In this case the instruction must have balanced with a `dealloc_ref [stack]` instruction to mark the end of the object's lifetime. Note that the `stack` attribute only specifies that stack allocation is possible. The final decision on stack allocation is done during llvm IR generation. This is because the decision also depends on the object size, which is not necessarily known at SIL level.

alloc_ref_dynamic

```
sil-instruction ::= 'alloc_ref_dynamic' ('[' 'objc' ']')? sil-operand ',' sil-type

%1 = alloc_ref_dynamic %0 : $@thick T.Type, $T
%1 = alloc_ref_dynamic [objc] %0 : $@objc_metatype T.Type, $T
```

```
// $T must be a class type
// %1 has type $T
```

Allocates an object of class type `T` or a subclass thereof. The dynamic type of the resulting object is specified via the metatype value `%0`. The object will be initialized with retain count 1; its state will be otherwise uninitialized. The optional `objc` attribute indicates that the object should be allocated using Objective-C's allocation methods (`+allocWithZone:`).

alloc_box

```
sil-instruction ::= 'alloc_box' sil-type

%1 = alloc_box $T
// %1 has two values:
//   %1#0 has type $@box T
//   %1#1 has type $*T
```

Allocates a reference-counted `@box` on the heap large enough to hold a value of type `T`, along with a retain count and any other metadata required by the runtime. The result of the instruction is a two-value operand; the first value is the reference-counted `@box` reference that owns the box, and the second value is the address of the value inside the box.

The box will be initialized with a retain count of 1; the storage will be uninitialized. The box owns the contained value, and releasing it to a retain count of zero destroys the contained value as if by `destroy_addr`. Releasing a box is undefined behavior if the box's value is uninitialized. To deallocate a box whose value has not been initialized, `dealloc_box` should be used.

alloc_value_buffer

```
sil-instruction ::= 'alloc_value_buffer' sil-type 'in' sil-operand

%1 = alloc_value_buffer $(Int, T) in %0 : $*Builtin.UnsafeValueBuffer
// The operand must have the exact type shown.
// The result has type $*(Int, T).
```

Given the address of an unallocated value buffer, allocate space in it for a value of the given type. This instruction has undefined behavior if the value buffer is currently allocated.

The type operand must be a lowered object type.

dealloc_stack

```
sil-instruction ::= 'dealloc_stack' sil-operand

dealloc_stack %0 : $*@local_storage T
// %0 must be of a local-storage $*@local_storage T type
```

Deallocates memory previously allocated by `alloc_stack`. The allocated value in memory must be uninitialized or destroyed prior to being deallocated. This instruction marks the end of the lifetime for the value created by the corresponding `alloc_stack` instruction. The operand must be the `@local_storage` of the shallowest live `alloc_stack` allocation preceding the deallocation. In other words, deallocations must be in last-in, first-out stack order.

dealloc_box

```
sil-instruction ::= 'dealloc_box' sil-operand

dealloc_box %0 : $@box T
```

Deallocates a box, bypassing the reference counting mechanism. The box variable must have a retain count of one. The boxed type must match the type passed to the corresponding `alloc_box` exactly, or else undefined behavior results.

This does not destroy the boxed value. The contents of the value must have been fully uninitialized or destroyed before `dealloc_box` is applied.

project_box

```
sil-instruction ::= 'project_box' sil-operand

%1 = project_box %0 : $@box T

// %1 has type $*T
```

Given a `@box T` reference, produces the address of the value inside the box.

dealloc_ref

```
sil-instruction ::= 'dealloc_ref' ('[' 'stack' ']')? sil-operand

dealloc_ref [stack] %0 : $T
// $T must be a class type
```

Deallocates an uninitialized class type instance, bypassing the reference counting mechanism.

The type of the operand must match the allocated type exactly, or else undefined behavior results.

The instance must have a retain count of one.

This does not destroy stored properties of the instance. The contents of stored properties must be fully uninitialized at the time `dealloc_ref` is applied.

The `stack` attribute indicates that the instruction is the balanced deallocation of its operand which must be a `alloc_ref [stack]`. In this case the instruction marks the end of the object's lifetime but has no other effect.

dealloc_partial_ref

```
sil-instruction ::= 'dealloc_partial_ref' sil-operand sil-metatype

dealloc_partial_ref %0 : $T, %1 : $U.Type
// $T must be a class type
// $T must be a subclass of U
```

Deallocates a partially-initialized class type instance, bypassing the reference counting mechanism.

The type of the operand must be a supertype of the allocated type, or else undefined behavior results.

The instance must have a retain count of one.

All stored properties in classes more derived than the given metatype value must be initialized, and all other stored properties must be uninitialized. The initialized stored properties are destroyed before deallocating the memory for the instance.

This does not destroy the reference type instance. The contents of the heap object must have been fully uninitialized or destroyed before `dealloc_ref` is applied.

dealloc_value_buffer

```
sil-instruction ::= 'dealloc_value_buffer' sil-type 'in' sil-operand

dealloc_value_buffer $(Int, T) in %0 : $*Builtin.UnsafeValueBuffer
// The operand must have the exact type shown.
```

Given the address of a value buffer, deallocate the storage in it. This instruction has undefined behavior if the value buffer is not currently allocated, or if it was allocated with a type other than the type operand.

The type operand must be a lowered object type.

project_value_buffer

```
sil-instruction ::= 'project_value_buffer' sil-type 'in' sil-operand

%1 = project_value_buffer $(Int, T) in %0 : $*Builtin.UnsafeValueBuffer
// The operand must have the exact type shown.
// The result has type $*(Int, T).
```

Given the address of a value buffer, return the address of the value storage in it. This instruction has undefined behavior if the value buffer is not currently allocated, or if it was allocated with a type other than the type operand.

The result is the same value as was originally returned by `alloc_value_buffer`.

The type operand must be a lowered object type.

12.10.2 Debug Information

Debug information is generally associated with allocations (alloc_stack or alloc_box) by having a Decl node attached to the allocation with a SILLocation. For declarations that have no allocation we have explicit instructions for doing this. This is used by 'let' declarations, which bind a value to a name and for var decls who are promoted into registers. The decl they refer to is attached to the instruction with a SILLocation.

debug_value

```
sil-instruction ::= debug_value sil-operand

debug_value %1 : $Int
```

This indicates that the value of a declaration with loadable type has changed value to the specified operand. The declaration in question is identified by the SILLocation attached to the debug_value instruction.

The operand must have loadable type.

debug_value_addr

```
sil-instruction ::= debug_value_addr sil-operand

debug_value_addr %7 : $*SomeProtocol
```

This indicates that the value of a declaration with address-only type has changed value to the specified operand. The declaration in question is identified by the SILLocation attached to the debug_value_addr instruction.

12.10.3 Accessing Memory

load

```
sil-instruction ::= 'load' sil-operand

%1 = load %0 : $*T
// %0 must be of a $*T address type for loadable type $T
// %1 will be of type $T
```

Loads the value at address `%0` from memory. `T` must be a loadable type. This does not affect the reference count, if any, of the loaded value; the value must be retained explicitly if necessary. It is undefined behavior to load from uninitialized memory or to load from an address that points to deallocated storage.

store

```
sil-instruction ::= 'store' sil-value 'to' sil-operand

store %0 to %1 : $*T
// $T must be a loadable type
```

Stores the value `%0` to memory at address `%1`. The type of %1 is `*T` and the type of `%0 is ``T`, which must be a loadable type. This will overwrite the memory at `%1`. If `%1` already references a value that requires `release` or other cleanup, that value must be loaded before being stored over and cleaned up. It is undefined behavior to store to an address that points to deallocated storage.

assign

```
sil-instruction ::= 'assign' sil-value 'to' sil-operand

assign %0 to %1 : $*T
// $T must be a loadable type
```

Represents an abstract assignment of the value `%0` to memory at address `%1` without specifying whether it is an initialization or a normal store. The type of %1 is `*T` and the type of `%0` is `T`, which must be a loadable type. This will overwrite the memory at `%1` and destroy the value currently held there.

The purpose of the `assign` instruction is to simplify the definitive initialization analysis on loadable variables by removing what would otherwise appear to be a load and use of the current value. It is produced by SILGen, which cannot know which assignments are meant to be initializations. If it is deemed to be an initialization, it can be replaced with a `store`; otherwise, it must be replaced with a sequence that also correctly destroys the current value.

This instruction is only valid in Raw SIL and is rewritten as appropriate by the definitive initialization pass.

mark_uninitialized

```
sil-instruction ::= 'mark_uninitialized' '[' mu_kind ']' sil-operand
mu_kind ::= 'var'
mu_kind ::= 'rootself'
mu_kind ::= 'derivedself'
mu_kind ::= 'derivedselfonly'
mu_kind ::= 'delegatingself'

%2 = mark_uninitialized [var] %1 : $*T
// $T must be an address
```

Indicates that a symbolic memory location is uninitialized, and must be explicitly initialized before it escapes or before the current function returns. This instruction returns its operands, and all accesses within the function must be performed against the return value of the mark_uninitialized instruction.

The kind of mark_uninitialized instruction specifies the type of data the mark_uninitialized instruction refers to:

- `var`: designates the start of a normal variable live range
- `rootself`: designates `self` in a struct, enum, or root class
- `derivedself`: designates `self` in a derived (non-root) class
- `derivedselfonly`: designates `self` in a derived (non-root) class whose stored properties have already been initialized
- `delegatingself`: designates `self` on a struct, enum, or class in a delegating constructor (one that calls self.init)

The purpose of the `mark_uninitialized` instruction is to enable definitive initialization analysis for global variables (when marked as 'globalvar') and instance variables (when marked as 'rootinit'), which need to be distinguished from simple allocations.

It is produced by SILGen, and is only valid in Raw SIL. It is rewritten as appropriate by the definitive initialization pass.

mark_function_escape

```
sil-instruction ::= 'mark_function_escape' sil-operand (',' sil-operand)

%2 = mark_function_escape %1 : $*T
```

Indicates that a function definition closes over a symbolic memory location. This instruction is variadic, and all of its operands must be addresses.

The purpose of the `mark_function_escape` instruction is to enable definitive initialization analysis for global variables and instance variables, which are not represented as box allocations.

It is produced by SILGen, and is only valid in Raw SIL. It is rewritten as appropriate by the definitive initialization pass.

copy_addr

```
sil-instruction ::= 'copy_addr' '[take]'? sil-value
                      'to' '[initialization]'? sil-operand
```

```
copy_addr [take] %0 to [initialization] %1 : $*T
// %0 and %1 must be of the same $*T address type
```

Loads the value at address `%0` from memory and assigns a copy of it back into memory at address `%1`. A bare `copy_addr` instruction when `T` is a non-trivial type:

```
copy_addr %0 to %1 : $*T
```

is equivalent to:

```
%new = load %0 : $*T        // Load the new value from the source
%old = load %1 : $*T        // Load the old value from the destination
strong_retain %new : $T            // Retain the new value
strong_release %old : $T           // Release the old
store %new to %1 : $*T      // Store the new value to the destination
```

except that `copy_addr` may be used even if `%0` is of an address-only type. The `copy_addr` may be given one or both of the `[take]` or `[initialization]` attributes:

- `[take]` destroys the value at the source address in the course of the copy.
- `[initialization]` indicates that the destination address is uninitialized. Without the attribute, the destination address is treated as already initialized, and the existing value will be destroyed before the new value is stored.

The three attributed forms thus behave like the following loadable type operations:

```
// take-assignment
  copy_addr [take] %0 to %1 : $*T
// is equivalent to:
  %new = load %0 : $*T
  %old = load %1 : $*T
  // no retain of %new!
  strong_release %old : $T
  store %new to %1 : $*T

// copy-initialization
  copy_addr %0 to [initialization] %1 : $*T
// is equivalent to:
  %new = load %0 : $*T
  strong_retain %new : $T
  // no load/release of %old!
  store %new to %1 : $*T

// take-initialization
  copy_addr [take] %0 to [initialization] %1 : $*T
// is equivalent to:
  %new = load %0 : $*T
  // no retain of %new!
  // no load/release of %old!
  store %new to %1 : $*T
```

If `T` is a trivial type, then `copy_addr` is always equivalent to its take-initialization form.

destroy_addr

```
sil-instruction ::= 'destroy_addr' sil-operand

destroy_addr %0 : $*T
// %0 must be of an address $*T type
```

Destroys the value in memory at address `%0`. If `T` is a non-trivial type, This is equivalent to:

```
%1 = load %0
strong_release %1
```

except that `destroy_addr` may be used even if `%0` is of an address-only type. This does not deallocate memory; it only destroys the pointed-to value, leaving the memory uninitialized.

If `T` is a trivial type, then `destroy_addr` is a no-op.

index_addr

```
sil-instruction ::= 'index_addr' sil-operand ',' sil-operand

%2 = index_addr %0 : $*T, %1 : $Builtin.Int<n>
// %0 must be of an address type $*T
// %1 must be of a builtin integer type
// %2 will be of type $*T
```

Given an address that references into an array of values, returns the address of the `%1`-th element relative to `%0`. The address must reference into a contiguous array. It is undefined to try to reference offsets within a non-array value, such as fields within a homogeneous struct or tuple type, or bytes within a value, using `index_addr`. (`Int8` address types have no special behavior in this regard, unlike `char*` or `void*` in C.) It is also undefined behavior to index out of bounds of an array, except to index the "past-the-end" address of the array.

index_raw_pointer

```
sil-instruction ::= 'index_raw_pointer' sil-operand ',' sil-operand

%2 = index_raw_pointer %0 : $Builtin.RawPointer, %1 : $Builtin.Int<n>
// %0 must be of $Builtin.RawPointer type
// %1 must be of a builtin integer type
// %2 will be of type $*T
```

Given a `Builtin.RawPointer` value `%0`, returns a pointer value at the byte offset `%1` relative to `%0`.

12.10.4 Reference Counting

These instructions handle reference counting of heap objects. Values of strong reference type have ownership semantics for the referenced heap object. Retain and release operations, however, are never implicit in SIL and always must be explicitly performed where needed. Retains and releases on the value may be freely moved, and balancing retains and releases may deleted, so long as an owning retain count is maintained for the uses of the value.

All reference-counting operations are defined to work correctly on null references (whether strong, unowned, or weak). A non-null reference must actually refer to a valid object of the indicated type (or a subtype). Address operands are required to be valid and non-null.

While SIL makes reference-counting operations explicit, the SIL type system also fully represents strength of reference. This is useful for several reasons:

1. Type-safety: it is impossible to erroneously emit SIL that naively uses a `@weak` or `@unowned` reference as if it were a strong reference.
2. Consistency: when a reference is kept in memory, instructions like `copy_addr` and `destroy_addr` implicitly carry the right semantics in the type of the address, rather than needing special variants or flags.
3. Ease of tooling: SIL directly stores the user's intended strength of reference, making it straightforward to generate instrumentation that would convey this to a memory profiler. In principle, with only a modest number of additions and restrictions on SIL, it would even be possible to drop all reference-counting instructions and use the type information to feed a garbage collector.

strong_retain

```
sil-instruction ::= 'strong_retain' sil-operand

strong_retain %0 : $T
// $T must be a reference type
```

Increases the strong retain count of the heap object referenced by `%0`.

strong_retain_autoreleased

```
sil-instruction ::= 'strong_retain_autoreleased' sil-operand

strong_retain_autoreleased %0 : $T
// $T must have a retainable pointer representation
```

Retains the heap object referenced by `%0` using the Objective-C ARC "autoreleased return value" optimization. The operand must be the result of an `apply` instruction with an Objective-C method callee, and the `strong_retain_autoreleased` instruction must be first use of the value after the defining `apply` instruction.

TODO: Specify all the other strong_retain_autoreleased constraints here.

strong_release

```
strong_release %0 : $T
// $T must be a reference type.
```

Decrements the strong reference count of the heap object referenced by `%0`. If the release operation brings the strong reference count of the object to zero, the object is destroyed and `@weak` references are cleared. When both its strong and unowned reference counts reach zero, the object's memory is deallocated.

strong_retain_unowned

```
sil-instruction ::= 'strong_retain_unowned' sil-operand

strong_retain_unowned %0 : $@unowned T
// $T must be a reference type
```

Asserts that the strong reference count of the heap object referenced by `%0` is still positive, then increases it by one.

unowned_retain

```
sil-instruction ::= 'unowned_retain' sil-operand

unowned_retain %0 : $@unowned T
// $T must be a reference type
```

Increments the unowned reference count of the heap object underlying `%0`.

unowned_release

```
sil-instruction ::= 'unowned_release' sil-operand

unowned_release %0 : $@unowned T
// $T must be a reference type
```

Decrements the unowned reference count of the heap object refereced by `%0`. When both its strong and unowned reference counts reach zero, the object's memory is deallocated.

load_weak

```
sil-instruction ::= 'load_weak' '[take]'? sil-operand

load_weak [take] %0 : $*@sil_weak Optional<T>
// $T must be an optional wrapping a reference type
```

Increments the strong reference count of the heap object held in the operand, which must be an initialized weak reference. The result is value of type `$Optional<T>`, except that it is `null` if the heap object has begun deallocation.

This operation must be atomic with respect to the final `strong_release` on the operand heap object. It need not be atomic with respect to `store_weak` operations on the same address.

store_weak

```
sil-instruction ::= 'store_weak' sil-value 'to' '[initialization]'? sil-operand

store_weak %0 to [initialization] %1 : $*@sil_weak Optional<T>
// $T must be an optional wrapping a reference type
```

Initializes or reassigns a weak reference. The operand may be `nil`.

If `[initialization]` is given, the weak reference must currently either be uninitialized or destroyed. If it is not given, the weak reference must currently be initialized.

This operation must be atomic with respect to the final `strong_release` on the operand (source) heap object. It need not be atomic with respect to `store_weak` or `load_weak` operations on the same address.

fix_lifetime

```
sil-instruction :: 'fix_lifetime' sil-operand

fix_lifetime %0 : $T
// Fix the lifetime of a value %0
```

```
fix_lifetime %1 : $*T
// Fix the lifetime of the memory object referenced by %1
```

Acts as a use of a value operand, or of the value in memory referenced by an address operand. Optimizations may not move operations that would destroy the value, such as `release_value`, `strong_release`, `copy_addr [take]`, or `destroy_addr`, past this instruction.

mark_dependence

```
sil-instruction :: 'mark_dependence' sil-operand 'on' sil-operand

%2 = mark_dependence %0 : $*T on %1 : $Builtin.NativeObject
```

Indicates that the validity of the first operand depends on the value of the second operand. Operations that would destroy the second value must not be moved before any instructions which depend on the result of this instruction, exactly as if the address had been obviously derived from that operand (e.g. using `ref_element_addr`).

The result is always equal to the first operand. The first operand will typically be an address, but it could be an address in a non-obvious form, such as a Builtin.RawPointer or a struct containing the same. Transformations should be somewhat forgiving here.

The second operand may have either object or address type. In the latter case, the dependency is on the current value stored in the address.

is_unique

```
sil-instruction ::= 'is_unique' sil-operand

%1 = is_unique %0 : $*T
// $T must be a reference-counted type
// %1 will be of type Builtin.Int1
```

Checks whether %0 is the address of a unique reference to a memory object. Returns 1 if the strong reference count is 1, and 0 if the strong reference count is greater than 1.

A discussion of the semantics can be found here: arcopts.is_unique.

is_unique_or_pinned

```
sil-instruction ::= 'is_unique_or_pinned' sil-operand

%1 = is_unique_or_pinned %0 : $*T
// $T must be a reference-counted type
// %1 will be of type Builtin.Int1
```

Checks whether %0 is the address of either a unique reference to a memory object or a reference to a pinned object. Returns 1 if the strong reference count is 1 or the object has been marked pinned by strong_pin.

copy_block

```
sil-instruction :: 'copy_block' sil-operand

%1 = copy_block %0 : $@convention(block) T -> U
```

Performs a copy of an Objective-C block. Unlike retains of other reference-counted types, this can produce a different value from the operand if the block is copied from the stack to the heap.

12.10.5 Literals

These instructions bind SIL values to literal constants or to global entities.

function_ref

```
sil-instruction ::= 'function_ref' sil-function-name ':' sil-type

%1 = function_ref @function : $@thin T -> U
// $@thin T -> U must be a thin function type
// %1 has type $T -> U
```

Creates a reference to a SIL function.

global_addr

```
sil-instruction ::= 'global_addr' sil-global-name ':' sil-type

%1 = global_addr @foo : $*Builtin.Word
```

Creates a reference to the address of a global variable.

integer_literal

```
sil-instruction ::= 'integer_literal' sil-type ',' int-literal

%1 = integer_literal $Builtin.Int<n>, 123
// $Builtin.Int<n> must be a builtin integer type
// %1 has type $Builtin.Int<n>
```

Creates an integer literal value. The result will be of type `Builtin.Int<n>`, which must be a builtin integer type. The literal value is specified using Swift's integer literal syntax.

float_literal

```
sil-instruction ::= 'float_literal' sil-type ',' int-literal

%1 = float_literal $Builtin.FP<n>, 0x3F800000
// $Builtin.FP<n> must be a builtin floating-point type
// %1 has type $Builtin.FP<n>
```

Creates a floating-point literal value. The result will be of type `` `Builtin.FP<n>`, which must be a builtin floating-point type. The literal value is specified as the bitwise representation of the floating point value, using Swift's hexadecimal integer literal syntax.

string_literal

```
sil-instruction ::= 'string_literal' encoding string-literal
encoding ::= 'utf8'
encoding ::= 'utf16'

%1 = string_literal "asdf"
// %1 has type $Builtin.RawPointer
```

Creates a reference to a string in the global string table. The result is a pointer to the data. The referenced string is always nul-terminated. The string literal value is specified using Swift's string literal syntax (though `\()` interpolations are not allowed).

12.10.6 Dynamic Dispatch

These instructions perform dynamic lookup of class and generic methods. They share a common set of attributes:

```
sil-method-attributes ::= '[' 'volatile'? ']'
```

The `volatile` attribute on a dynamic dispatch instruction indicates that the method lookup is semantically required (as, for example, in Objective-C). When the type of a dynamic dispatch instruction's operand is known, optimization passes can promote non-`volatile` dispatch instructions into static `function_ref` instructions.

If a dynamic dispatch instruction references an Objective-C method (indicated by the `foreign` marker on a method reference, as in `#NSObject.description!1.foreign`), then the instruction represents an `objc_msgSend` invocation. `objc_msgSend` invocations can only be used as the callee of an `apply` instruction or `partial_apply` instruction. They cannot be stored or used as `apply` or `partial_apply` arguments. `objc_msgSend` invocations must always be `volatile`.

class_method

```
sil-instruction ::= 'class_method' sil-method-attributes?
                      sil-operand ',' sil-decl-ref ':' sil-type

%1 = class_method %0 : $T, #T.method!1 : $@thin U -> V
// %0 must be of a class type or class metatype $T
// #T.method!1 must be a reference to a dynamically-dispatched method of T or
// of one of its superclasses, at uncurry level >= 1
// %1 will be of type $U -> V
```

Looks up a method based on the dynamic type of a class or class metatype instance. It is undefined behavior if the class value is null and the method is not an Objective-C method.

If:

- the instruction is not `[volatile]`,
- the referenced method is not a `foreign` method,
- and the static type of the class instance is known, or the method is known to be final,

then the instruction is a candidate for devirtualization optimization. A devirtualization pass can consult the module's *VTables* to find the SIL function that implements the method and promote the instruction to a static *function_ref*.

super_method

```
sil-instruction ::= 'super_method' sil-method-attributes?
                      sil-operand ',' sil-decl-ref ':' sil-type

%1 = super_method %0 : $T, #Super.method!1.foreign : $@thin U -> V
// %0 must be of a non-root class type or class metatype $T
// #Super.method!1.foreign must be a reference to an ObjC method of T's
// superclass or of one of its ancestor classes, at uncurry level >= 1
// %1 will be of type $@thin U -> V
```

Looks up a method in the superclass of a class or class metatype instance. Note that for native Swift methods, `super.method` calls are statically dispatched, so this instruction is only valid for Objective-C methods. It is undefined behavior if the class value is null and the method is not an Objective-C method.

witness_method

```
sil-instruction ::= 'witness_method' sil-method-attributes?
                      sil-type ',' sil-decl-ref ':' sil-type

%1 = witness_method $T, #Proto.method!1 \
  : $@thin @cc(witness_method) <Self: Proto> U -> V
// $T must be an archetype
// #Proto.method!1 must be a reference to a method of one of the protocol
//   constraints on T
// <Self: Proto> U -> V must be the type of the referenced method,
//   generic on Self
// %1 will be of type $@thin <Self: Proto> U -> V
```

Looks up the implementation of a protocol method for a generic type variable constrained by that protocol. The result will be generic on the `Self` archetype of the original protocol and have the `witness_method` calling convention. If the referenced protocol is an `@objc` protocol, the resulting type has the `objc` calling convention.

dynamic_method

```
sil-instruction ::= 'dynamic_method' sil-method-attributes?
                    sil-operand ',' sil-decl-ref ':' sil-type

%1 = dynamic_method %0 : $P, #X.method!1 : $@thin U -> V
// %0 must be of a protocol or protocol composition type $P,
// where $P contains the Swift.DynamicLookup protocol
// #X.method!1 must be a reference to an @objc method of any class
// or protocol type
//
// The "self" argument of the method type $@thin U -> V must be
//   Builtin.ObjCPointer
```

Looks up the implementation of an Objective-C method with the same selector as the named method for the dynamic type of the value inside an existential container. The "self" operand of the result function value is represented using an opaque type, the value for which must be projected out as a value of type `Builtin.ObjCPointer`.

It is undefined behavior if the dynamic type of the operand does not have an implementation for the Objective-C method with the selector to which the `dynamic_method` instruction refers, or if that implementation has parameter or result types that are incompatible with the method referenced by `dynamic_method`. This instruction should only be used in cases where its result will be immediately consumed by an operation that performs the selector check itself

(e.g., an `apply` that lowers to `objc_msgSend`). To query whether the operand has an implementation for the given method and safely handle the case where it does not, use *dynamic_method_br*.

12.10.7 Function Application

These instructions call functions or wrap them in partial application or specialization thunks.

apply

```
sil-instruction ::= 'apply' '[nothrow]'? sil-value
                      sil-apply-substitution-list?
                      '(' (sil-value (',' sil-value)*)? ')'
                      ':' sil-type

sil-apply-substitution-list ::= '<' sil-substitution
                                    (',' sil-substitution)* '>'
sil-substitution ::= type '=' type

%r = apply %0(%1, %2, ...) : $(A, B, ...) -> R
// Note that the type of the callee '%0' is specified *after* the arguments
// %0 must be of a concrete function type $(A, B, ...) -> R
// %1, %2, etc. must be of the argument types $A, $B, etc.
// %r will be of the return type $R

%r = apply %0<T = A, U = B>(%1, %2, ...) : $<T, U>(T, U, ...) -> R
// %0 must be of a polymorphic function type $<T, U>(T, U, ...) -> R
// %1, %2, etc. must be of the argument types after substitution $A, $B, etc.
// %r will be of the substituted return type $R'
```

Transfers control to function `%0`, passing it the given arguments. In the instruction syntax, the type of the callee is specified after the argument list; the types of the argument and of the defined value are derived from the function type of the callee. The input argument tuple type is destructured, and each element is passed as an individual argument. The `apply` instruction does no retaining or releasing of its arguments by itself; the *calling convention*'s retain/release policy must be handled by separate explicit `retain` and `release` instructions. The return value will likewise not be implicitly retained or released.

The callee value must have function type. That function type may not have an error result, except the instruction has the `nothrow` attribute set. The `nothrow` attribute specifies that the callee has an error result but does not actually throw. For the regular case of calling a function with error result, use `try_apply`.

NB: If the callee value is of a thick function type, `apply` currently consumes the callee value at +1 strong retain count.

If the callee is generic, all of its generic parameters must be bound by the given substitution list. The arguments and return value is given with these generic substitutions applied.

partial_apply

```
sil-instruction ::= 'partial_apply' sil-value
                      sil-apply-substitution-list?
                      '(' (sil-value (',' sil-value)*)? ')'
                      ':' sil-type

%c = partial_apply %0(%1, %2, ...) : $(Z..., A, B, ...) -> R
```

```
// Note that the type of the callee '%0' is specified *after* the arguments
// %0 must be of a concrete function type $(Z..., A, B, ...) -> R
// %1, %2, etc. must be of the argument types $A, $B, etc.,
//   of the tail part of the argument tuple of %0
// %c will be of the partially-applied thick function type (Z...) -> R

%c = partial_apply %0<T = A, U = B>(%1, %2, ...) : $(Z..., T, U, ...) -> R
// %0 must be of a polymorphic function type $<T, U>(T, U, ...) -> R
// %1, %2, etc. must be of the argument types after substitution $A, $B, etc.
//   of the tail part of the argument tuple of %0
// %r will be of the substituted thick function type $(Z'...) -> R'
```

Creates a closure by partially applying the function `%0` to a partial sequence of its arguments. In the instruction syntax, the type of the callee is specified after the argument list; the types of the argument and of the defined value are derived from the function type of the callee. The closure context will be allocated with retain count 1 and initialized to contain the values `%1`, `%2`, etc. The closed-over values will not be retained; that must be done separately before the `partial_apply`. The closure does however take ownership of the partially applied arguments; when the closure reference count reaches zero, the contained values will be destroyed.

If the callee is generic, all of its generic parameters must be bound by the given substitution list. The arguments are given with these generic substitutions applied, and the resulting closure is of concrete function type with the given substitutions applied. The generic parameters themselves cannot be partially applied; all of them must be bound. The result is always a concrete function.

TODO: The instruction, when applied to a generic function, currently implicitly performs abstraction difference transformations enabled by the given substitutions, such as promoting address-only arguments and returns to register arguments. This should be fixed.

This instruction is used to implement both curry thunks and closures. A curried function in Swift:

```
func foo(a:A)(b:B)(c:C)(d:D) -> E { /* body of foo */ }
```

emits curry thunks in SIL as follows (retains and releases omitted for clarity):

```
func @foo : $@thin A -> B -> C -> D -> E {
entry(%a : $A):
  %foo_1 = function_ref @foo_1 : $@thin (B, A) -> C -> D -> E
  %thunk = partial_apply %foo_1(%a) : $@thin (B, A) -> C -> D -> E
  return %thunk : $B -> C -> D -> E
}

func @foo_1 : $@thin (B, A) -> C -> D -> E {
entry(%b : $B, %a : $A):
  %foo_2 = function_ref @foo_2 : $@thin (C, B, A) -> D -> E
  %thunk = partial_apply %foo_2(%b, %a) : $@thin (C, B, A) -> D -> E
  return %thunk : $(B, A) -> C -> D -> E
}

func @foo_2 : $@thin (C, B, A) -> D -> E {
entry(%c : $C, %b : $B, %a : $A):
  %foo_3 = function_ref @foo_3 : $@thin (D, C, B, A) -> E
  %thunk = partial_apply %foo_3(%c, %b, %a) : $@thin (D, C, B, A) -> E
  return %thunk : $(C, B, A) -> D -> E
}

func @foo_3 : $@thin (D, C, B, A) -> E {
entry(%d : $D, %c : $C, %b : $B, %a : $A):
  // ... body of foo ...
```

```
}
```

A local function in Swift that captures context, such as `bar` in the following example:

```
func foo(x:Int) -> Int {
  func bar(y:Int) -> Int {
    return x + y
  }
  return bar(1)
}
```

lowers to an uncurried entry point and is curried in the enclosing function:

```
func @bar : $@thin (Int, @box Int, *Int) -> Int {
entry(%y : $Int, %x_box : $@box Int, %x_address : $*Int):
  // ... body of bar ...
}

func @foo : $@thin Int -> Int {
entry(%x : $Int):
  // Create a box for the 'x' variable
  %x_box = alloc_box $Int
  store %x to %x_box#1 : $*Int

  // Create the bar closure
  %bar_uncurried = function_ref @bar : $(Int, Int) -> Int
  %bar = partial_apply %bar_uncurried(%x_box#0, %x_box#1) \
    : $(Int, Builtin.ObjectPointer, *Int) -> Int

  // Apply it
  %1 = integer_literal $Int, 1
  %ret = apply %bar(%1) : $(Int) -> Int

  // Clean up
  release %bar : $(Int) -> Int
  return %ret : $Int
}
```

builtin

```
sil-instruction ::= 'builtin' string-literal
                      sil-apply-substitution-list?
                      '(' (sil-operand (',' sil-operand)*)? ')'
                      ':' sil-type

%1 = builtin "foo"(%1 : $T, %2 : $U) : $V
// "foo" must name a function in the Builtin module
```

Invokes functionality built into the backend code generator, such as LLVM- level instructions and intrinsics.

12.10.8 Metatypes

These instructions access metatypes, either statically by type name or dynamically by introspecting class or generic values.

metatype

```
sil-instruction ::= 'metatype' sil-type

%1 = metatype $T.metatype
// %1 has type $T.metatype
```

Creates a reference to the metatype object for type `T`.

value_metatype

```
sil-instruction ::= 'value_metatype' sil-type ',' sil-operand

%1 = value_metatype $T.metatype, %0 : $T
// %0 must be a value or address of type $T
// %1 will be of type $T.metatype
```

Obtains a reference to the dynamic metatype of the value `%0`.

existential_metatype

```
sil-instruction ::= 'existential_metatype' sil-type ',' sil-operand

%1 = existential_metatype $P.metatype, %0 : $P
// %0 must be a value of class protocol or protocol composition
//   type $P, or an address of address-only protocol type $*P
// %1 will be a $P.metatype value referencing the metatype of the
//   concrete value inside %0
```

Obtains the metatype of the concrete value referenced by the existential container referenced by `%0`.

objc_protocol

```
sil-instruction ::= 'objc_protocol' protocol-decl : sil-type

%0 = objc_protocol #ObjCProto : $Protocol
```

TODO Fill this in.

12.10.9 Aggregate Types

These instructions construct and project elements from structs, tuples, and class instances.

retain_value

```
sil-instruction ::= 'retain_value' sil-operand

retain_value %0 : $A
```

Retains a loadable value, which simply retains any references it holds.

For trivial types, this is a no-op. For reference types, this is equivalent to a `strong_retain`. For `@unowned` types, this is equivalent to an `unowned_retain`. In each of these cases, those are the preferred forms.

For aggregate types, especially enums, it is typically both easier and more efficient to reason about aggregate copies than it is to reason about copies of the subobjects.

release_value

```
sil-instruction ::= 'release_value' sil-operand

release_value %0 : $A
```

Destroys a loadable value, by releasing any retainable pointers within it.

This is defined to be equivalent to storing the operand into a stack allocation and using 'destroy_addr' to destroy the object there.

For trivial types, this is a no-op. For reference types, this is equivalent to a `strong_release`. For `@unowned` types, this is equivalent to an `unowned_release`. In each of these cases, those are the preferred forms.

For aggregate types, especially enums, it is typically both easier and more efficient to reason about aggregate destroys than it is to reason about destroys of the subobjects.

autorelease_value

```
sil-instruction ::= 'autorelease_value' sil-operand

autorelease_value %0 : $A
```

TODO Complete this section.

tuple

```
sil-instruction ::= 'tuple' sil-tuple-elements
sil-tuple-elements ::= '(' (sil-operand (',' sil-operand)*)? ')'
sil-tuple-elements ::= sil-type '(' (sil-value (',' sil-value)*)? ')'

%1 = tuple (%a : $A, %b : $B, ...)
// $A, $B, etc. must be loadable non-address types
// %1 will be of the "simple" tuple type $(A, B, ...)

%1 = tuple $(a:A, b:B, ...) (%a, %b, ...)
// (a:A, b:B, ...) must be a loadable tuple type
// %1 will be of the type $(a:A, b:B, ...)
```

Creates a loadable tuple value by aggregating multiple loadable values.

If the destination type is a "simple" tuple type, that is, it has no keyword argument labels or variadic arguments, then the first notation can be used, which interleaves the element values and types. If keyword names or variadic fields are specified, then the second notation must be used, which spells out the tuple type before the fields.

tuple_extract

```
sil-instruction ::= 'tuple_extract' sil-operand ',' int-literal

%1 = tuple_extract %0 : $(T...), 123
// %0 must be of a loadable tuple type $(T...)
// %1 will be of the type of the selected element of %0
```

Extracts an element from a loadable tuple value.

tuple_element_addr

```
sil-instruction ::= 'tuple_element_addr' sil-operand ',' int-literal

%1 = tuple_element_addr %0 : $*(T...), 123
// %0 must of a $*(T...) address-of-tuple type
// %1 will be of address type $*U where U is the type of the 123rd
//   element of T
```

Given the address of a tuple in memory, derives the address of an element within that value.

struct

```
sil-instruction ::= 'struct' sil-type '(' (sil-operand (',' sil-operand)*)? ')'

%1 = struct $S (%a : $A, %b : $B, ...)
// $S must be a loadable struct type
// $A, $B, ... must be the types of the physical 'var' fields of $S in order
// %1 will be of type $S
```

Creates a value of a loadable struct type by aggregating multiple loadable values.

struct_extract

```
sil-instruction ::= 'struct_extract' sil-operand ',' sil-decl-ref

%1 = struct_extract %0 : $S, #S.field
// %0 must be of a loadable struct type $S
// #S.field must be a physical 'var' field of $S
// %1 will be of the type of the selected field of %0
```

Extracts a physical field from a loadable struct value.

struct_element_addr

```
sil-instruction ::= 'struct_element_addr' sil-operand ',' sil-decl-ref

%1 = struct_element_addr %0 : $*S, #S.field
// %0 must be of a struct type $S
// #S.field must be a physical 'var' field of $S
// %1 will be the address of the selected field of %0
```

Given the address of a struct value in memory, derives the address of a physical field within the value.

ref_element_addr

```
sil-instruction ::= 'ref_element_addr' sil-operand ',' sil-decl-ref

%1 = ref_element_addr %0 : $C, #C.field
// %0 must be a value of class type $C
// #C.field must be a non-static physical field of $C
// %1 will be of type $*U where U is the type of the selected field
//   of C
```

Given an instance of a class, derives the address of a physical instance variable inside the instance. It is undefined behavior if the class value is null.

12.10.10 Enums

These instructions construct values of enum type. Loadable enum values are created with the *enum* instruction. Address-only enums require two-step initialization. First, if the case requires data, that data is stored into the enum at the address projected by *init_enum_data_addr*. This step is skipped for cases without data. Finally, the tag for the enum is injected with an *inject_enum_addr* instruction:

```
enum AddressOnlyEnum {
  case HasData(AddressOnlyType)
  case NoData
}

sil @init_with_data : $(AddressOnlyType) -> AddressOnlyEnum {
entry(%0 : $*AddressOnlyEnum, %1 : $*AddressOnlyType):
  // Store the data argument for the case.
  %2 = init_enum_data_addr %0 : $*AddressOnlyEnum, #AddressOnlyEnum.HasData
  copy_addr [take] %2 to [initialization] %1 : $*AddressOnlyType
  // Inject the tag.
  inject_enum_addr %0 : $*AddressOnlyEnum, #AddressOnlyEnum.HasData
  return
}

sil @init_without_data : $() -> AddressOnlyEnum {
  // No data. We only need to inject the tag.
  inject_enum_addr %0 : $*AddressOnlyEnum, #AddressOnlyEnum.NoData
  return
}
```

Accessing the value of a loadable enum is inseparable from dispatching on its discriminator and is done with the *switch_enum* terminator:

```
enum Foo { case A(Int), B(String) }

sil @switch_foo : $(Foo) -> () {
entry(%foo : $Foo):
  switch_enum %foo : $Foo, case #Foo.A: a_dest, case #Foo.B: b_dest

a_dest(%a : $Int):
  /* use %a */

b_dest(%b : $String):
  /* use %b */
}
```

An address-only enum can be tested by branching on it using the *switch_enum_addr* terminator. Its value can then be taken by destructively projecting the enum value with *unchecked_take_enum_data_addr*:

```
enum Foo<T> { case A(T), B(String) }

sil @switch_foo : $<T> (Foo<T>) -> () {
entry(%foo : $*Foo<T>):
  switch_enum_addr %foo : $*Foo<T>, case #Foo.A: a_dest, case #Foo.B: b_dest

a_dest:
  %a = unchecked_take_enum_data_addr %foo : $*Foo<T>, #Foo.A
  /* use %a */

b_dest:
  %b = unchecked_take_enum_data_addr %foo : $*Foo<T>, #Foo.B
  /* use %b */
}
```

enum

```
sil-instruction ::= 'enum' sil-type ',' sil-decl-ref (',' sil-operand)?

%1 = enum $U, #U.EmptyCase
%1 = enum $U, #U.DataCase, %0 : $T
// $U must be an enum type
// #U.DataCase or #U.EmptyCase must be a case of enum $U
// If #U.Case has a data type $T, %0 must be a value of type $T
// If #U.Case has no data type, the operand must be omitted
// %1 will be of type $U
```

Creates a loadable enum value in the given `case`. If the `case` has a data type, the enum value will contain the operand value.

unchecked_enum_data

```
sil-instruction ::= 'unchecked_enum_data' sil-operand ',' sil-decl-ref

%1 = unchecked_enum_data %0 : $U, #U.DataCase
// $U must be an enum type
// #U.DataCase must be a case of enum $U with data
// %1 will be of object type $T for the data type of case U.DataCase
```

Unsafely extracts the payload data for an enum `case` from an enum value. It is undefined behavior if the enum does not contain a value of the given case.

init_enum_data_addr

```
sil-instruction ::= 'init_enum_data_addr' sil-operand ',' sil-decl-ref

%1 = init_enum_data_addr %0 : $*U, #U.DataCase
// $U must be an enum type
// #U.DataCase must be a case of enum $U with data
// %1 will be of address type $*T for the data type of case U.DataCase
```

Projects the address of the data for an enum `case` inside an enum. This does not modify the enum or check its value. It is intended to be used as part of the initialization sequence for an address-only enum. Storing to the `init_enum_data_addr` for a case followed by `inject_enum_addr` with that same case is guaranteed to result in a fully-initialized enum value of that case being stored. Loading from the `init_enum_data_addr` of an initialized enum value or injecting a mismatched case tag is undefined behavior.

The address is invalidated as soon as the operand enum is fully initialized by an `inject_enum_addr`.

inject_enum_addr

```
sil-instruction ::= 'inject_enum_addr' sil-operand ',' sil-decl-ref

inject_enum_addr %0 : $*U, #U.Case
// $U must be an enum type
// #U.Case must be a case of enum $U
// %0 will be overlaid with the tag for #U.Case
```

Initializes the enum value referenced by the given address by overlaying the tag for the given case. If the case has no data, this instruction is sufficient to initialize the enum value. If the case has data, the data must be stored into the enum at the `init_enum_data_addr` address for the case *before* `inject_enum_addr` is applied. It is undefined behavior if `inject_enum_addr` is applied for a case with data to an uninitialized enum, or if `inject_enum_addr` is applied for a case with data when data for a mismatched case has been stored to the enum.

unchecked_take_enum_data_addr

```
sil-instruction ::= 'unchecked_take_enum_data_addr' sil-operand ',' sil-decl-ref

%1 = unchecked_take_enum_data_addr %0 : $*U, #U.DataCase
// $U must be an enum type
// #U.DataCase must be a case of enum $U with data
// %1 will be of address type $*T for the data type of case U.DataCase
```

Invalidates an enum value, and takes the address of the payload for the given enum `case` in-place in memory. The referenced enum value is no longer valid, but the payload value referenced by the result address is valid and must be destroyed. It is undefined behavior if the referenced enum does not contain a value of the given `case`. The result shares memory with the original enum value; the enum memory cannot be reinitialized as an enum until the payload has also been invalidated.

(1.0 only)

For the first payloaded case of an enum, `unchecked_take_enum_data_addr` is guaranteed to have no side effects; the enum value will not be invalidated.

select_enum

```
sil-instruction ::= 'select_enum' sil-operand sil-select-case*
                    (',' 'default' sil-value)?
                    ':' sil-type

%n = select_enum %0 : $U,      \
  case #U.Case1: %1,           \
  case #U.Case2: %2, /* ... */ \
  default %3 : $T
```

```
// $U must be an enum type
// #U.Case1, Case2, etc. must be cases of enum $U
// %1, %2, %3, etc. must have type $T
// %n has type $T
```

Selects one of the "case" or "default" operands based on the case of an enum value. This is equivalent to a trivial *switch_enum* branch sequence:

```
entry:
  switch_enum %0 : $U,            \
    case #U.Case1: bb1,           \
    case #U.Case2: bb2, /* ... */ \
    default bb_default
bb1:
  br cont(%1 : $T) // value for #U.Case1
bb2:
  br cont(%2 : $T) // value for #U.Case2
bb_default:
  br cont(%3 : $T) // value for default
cont(%n : $T):
  // use argument %n
```

but turns the control flow dependency into a data flow dependency. For address-only enums, *select_enum_addr* offers the same functionality for an indirectly referenced enum value in memory.

select_enum_addr

```
sil-instruction ::= 'select_enum_addr' sil-operand sil-select-case*
                    (',' 'default' sil-value)?
                    ':' sil-type

%n = select_enum_addr %0 : $*U,      \
  case #U.Case1: %1,           \
  case #U.Case2: %2, /* ... */ \
  default %3 : $T

// %0 must be the address of an enum type $*U
// #U.Case1, Case2, etc. must be cases of enum $U
// %1, %2, %3, etc. must have type $T
// %n has type $T
```

Selects one of the "case" or "default" operands based on the case of the referenced enum value. This is the address-only counterpart to *select_enum*.

12.10.11 Protocol and Protocol Composition Types

These instructions create and manipulate values of protocol and protocol composition type. From SIL's perspective, protocol and protocol composition types consist of an *existential container*, which is a generic container for a value of unknown runtime type, referred to as an "existential type" in type theory. The existential container consists of a reference to the *witness table(s)* for the protocol(s) referred to by the protocol type and a reference to the underlying *concrete value*, which may be either stored in-line inside the existential container for small values or allocated separately into a buffer owned and managed by the existential container for larger values.

Depending on the constraints applied to an existential type, an existential container may use one of several representations:

- **Opaque existential containers**: If none of the protocols in a protocol type are class protocols, then the existential container for that type is address-only and referred to in the implementation as an *opaque existential container*. The value semantics of the existential container propagate to the contained concrete value. Applying `copy_addr` to an opaque existential container copies the contained concrete value, deallocating or reallocating the destination container's owned buffer if necessary. Applying `destroy_addr` to an opaque existential container destroys the concrete value and deallocates any buffers owned by the existential container. The following instructions manipulate opaque existential containers:
  - *init_existential_addr*
  - *open_existential_addr*
  - *deinit_existential_addr*
- **Class existential containers**: If a protocol type is constrained by one or more class protocols, then the existential container for that type is loadable and referred to in the implementation as a *class existential container*. Class existential containers have reference semantics and can be `retain`-ed and `release`-d. The following instructions manipulate class existential containers:
  - *init_existential_ref*
  - *open_existential_ref*
- **Metatype existential containers**: Existential metatypes use a container consisting of the type metadata for the conforming type along with the protocol conformances. Metatype existential containers are trivial types. The following instructions manipulate metatype existential containers:
  - *init_existential_metatype*
  - *open_existential_metatype*
- **Boxed existential containers**: The standard library `ErrorType` protocol uses a size-optimized reference-counted container, which indirectly stores the conforming value. Boxed existential containers can be `retain`-ed and `release`-d. The following instructions manipulate boxed existential containers:
  - *alloc_existential_box*
  - *open_existential_box*
  - *dealloc_existential_box*

Some existential types may additionally support specialized representations when they contain certain known concrete types. For example, when Objective-C interop is available, the `ErrorType` protocol existential supports a class existential container representation for `NSError` objects, so it can be initialized from one using `init_existential_ref` instead of the more expensive `alloc_existential_box`:

```
bb(%nserror: $NSError):
  // The slow general way to form an ErrorType, allocating a box and
  // storing to its value buffer:
  %error1 = alloc_existential_box $ErrorType, $NSError
  strong_retain %nserror: $NSError
  store %nserror to %error1#1 : $NSError

  // The fast path supported for NSError:
  strong_retain %nserror: $NSError
  %error2 = init_existential_ref %nserror: $NSError, $ErrorType
```

init_existential_addr

```
sil-instruction ::= 'init_existential_addr' sil-operand ',' sil-type

%1 = init_existential_addr %0 : $*P, $T
// %0 must be of a $*P address type for non-class protocol or protocol
//   composition type P
// $T must be an AST type that fulfills protocol(s) P
// %1 will be of type $*T', where T' is the maximally abstract lowering
//    of type T
```

Partially initializes the memory referenced by `%0` with an existential container prepared to contain a value of type `$T`. The result of the instruction is an address referencing the storage for the contained value, which remains uninitialized. The contained value must be `store`-d or `copy_addr`-ed to in order for the existential value to be fully initialized. If the existential container needs to be destroyed while the contained value is uninitialized, `deinit_existential_addr` must be used to do so. A fully initialized existential container can be destroyed with `destroy_addr` as usual. It is undefined behavior to `destroy_addr` a partially-initialized existential container.

deinit_existential_addr

```
sil-instruction ::= 'deinit_existential_addr' sil-operand

deinit_existential_addr %0 : $*P
// %0 must be of a $*P address type for non-class protocol or protocol
// composition type P
```

Undoes the partial initialization performed by `init_existential_addr`. `deinit_existential_addr` is only valid for existential containers that have been partially initialized by `init_existential_addr` but haven't had their contained value initialized. A fully initialized existential must be destroyed with `destroy_addr`.

open_existential_addr

```
sil-instruction ::= 'open_existential_addr' sil-operand 'to' sil-type

%1 = open_existential_addr %0 : $*P to $*@opened P
// %0 must be of a $*P type for non-class protocol or protocol composition
//   type P
// $*@opened P must be a unique archetype that refers to an opened
// existential type P.
// %1 will be of type $*P
```

Obtains the address of the concrete value inside the existential container referenced by `%0`. The protocol conformances associated with this existential container are associated directly with the archetype `$*@opened P`. This pointer can be used with any operation on archetypes, such as `witness_method`.

init_existential_ref

```
sil-instruction ::= 'init_existential_ref' sil-operand ':' sil-type ','
                                           sil-type

%1 = init_existential_ref %0 : $C' : $C, $P
// %0 must be of class type $C', lowered from AST type $C, conforming to
```

```
//    protocol(s) $P
// $P must be a class protocol or protocol composition type
// %1 will be of type $P
```

Creates a class existential container of type `$P` containing a reference to the class instance `%0`.

open_existential_ref

```
sil-instruction ::= 'open_existential_ref' sil-operand 'to' sil-type

%1 = open_existential_ref %0 : $P to $@opened P
// %0 must be of a $P type for a class protocol or protocol composition
// $@opened P must be a unique archetype that refers to an opened
//   existential type P
// %1 will be of type $@opened P
```

Extracts the class instance reference from a class existential container. The protocol conformances associated with this existential container are associated directly with the archetype `@opened P`. This pointer can be used with any operation on archetypes, such as `witness_method`. When the operand is of metatype type, the result will be the metatype of the opened archetype.

init_existential_metatype

```
sil-instruction ::= 'init_existential_metatype' sil-operand ',' sil-type

%1 = init_existential_metatype $0 : $@<rep> T.Type, $@<rep> P.Type
// %0 must be of a metatype type $@<rep> T.Type where T: P
// %@<rep> P.Type must be the existential metatype of a protocol or protocol
//    composition, with the same metatype representation <rep>
// %1 will be of type $@<rep> P.Type
```

Creates a metatype existential container of type `$P.Type` containing the conforming metatype of `$T`.

open_existential_metatype

```
sil-instruction ::= 'open_existential_metatype' sil-operand 'to' sil-type

%1 = open_existential_metatype %0 : $@<rep> P.Type to $@<rep> (@opened P).Type
// %0 must be of a $P.Type existential metatype for a protocol or protocol
//    composition
// $@<rep> (@opened P).Type must be the metatype of a unique archetype that
//   refers to an opened existential type P, with the same metatype
//   representation <rep>
// %1 will be of type $@<rep> (@opened P).Type
```

Extracts the metatype from an existential metatype. The protocol conformances associated with this existential container are associated directly with the archetype `@opened P`.

alloc_existential_box

```
sil-instruction ::= 'alloc_existential_box' sil-type ',' sil-type

%1 = alloc_existential_box $P, $T
// $P must be a protocol or protocol composition type with boxed
//   representation
// $T must be an AST type that conforms to P
// %1#0 will be of type $P
// %1#1 will be of type $*T', where T' is the most abstracted lowering of T
```

Allocates a boxed existential container of type `$P` with space to hold a value of type `$T'`. The box is not fully initialized until a valid value has been stored into the box. If the box must be deallocated before it is fully initialized, `dealloc_existential_box` must be used. A fully initialized box can be `retain`-ed and `release`-d like any reference-counted type. The address `%0#1` is dependent on the lifetime of the owner reference `%0#0`.

open_existential_box

```
sil-instruction ::= 'open_existential_box' sil-operand 'to' sil-type

%1 = open_existential_box %0 : $P to $*@opened P
// %0 must be a value of boxed protocol or protocol composition type $P
// %@opened P must be the address type of a unique archetype that refers to
///   an opened existential type P
// %1 will be of type $*@opened P
```

Projects the address of the value inside a boxed existential container, and uses the enclosed type and protocol conformance metadata to bind the opened archetype `$@opened P`. The result address is dependent on both the owning box and the enclosing function; in order to "open" a boxed existential that has directly adopted a class reference, temporary scratch space may need to have been allocated.

dealloc_existential_box

```
sil-instruction ::= 'dealloc_existential_box' sil-operand, sil-type

dealloc_existential_box %0 : $P, $T
// %0 must be an uninitialized box of boxed existential container type $P
// $T must be the AST type for which the box was allocated
```

Deallocates a boxed existential container. The value inside the existential buffer is not destroyed; either the box must be uninitialized, or the value must have been projected out and destroyed beforehand. It is undefined behavior if the concrete type `$T` is not the same type for which the box was allocated with `alloc_existential_box`.

12.10.12 Blocks

project_block_storage

```
sil-instruction ::= 'project_block_storage' sil-operand ':' sil-type
```

init_block_storage_header

TODO Fill this in. The printing of this instruction looks incomplete on trunk currently.

12.10.13 Unchecked Conversions

These instructions implement type conversions which are not checked. These are either user-level conversions that are always safe and do not need to be checked, or implementation detail conversions that are unchecked for performance or flexibility.

upcast

```
sil-instruction ::= 'upcast' sil-operand 'to' sil-type

%1 = upcast %0 : $D to $B
// $D and $B must be class types or metatypes, with B a superclass of D
// %1 will have type $B
```

Represents a conversion from a derived class instance or metatype to a superclass, or from a base-class-constrained archetype to its base class.

address_to_pointer

```
sil-instruction ::= 'address_to_pointer' sil-operand 'to' sil-type

%1 = address_to_pointer %0 : $*T to $Builtin.RawPointer
// %0 must be of an address type $*T
// %1 will be of type Builtin.RawPointer
```

Creates a `Builtin.RawPointer` value corresponding to the address `%0`. Converting the result pointer back to an address of the same type will give an address equivalent to `%0`. It is undefined behavior to cast the `RawPointer` to any address type other than its original address type or any *layout compatible types*.

pointer_to_address

```
sil-instruction ::= 'pointer_to_address' sil-operand 'to' sil-type

%1 = pointer_to_address %0 : $Builtin.RawPointer to $*T
// %1 will be of type $*T
```

Creates an address value corresponding to the `Builtin.RawPointer` value `%0`. Converting a `RawPointer` back to an address of the same type as its originating `address_to_pointer` instruction gives back an equivalent address. It is undefined behavior to cast the `RawPointer` back to any type other than its original address type or *layout compatible types*. It is also undefined behavior to cast a `RawPointer` from a heap object to any address type.

unchecked_ref_cast

```
sil-instruction ::= 'unchecked_ref_cast' sil-operand 'to' sil-type

%1 = unchecked_ref_cast %0 : $A to $B
// %0 must be an object of type $A
// $A must be a type with retainable pointer representation
// %1 will be of type $B
// $B must be a type with retainable pointer representation
```

Converts a heap object reference to another heap object reference type. This conversion is unchecked, and it is undefined behavior if the destination type is not a valid type for the heap object. The heap object reference on either side of the cast may be a class existential, and may be wrapped in one level of Optional.

unchecked_ref_cast_addr

```
sil-instruction ::= 'unchecked_ref_cast_addr'
                    sil-type 'in' sil-operand 'to'
                    sil-type 'in' sil-operand

unchecked_ref_cast_addr $A in %0 : $*A to $B in %1 : $*B
// %0 must be the address of an object of type $A
// $A must be a type with retainable pointer representation
// %1 must be the address of storage for an object of type $B
// $B must be a retainable pointer representation
```

Loads a heap object reference from an address and stores it at the address of another uninitialized heap object reference. The loaded reference is always taken, and the stored reference is initialized. This conversion is unchecked, and it is undefined behavior if the destination type is not a valid type for the heap object. The heap object reference on either side of the cast may be a class existential, and may be wrapped in one level of Optional.

unchecked_addr_cast

```
sil-instruction ::= 'unchecked_addr_cast' sil-operand 'to' sil-type

%1 = unchecked_addr_cast %0 : $*A to $*B
// %0 must be an address
// %1 will be of type $*B
```

Converts an address to a different address type. Using the resulting address is undefined unless `B` is layout compatible with `A`. The layout of `A` may be smaller than that of `B` as long as the lower order bytes have identical layout.

unchecked_trivial_bit_cast

```
sil-instruction ::= 'unchecked_trivial_bit_cast' sil-operand 'to' sil-type

%1 = unchecked_trivial_bit_cast %0 : $Builtin.NativeObject to $Builtin.Word
// %0 must be an object.
// %1 must be an object with trivial type.
```

Bitcasts an object of type `A` to be of same sized or smaller type `B` with the constraint that `B` must be trivial. This can be used for bitcasting among trivial types, but more importantly is a one way bitcast from non-trivial types to trivial types.

unchecked_bitwise_cast

```
sil-instruction ::= 'unchecked_bitwise_cast' sil-operand 'to' sil-type

%1 = unchecked_bitwise_cast %0 : $A to $B
```

Bitwise copies an object of type `A` into a new object of type `B` of the same size or smaller.

ref_to_raw_pointer

```
sil-instruction ::= 'ref_to_raw_pointer' sil-operand 'to' sil-type

%1 = ref_to_raw_pointer %0 : $C to $Builtin.RawPointer
// $C must be a class type, or Builtin.ObjectPointer, or Builtin.ObjCPointer
// %1 will be of type $Builtin.RawPointer
```

Converts a heap object reference to a `Builtin.RawPointer`. The `RawPointer` result can be cast back to the originating class type but does not have ownership semantics. It is undefined behavior to cast a `RawPointer` from a heap object reference to an address using `pointer_to_address`.

raw_pointer_to_ref

```
sil-instruction ::= 'raw_pointer_to_ref' sil-operand 'to' sil-type

%1 = raw_pointer_to_ref %0 : $Builtin.RawPointer to $C
// $C must be a class type, or Builtin.ObjectPointer, or Builtin.ObjCPointer
// %1 will be of type $C
```

Converts a `Builtin.RawPointer` back to a heap object reference. Casting a heap object reference to `Builtin.RawPointer` back to the same type gives an equivalent heap object reference (though the raw pointer has no ownership semantics for the object on its own). It is undefined behavior to cast a `RawPointer` to a type unrelated to the dynamic type of the heap object. It is also undefined behavior to cast a `RawPointer` from an address to any heap object type.

ref_to_unowned

```
sil-instruction ::= 'ref_to_unowned' sil-operand

%1 = unowned_to_ref %0 : T
// $T must be a reference type
// %1 will have type $@unowned T
```

Adds the `@unowned` qualifier to the type of a reference to a heap object. No runtime effect.

unowned_to_ref

```
sil-instruction ::= 'unowned_to_ref' sil-operand

%1 = unowned_to_ref %0 : $@unowned T
// $T must be a reference type
// %1 will have type $T
```

Strips the `@unowned` qualifier off the type of a reference to a heap object. No runtime effect.

ref_to_unmanaged

TODO

unmanaged_to_ref

TODO

convert_function

```
sil-instruction ::= 'convert_function' sil-operand 'to' sil-type

%1 = convert_function %0 : $T -> U to $T' -> U'
// %0 must be of a function type $T -> U ABI-compatible with $T' -> U'
//   (see below)
// %1 will be of type $T' -> U'
```

Performs a conversion of the function `%0` to type `T`, which must be ABI- compatible with the type of `%0`. Function types are ABI-compatible if their input and result types are tuple types that, after destructuring, differ only in the following ways:

- Corresponding tuple elements may add, remove, or change keyword names. `(a:Int, b:Float, UnicodeScalar) -> ()` and `(x:Int, Float, z:UnicodeScalar) -> ()` are ABI compatible.
- A class tuple element of the destination type may be a superclass or subclass of the source type's corresponding tuple element.

The function types may also differ in attributes, with the following caveats:

- The `convention` attribute cannot be changed.
- A `@noreturn` function may be converted to a non-`@noreturn` type and vice-versa.

thin_function_to_pointer

TODO

pointer_to_thin_function

TODO

ref_to_bridge_object

```
sil-instruction ::= 'ref_to_bridge_object' sil-operand, sil-operand

%2 = ref_to_bridge_object %0 : $C, %1 : $Builtin.Word
// %1 must be of reference type $C
// %2 will be of type Builtin.BridgeObject
```

Creates a `Builtin.BridgeObject` that references `%0`, with spare bits in the pointer representation populated by bitwise-OR-ing in the value of `%1`. It is undefined behavior if this bitwise OR operation affects the reference identity of `%0`; in other words, after the following instruction sequence:

```
%b = ref_to_bridge_object %r : $C, %w : $Builtin.Word
%r2 = bridge_object_to_ref %b : $Builtin.BridgeObject to $C
```

`%r` and `%r2` must be equivalent. In particular, it is assumed that retaining or releasing the `BridgeObject` is equivalent to retaining or releasing the original reference, and that the above `ref_to_bridge_object` / `bridge_object_to_ref` round-trip can be folded away to a no-op.

On platforms with ObjC interop, there is additionally a platform-specific bit in the pointer representation of a `BridgeObject` that is reserved to indicate whether the referenced object has native Swift refcounting. It is undefined behavior to set this bit when the first operand references an Objective-C object.

bridge_object_to_ref

```
sil-instruction ::= 'bridge_object_to_ref' sil-operand 'to' sil-type

%1 = bridge_object_to_ref %0 : $Builtin.BridgeObject to $C
// $C must be a reference type
// %1 will be of type $C
```

Extracts the object reference from a `Builtin.BridgeObject`, masking out any spare bits.

bridge_object_to_word

```
sil-instruction ::= 'bridge_object_to_word' sil-operand 'to' sil-type

%1 = bridge_object_to_word %0 : $Builtin.BridgeObject to $Builtin.Word
// %1 will be of type $Builtin.Word
```

Provides the bit pattern of a `Builtin.BridgeObject` as an integer.

thin_to_thick_function

```
sil-instruction ::= 'thin_to_thick_function' sil-operand 'to' sil-type

%1 = thin_to_thick_function %0 : $@convention(thin) T -> U to $T -> U
// %0 must be of a thin function type $@convention(thin) T -> U
// The destination type must be the corresponding thick function type
// %1 will be of type $T -> U
```

Converts a thin function value, that is, a bare function pointer with no context information, into a thick function value with ignored context. Applying the resulting thick function value is equivalent to applying the original thin value. The `thin_to_thick_function` conversion may be eliminated if the context is proven not to be needed.

thick_to_objc_metatype

```
sil-instruction ::= 'thick_to_objc_metatype' sil-operand 'to' sil-type

%1 = thick_to_objc_metatype %0 : $@thick T.metatype to $@objc_metatype T.metatype
// %0 must be of a thick metatype type $@thick T.metatype
// The destination type must be the corresponding Objective-C metatype type
// %1 will be of type $@objc_metatype T.metatype
```

Converts a thick metatype to an Objective-C class metatype. `T` must be of class, class protocol, or class protocol composition type.

objc_to_thick_metatype

```
sil-instruction ::= 'objc_to_thick_metatype' sil-operand 'to' sil-type

%1 = objc_to_thick_metatype %0 : $@objc_metatype T.metatype to $@thick T.metatype
// %0 must be of an Objective-C metatype type $@objc_metatype T.metatype
// The destination type must be the corresponding thick metatype type
// %1 will be of type $@thick T.metatype
```

Converts an Objective-C class metatype to a thick metatype. `T` must be of class, class protocol, or class protocol composition type.

objc_metatype_to_object

TODO

objc_existential_metatype_to_object

TODO

is_nonnull

```
sil-instruction ::= 'is_nonnull' sil-operand

%1 = is_nonnull %0 : $C
// %0 must be of reference or function type $C
// %1 will be of type Builtin.Int1
```

Checks whether a reference type value is null, returning 1 if the value is not null, or 0 if it is null. If the value is a function type, it checks the function pointer (not the data pointer) for null.

This is not a sensical thing for SIL to represent given that reference types are non-nullable, but makes sense at the machine level. This is a horrible hack that should go away someday.

12.10.14 Checked Conversions

Some user-level cast operations can fail and thus require runtime checking.

The *unconditional_checked_cast_addr* and *unconditional_checked_cast* instructions performs an unconditional checked cast; it is a runtime failure if the cast fails. The *checked_cast_addr_br* and *checked_cast_br* terminator instruction performs a conditional checked cast; it branches to one of two destinations based on whether the cast succeeds or not.

unconditional_checked_cast

```
sil-instruction ::= 'unconditional_checked_cast' sil-operand 'to' sil-type

%1 = unconditional_checked_cast %0 : $A to $B
%1 = unconditional_checked_cast %0 : $*A to $*B
// $A and $B must be both objects or both addresses
// %1 will be of type $B or $*B
```

Performs a checked scalar conversion, causing a runtime failure if the conversion fails.

unconditional_checked_cast_addr

```
sil-instruction ::= 'unconditional_checked_cast_addr'
                    sil-cast-consumption-kind
                    sil-type 'in' sil-operand 'to'
                    sil-type 'in' sil-operand
sil-cast-consumption-kind ::= 'take_always'
sil-cast-consumption-kind ::= 'take_on_success'
sil-cast-consumption-kind ::= 'copy_on_success'

%1 = unconditional_checked_cast_addr take_on_success $A in %0 : $*@thick A to $B in
↪$*@thick B
// $A and $B must be both addresses
// %1 will be of type $*B
```

Performs a checked indirect conversion, causing a runtime failure if the conversion fails.

12.10.15 Runtime Failures

cond_fail

```
sil-instruction ::= 'cond_fail' sil-operand

cond_fail %0 : $Builtin.Int1
// %0 must be of type $Builtin.Int1
```

This instruction produces a *runtime failure* if the operand is one. Execution proceeds normally if the operand is zero.

12.10.16 Terminators

These instructions terminate a basic block. Every basic block must end with a terminator. Terminators may only appear as the final instruction of a basic block.

unreachable

```
sil-terminator ::= 'unreachable'

unreachable
```

Indicates that control flow must not reach the end of the current basic block. It is a dataflow error if an unreachable terminator is reachable from the entry point of a function and is not immediately preceded by an `apply` of a `@noreturn` function.

return

```
sil-terminator ::= 'return' sil-operand

return %0 : $T
// $T must be the return type of the current function
```

Exits the current function and returns control to the calling function. If the current function was invoked with an `apply` instruction, the result of that function will be the operand of this `return` instruction. If the current function was invoked with a `try_apply` instruction, control resumes at the normal destination, and the value of the basic block argument will be the operand of this ``return`` instruction.

`return` does not retain or release its operand or any other values.

A function must not contain more than one `return` instruction.

autorelease_return

```
sil-terminator ::= 'autorelease_return' sil-operand

autorelease_return %0 : $T
// $T must be the return type of the current function, which must be of
//   class type
```

Exits the current function and returns control to the calling function. The result of the `apply` instruction that invoked the current function will be the operand of this `return` instruction. The return value is autoreleased into the active Objective-C autorelease pool using the "autoreleased return value" optimization. The current function must use the `@cc(objc_method)` calling convention.

throw

```
sil-terminator ::= 'throw' sil-operand

throw %0 : $T
// $T must be the error result type of the current function
```

Exits the current function and returns control to the calling function. The current function must have an error result, and so the function must have been invoked with a `try_apply` instruction. Control will resume in the error destination of that instruction, and the basic block argument will be the operand of the ``throw``.

`throw` does not retain or release its operand or any other values.

A function must not contain more than one `throw` instruction.

br

```
sil-terminator ::= 'br' sil-identifier
                     '(' (sil-operand (',' sil-operand)*)? ')'

br label (%0 : $A, %1 : $B, ...)
// `label` must refer to a basic block label within the current function
// %0, %1, etc. must be of the types of `label`'s arguments
```

Unconditionally transfers control from the current basic block to the block labeled `label`, binding the given values to the arguments of the destination basic block.

cond_br

```
sil-terminator ::= 'cond_br' sil-operand ','
                     sil-identifier '(' (sil-operand (',' sil-operand)*)? ')' ','
                     sil-identifier '(' (sil-operand (',' sil-operand)*)? ')'

cond_br %0 : $Builtin.Int1, true_label (%a : $A, %b : $B, ...), \
                              false_label (%x : $X, %y : $Y, ...)
// %0 must be of $Builtin.Int1 type
// `true_label` and `false_label` must refer to block labels within the
//   current function and must not be identical
// %a, %b, etc. must be of the types of `true_label`'s arguments
// %x, %y, etc. must be of the types of `false_label`'s arguments
```

Conditionally branches to `true_label` if `%0` is equal to `1` or to `false_label` if `%0` is equal to `0`, binding the corresponding set of values to the the arguments of the chosen destination block.

switch_value

```
sil-terminator ::= 'switch_value' sil-operand
                     (',' sil-switch-value-case)*
                     (',' sil-switch-default)?
sil-switch-value-case ::= 'case' sil-value ':' sil-identifier
sil-switch-default ::= 'default' sil-identifier

switch_value %0 : $Builtin.Int<n>, case %1: label1, \
                                   case %2: label2, \
                                   ...,            \
                                   default labelN

// %0 must be a value of builtin integer type $Builtin.Int<n>
// `label1` through `labelN` must refer to block labels within the current
//   function
// FIXME: All destination labels currently must take no arguments
```

Conditionally branches to one of several destination basic blocks based on a value of builtin integer or function type. If the operand value matches one of the `case` values of the instruction, control is transferred to the corresponding basic block. If there is a `default` basic block, control is transferred to it if the value does not match any of the `case` values. It is undefined behavior if the value does not match any cases and no `default` branch is provided.

select_value

```
sil-instruction ::= 'select_value' sil-operand sil-select-value-case*
                    (',' 'default' sil-value)?
                    ':' sil-type
sil-selct-value-case ::= 'case' sil-value ':' sil-value


%n = select_value %0 : $U, \
  case %c1: %r1,           \
  case %c2: %r2, /* ... */ \
  default %r3 : $T

// $U must be a builtin type. Only integers types are supported currently.
// c1, c2, etc must be of type $U
```

```
// %r1, %r2, %r3, etc. must have type $T
// %n has type $T
```

Selects one of the "case" or "default" operands based on the case of an value. This is equivalent to a trivial *switch_value* branch sequence:

```
entry:
  switch_value %0 : $U,            \
    case %c1: bb1,           \
    case %c2: bb2, /* ... */ \
    default bb_default
bb1:
  br cont(%r1 : $T) // value for %c1
bb2:
  br cont(%r2 : $T) // value for %c2
bb_default:
  br cont(%r3 : $T) // value for default
cont(%n : $T):
  // use argument %n
```

but turns the control flow dependency into a data flow dependency.

switch_enum

```
sil-terminator ::= 'switch_enum' sil-operand
                     (',' sil-switch-enum-case)*
                     (',' sil-switch-default)?
sil-switch-enum-case ::= 'case' sil-decl-ref ':' sil-identifier

switch_enum %0 : $U, case #U.Foo: label1, \
                      case #U.Bar: label2, \
                      ...,                 \
                      default labelN

// %0 must be a value of enum type $U
// #U.Foo, #U.Bar, etc. must be 'case' declarations inside $U
// `label1` through `labelN` must refer to block labels within the current
//   function
// label1 must take either no basic block arguments, or a single argument
//   of the type of #U.Foo's data
// label2 must take either no basic block arguments, or a single argument
//   of the type of #U.Bar's data, etc.
// labelN must take no basic block arguments
```

Conditionally branches to one of several destination basic blocks based on the discriminator in a loadable `enum` value. Unlike `switch_int`, `switch_enum` requires coverage of the operand type: If the `enum` type is resilient, the `default` branch is required; if the `enum` type is fragile, the `default` branch is required unless a destination is assigned to every `case` of the `enum`. The destination basic block for a `case` may take an argument of the corresponding `enum` `case`'s data type (or of the address type, if the operand is an address). If the branch is taken, the destination's argument will be bound to the associated data inside the original enum value. For example:

```
enum Foo {
  case Nothing
  case OneInt(Int)
  case TwoInts(Int, Int)
}
```

```
sil @sum_of_foo : $Foo -> Int {
entry(%x : $Foo):
  switch_enum %x : $Foo,        \
    case #Foo.Nothing: nothing, \
    case #Foo.OneInt:  one_int, \
    case #Foo.TwoInts: two_ints

nothing:
  %zero = integer_literal 0 : $Int
  return %zero : $Int

one_int(%y : $Int):
  return %y : $Int

two_ints(%ab : $(Int, Int)):
  %a = tuple_extract %ab : $(Int, Int), 0
  %b = tuple_extract %ab : $(Int, Int), 1
  %add = function_ref @add : $(Int, Int) -> Int
  %result = apply %add(%a, %b) : $(Int, Int) -> Int
  return %result : $Int
}
```

On a path dominated by a destination block of `switch_enum`, copying or destroying the basic block argument has equivalent reference counting semantics to copying or destroying the `switch_enum` operand:

```
  // This retain_value...
  retain_value %e1 : $Enum
  switch_enum %e1, case #Enum.A: a, case #Enum.B: b
a(%a : $A):
  // ...is balanced by this release_value
  release_value %a
b(%b : $B):
  // ...and this one
  release_value %b
```

switch_enum_addr

```
sil-terminator ::= 'switch_enum_addr' sil-operand
                     (',' sil-switch-enum-case)*
                     (',' sil-switch-default)?

switch_enum_addr %0 : $*U, case #U.Foo: label1, \
                                        case #U.Bar: label2, \
                                        ...,                 \
                                        default labelN

// %0 must be the address of an enum type $*U
// #U.Foo, #U.Bar, etc. must be cases of $U
// `label1` through `labelN` must refer to block labels within the current
//   function
// The destinations must take no basic block arguments
```

Conditionally branches to one of several destination basic blocks based on the discriminator in the enum value referenced by the address operand.

Unlike `switch_int`, `switch_enum` requires coverage of the operand type: If the `enum` type is resilient, the `default` branch is required; if the `enum` type is fragile, the `default` branch is required unless a destination is assigned to every `case` of the `enum`. Unlike `switch_enum`, the payload value is not passed to the destination basic blocks; it must be projected out separately with *unchecked_take_enum_data_addr*.

dynamic_method_br

```
sil-terminator ::= 'dynamic_method_br' sil-operand ',' sil-decl-ref
                     ',' sil-identifier ',' sil-identifier

dynamic_method_br %0 : $P, #X.method!1, bb1, bb2
// %0 must be of protocol type
// #X.method!1 must be a reference to an @objc method of any class
// or protocol type
```

Looks up the implementation of an Objective-C method with the same selector as the named method for the dynamic type of the value inside an existential container. The "self" operand of the result function value is represented using an opaque type, the value for which must be projected out as a value of type `Builtin.ObjCPointer`.

If the operand is determined to have the named method, this instruction branches to `bb1`, passing it the uncurried function corresponding to the method found. If the operand does not have the named method, this instruction branches to `bb2`.

checked_cast_br

```
sil-terminator ::= 'checked_cast_br' sil-checked-cast-exact?
                    sil-operand 'to' sil-type ','
                    sil-identifier ',' sil-identifier
sil-checked-cast-exact ::= '[' 'exact' ']'

checked_cast_br %0 : $A to $B, bb1, bb2
checked_cast_br %0 : $*A to $*B, bb1, bb2
checked_cast_br [exact] %0 : $A to $A, bb1, bb2
// $A and $B must be both object types or both address types
// bb1 must take a single argument of type $B or $*B
// bb2 must take no arguments
```

Performs a checked scalar conversion from `$A` to `$B`. If the conversion succeeds, control is transferred to `bb1`, and the result of the cast is passed into `bb1` as an argument. If the conversion fails, control is transferred to `bb2`.

An exact cast checks whether the dynamic type is exactly the target type, not any possible subtype of it. The source and target types must be class types.

checked_cast_addr_br

```
sil-terminator ::= 'checked_cast_addr_br'
                    sil-cast-consumption-kind
                    sil-type 'in' sil-operand 'to'
                    sil-stype 'in' sil-operand ','
                    sil-identifier ',' sil-identifier
sil-cast-consumption-kind ::= 'take_always'
sil-cast-consumption-kind ::= 'take_on_success'
sil-cast-consumption-kind ::= 'copy_on_success'
```

```
checked_cast_addr_br take_always $A in %0 : $*@thick A to $B in %2 : $*@thick B, bb1,␣
↪bb2
// $A and $B must be both address types
// bb1 must take a single argument of type $*B
// bb2 must take no arguments
```

Performs a checked indirect conversion from `$A` to `$B`. If the conversion succeeds, control is transferred to `bb1`, and the result of the cast is left in the destination. If the conversion fails, control is transferred to `bb2`.

try_apply

```
sil-terminator ::= 'try_apply' sil-value
                       sil-apply-substitution-list?
                       '(' (sil-value (',' sil-value)*)? ')'
                       ':' sil-type
  'normal' sil-identifier, 'error' sil-identifier

try_apply %0(%1, %2, ...) : $(A, B, ...) -> (R, @error E),
  normal bb1, error bb2
bb1(%3 : R):
bb2(%4 : E):

// Note that the type of the callee '%0' is specified *after* the arguments
// %0 must be of a concrete function type $(A, B, ...) -> (R, @error E)
// %1, %2, etc. must be of the argument types $A, $B, etc.
```

Transfers control to the function specified by `%0`, passing it the given arguments. When `%0` returns, control resumes in either the normal destination (if it returns with `return`) or the error destination (if it returns with `throw`).

`%0` must have a function type with an error result.

The rules on generic substitutions are identical to those of `apply`.

12.10.17 Assertion configuration

To be able to support disabling assertions at compile time there is a builtin `assertion_configuration` function. A call to this function can be replaced at compile time by a constant or can stay opaque.

All calls to the `assert_configuration` function are replaced by the constant propagation pass to the appropriate constant depending on compile time settings. Subsequent passes remove dependent unwanted control flow. Using this mechanism we support conditionally enabling/disabling of code in SIL libraries depending on the assertion configuration selected when the library is linked into user code.

There are three assertion configurations: Debug (0), Release (1) and DisableReplacement (-1).

The optimization flag or a special assert configuration flag determines the value. Depending on the configuration value assertions in the standard library will be executed or not.

The standard library uses this builtin to define an assert that can be disabled at compile time.

```
func assert(...) {
  if (Int32(Builtin.assert_configuration()) == 0) {
    _fatal_error_message(message, ...)
  }
}
```

The `assert_configuration` function application is serialized when we build the standard library (we recognize the `-parse-stdlib` option and don't do the constant replacement but leave the function application to be serialized to sil).

The compiler flag that influences the value of the `assert_configuration` funtion application is the optimization flag: at `-Onone` the application will be replaced by ``Debug`` at higher optimization levels the instruction will be replaced by `Release`. Optionally, the value to use for replacement can be specified with the `-AssertConf` flag which overwrites the value selected by the optimization flag (possible values are `Debug`, `Release`, `DisableReplacement`).

If the call to the `assert_configuration` function stays opaque until IRGen, IRGen will replace the application by the constant representing Debug mode (0). This happens we can build the standard library .dylib. The generate sil will retain the function call but the generated .dylib will contain code with assertions enabled.

CHAPTER 13

Type Checker Design and Implementation

13.1 Purpose

This document describes the design and implementation of the Swift type checker. It is intended for developers who wish to modify, extend, or improve on the type checker, or simply to understand in greater depth how the Swift type system works. Familiarity with the Swift programming language is assumed.

13.2 Approach

The Swift language and its type system incorporate a number of popular language features, including object-oriented programming via classes, function and operator overloading, subtyping, and constrained parametric polymorphism. Swift makes extensive use of type inference, allowing one to omit the types of many variables and expressions. For example:

```
func round(x: Double) -> Int { /* ... */ }
var pi: Double = 3.14159
var three = round(pi) // 'three' has type 'Int'

func identity<T>(x: T) -> T { return x }
var eFloat: Float = -identity(2.71828)  // numeric literal gets type 'Float'
```

Swift's type inference allows type information to flow in two directions. As in most mainstream languages, type information can flow from the leaves of the expression tree (e.g., the expression 'pi', which refers to a double) up to the root (the type of the variable 'three'). However, Swift also allows type information to flow from the context (e.g., the fixed type of the variable 'eFloat') at the root of the expression tree down to the leaves (the type of the numeric literal 2.71828). This bi-directional type inference is common in languages that use ML-like type systems, but is not present in mainstream languages like C++, Java, C#, or Objective-C.

Swift implements bi-directional type inference using a constraint-based type checker that is reminiscent of the classical Hindley-Milner type inference algorithm. The use of a constraint system allows a straightforward, general presentation of language semantics that is decoupled from the actual implementation of the solver. It is expected that the constraints themselves will be relatively stable, while the solver will evolve over time to improve performance and diagnostics.

The Swift language contains a number of features not part of the Hindley-Milner type system, including constrained polymorphic types and function overloading, which complicate the presentation and implementation somewhat. On the other hand, Swift limits the scope of type inference to a single expression or statement, for purely practical reasons: we expect that we can provide better performance and vastly better diagnostics when the problem is limited in scope.

Type checking proceeds in three main stages:

Constraint Generation Given an input expression and (possibly) additional contextual information, generate a set of type constraints that describes the relationships among the types of the various subexpressions. The generated constraints may contain unknown types, represented by type variables, which will be determined by the solver.

Constraint Solving Solve the system of constraints by assigning concrete types to each of the type variables in the constraint system. The constraint solver should provide the most specific solution possible among different alternatives.

Solution Application Given the input expression, the set of type constraints generated from that expression, and the set of assignments of concrete types to each of the type variables, produce a well-typed expression that makes all implicit conversions (and other transformations) explicit and resolves all unknown types and overloads. This step cannot fail.

The following sections describe these three stages of type checking, as well as matters of performance and diagnostics.

13.3 Constraints

A constraint system consists of a set of type constraints. Each type constraint either places a requirement on a single type (e.g., it is an integer literal type) or relates two types (e.g., one is a subtype of the other). The types described in constraints can be any type in the Swift type system including, e.g., builtin types, tuple types, function types, enum/struct/class types, protocol types, and generic types. Additionally, a type can be a type variable `T` (which are typically numbered, `T0`, `T1`, `T2`, etc., and are introduced as needed). Type variables can be used in place of any other type, e.g., a tuple type `(T0, Int, (T0) -> Int)` involving the type variable `T0`.

There are a number of different kinds of constraints used to describe the Swift type system:

Equality An equality constraint requires two types to be identical. For example, the constraint `T0 == T1` effectively ensures that `T0` and `T1` get the same concrete type binding. There are two different flavors of equality constraints:

- Exact equality constraints, or "binding", written `T0 := X` for some type variable `T0` and type `X`, which requires that `T0` be exactly identical to `X`;
- Equality constraints, written `X == Y` for types `X` and `Y`, which require `X` and `Y` to have the same type, ignoring lvalue types in the process. For example, the constraint `T0 == X` would be satisfied by assigning `T0` the type `X` and by assigning `T0` the type `@lvalue X`.

Subtyping A subtype constraint requires the first type to be equivalent to or a subtype of the second. For example, a class type `Dog` is a subtype of a class type `Animal` if `Dog` inherits from `Animal` either directly or indirectly. Subtyping constraints are written `X < Y`.

Conversion A conversion constraint requires that the first type be convertible to the second, which includes subtyping and equality. Additionally, it allows a user-defined conversion function to be called. Conversion constraints are written `X <c Y`, read as `` `X can be converted to `Y` ``.

Construction A construction constraint, written `X <C Y` requires that the second type be a nominal type with a constructor that accepts a value of the first type. For example, the constraint``Int <C String`` is satisfiable because `String` has a constructor that accepts an `Int`.

Member A member constraint `X[.name] == Y` specifies that the first type (`X`) have a member (or an overloaded set of members) with the given name, and that the type of that member be bound to the second type (`Y`). There are two flavors of member constraint: value member constraints, which refer to the member in an expression

context, and type member constraints, which refer to the member in a type context (and therefore can only refer to types).

Conformance A conformance constraint `X conforms to Y` specifies that the first type (''X'') must conform to the protocol `Y`.

Checked cast A constraint describing a checked cast from the first type to the second, i.e., for ''x as T''.

Applicable function An applicable function requires that both types are function types with the same input and output types. It is used when the function type on the left-hand side is being split into its input and output types for function application purposes. Note, that it does not require the type attributes to match.

Overload binding An overload binding constraint binds a type variable by selecting a particular choice from an overload set. Multiple overloads are represented by a disjunction constraint.

Conjunction A constraint that is the conjunction of two or more other constraints. Typically used within a disjunction.

Disjunction A constraint that is the disjunction of two or more constraints. Disjunctions are used to model different decisions that the solver could make, i.e., the sets of overloaded functions from which the solver could choose, or different potential conversions, each of which might resolve in a (different) solution.

Archetype An archetype constraint requires that the constrained type be bound to an archetype. This is a very specific kind of constraint that is only used for calls to operators in protocols.

Class A class constraint requires that the constrained type be bound to a class type.

Self object of protocol An internal-use-only constraint that describes the conformance of a `Self` type to a protocol. It is similar to a conformance constraint but "looser" because it allows a protocol type to be the self object of its own protocol (even when an existential type would not conform to its own protocol).

Dynamic lookup value A constraint that requires that the constrained type be DynamicLookup or an lvalue thereof.

13.3.1 Constraint Generation

The process of constraint generation produces a constraint system that relates the types of the various subexpressions within an expression. Programmatically, constraint generation walks an expression from the leaves up to the root, assigning a type (which often involves type variables) to each subexpression as it goes.

Constraint generation is driven by the syntax of the expression, and each different kind of expression—function application, member access, etc.—generates a specific set of constraints. Here, we enumerate the primary expression kinds in the language and describe the type assigned to the expression and the constraints generated from such as expression. We use `T(a)` to refer to the type assigned to the subexpression `a`. The constraints and types generated from the primary expression kinds are:

Declaration reference An expression that refers to a declaration `x` is assigned the type of a reference to `x`. For example, if `x` is declared as `var x: Int`, the expression `x` is assigned the type `@lvalue Int`. No constraints are generated.

When a name refers to a set of overloaded declarations, the selection of the appropriate declaration is handled by the solver. This particular issue is discussed in the *Overloading* section. Additionally, when the name refers to a generic function or a generic type, the declaration reference may introduce new type variables; see the *Polymorphic Types* section for more information.

Member reference A member reference expression `a.b` is assigned the type `T0` for a fresh type variable `T0`. In addition, the expression generates the value member constraint `T(a).b == T0`. Member references may end up resolving to a member of a nominal type or an element of a tuple; in the latter case, the name (`b`) may either be an identifier or a positional argument (e.g., `1`).

Note that resolution of the member constraint can refer to a set of overloaded declarations; this is described further in the *Overloading* section.

Unresolved member reference An unresolved member reference `.name` refers to a member of a enum type. The enum type is assumed to have a fresh variable type `T0` (since that type can only be known from context), and a value member constraint `T0.name == T1`, for fresh type variable `T1`, captures the fact that it has a member named `name` with some as-yet-unknown type `T1`. The type of the unresolved member reference is `T1`, the type of the member. When the unresolved member reference is actually a call `.name(x)`, the function application is folded into the constraints generated by the unresolved member reference.

Note that the constraint system above actually has insufficient information to determine the type `T0` without additional contextual information. The *Overloading* section describes how the overload-selection mechanism is used to resolve this problem.

Function application A function application `a(b)` generates two constraints. First, the applicable function constraint `T0 -> T1 ==Fn T(a)` (for fresh type variables `T0` and `T1`) captures the rvalue-to-lvalue conversion applied on the function (`a`) and decomposes the function type into its argument and result types. Second, the conversion constraint `T(b) <c T0` captures the requirement that the actual argument type (`b`) be convertible to the argument type of the function. Finally, the expression is given the type `T1`, i.e., the result type of the function.

Construction A type construction‘‘A(b)‘‘, where `A` refers to a type, generates a construction constraint `T(b) <C A`, which requires that `A` have a constructor that accepts `b`. The type of the expression is `A`.

Note that construction and function application use the same syntax. Here, the constraint generator performs a shallow analysis of the type of the “function” argument (`A` or `a`, in the exposition above); if it obviously has metatype type, the expression is considered a coercion/construction rather than a function application. This particular area of the language needs more work.

Subscripting A subscript operation `a[b]` is similar to function application. A value member constraint `T(a).subscript == T0 -> T1` treats the subscript as a function from the key type to the value type, represented by fresh type variables `T0` and `T1`, respectively. The constraint `T(b) <c T0` requires the key argument to be convertible to the key type, and the type of the subscript operation is `T1`.

Literals A literal expression, such as `17`, `1.5`, or `"Hello, world!`, is assigned a fresh type variable `T0`. Additionally, a literal constraint is placed on that type variable depending on the kind of literal, e.g., “`T0` is an integer literal.”

Closures A closure is assigned a function type based on the parameters and return type. When a parameter has no specified type or is positional (`$1`, `$2`, etc.), it is assigned a fresh type variable to capture the type. Similarly, if the return type is omitted, it is assigned a fresh type variable.

When the body of the closure is a single expression, that expression participates in the type checking of its enclosing expression directly. Otherwise, the body of the closure is separately type-checked once the type checking of its context has computed a complete function type.

Array allocation An array allocation `new A[s]` is assigned the type `A[]`. The type checker (separately) checks that `T(s)` is an array bound type.

Address of An address-of expression `&a` always returns an `@inout` type. Therefore, it is assigned the type `@inout T0` for a fresh type variable `T0`. The subtyping constraint `@inout T0 < @lvalue T(a)` captures the requirement that input expression be an lvalue of some type.

Ternary operator A ternary operator‘‘x ? y : z‘‘ generates a number of constraints. The type `T(x)` must conform to the `LogicValue` protocol to determine which branch is taken. Then, a new type variable `T0` is introduced to capture the result type, and the constraints `T(y) <c T0` and `T(z) <c T0` capture the need for both branches of the ternary operator to convert to a common type.

There are a number of other expression kinds within the language; see the constraint generator for their mapping to constraints.

Overloading

Overloading is the process of giving multiple, different definitions to the same name. For example, we might overload a `negate` function to work on both `Int` and `Double` types, e.g.:

```
func negate(x: Int) -> Int { return -x }
func negate(x: Double) -> Double { return -x }
```

Given that there are two definitions of `negate`, what is the type of the declaration reference expression `negate`? If one selects the first overload, the type is `(Int) -> Int`; for the second overload, the type is `(Double) -> Double`. However, constraint generation needs to assign some specific type to the expression, so that its parent expressions can refer to that type.

Overloading in the type checker is modeled by introducing a fresh type variable (call it `T0`) for the type of the reference to an overloaded declaration. Then, a disjunction constraint is introduced, in which each term binds that type variable (via an exact equality constraint) to the type produced by one of the overloads in the overload set. In our negate example, the disjunction is `T0 := (Int) -> Int or T0 := (Double) -> Double`. The constraint solver, discussed in the later section on *Constraint Solving*, explores both possible bindings, and the overloaded reference resolves to whichever binding results in a solution that satisfies all constraints[1].

Overloading can be introduced both by expressions that refer to sets of overloaded declarations and by member constraints that end up resolving to a set of overloaded declarations. One particularly interesting case is the unresolved member reference, e.g., `.name`. As noted in the prior section, this generates the constraint `T0.name == T1`, where `T0` is a fresh type variable that will be bound to the enum type and `T1` is a fresh type variable that will be bound to the type of the selected member. The issue noted in the prior section is that this constraint does not give the solver enough information to determine `T0` without guesswork. However, we note that the type of a enum member actually has a regular structure. For example, consider the `Optional` type:

```
enum Optional<T> {
  case None
  case Some(T)
}
```

The type of `Optional<T>.Vone` is `Optional<T>`, while the type of `Optional<T>.Some` is `(T) -> Optional<T>`. In fact, the type of a enum element can have one of two forms: it can be `T0`, for a enum element that has no extra data, or it can be `T2 -> T0`, where `T2` is the data associated with the enum element. For the latter case, the actual arguments are parsed as part of the unresolved member reference, so that a function application constraint describes their conversion to the input tyoe `T2`.

Polymorphic Types

The Swift language includes generics, a system of constrained parameter polymorphism that enables polymorphic types and functions. For example, one can implement a `min` function as, e.g.,:

```
func min<T : Comparable>(x: T, y: T) -> T {
  if y < x { return y }
  return x
}
```

Here, `T` is a generic parameter that can be replaced with any concrete type, so long as that type conforms to the protocol `Comparable`. The type of `min` is (internally) written as `<T : Comparable> (x: T, y: T) -> T`, which can be read as "for all `T`, where `T` conforms to `Comparable`, the type of the function is `(x: T, y: T) -> T`. Different uses of the `min` function may have different bindings for the generic parameter``T``.

[1] It is possible that both overloads will result in a solution, in which case the solutions will be ranked based on the rules discussed in the section *Comparing Solutions*.

When the constraint generator encounters a reference to a generic function, it immediately replaces each of the generic parameters within the function type with a fresh type variable, introduces constraints on that type variable to match the constraints listed in the generic function, and produces a monomorphic function type based on the newly-generated type variables. For example, the first occurrence of the declaration reference expression `min` would result in a type `(x : T0, y : T0) -> T0`, where `T0` is a fresh type variable, as well as the subtype constraint `T0 < Comparable`, which expresses protocol conformance. The next occurrence of the declaration reference expression `min` would produce the type `(x : T1, y : T1) -> T1`, where `T1` is a fresh type variable (and therefore distinct from `T0`), and so on. This replacement process is referred to as "opening" the generic function type, and is a fairly simple (but effective) way to model the use of polymorphic functions within the constraint system without complicating the solver. Note that this immediate opening of generic function types is only valid because Swift does not support first-class polymorphic functions, e.g., one cannot declare a variable of type `<T> T -> T`.

Uses of generic types are also immediately opened by the constraint solver. For example, consider the following generic dictionary type:

```
class Dictionary<Key : Hashable, Value> {
  // ...
}
```

When the constraint solver encounters the expression '' Dictionary()'', it opens up the type `Dictionary`—which has not been provided with any specific generic arguments—to the type `Dictionary<T0, T1>`, for fresh type variables `T0` and `T1`, and introduces the constraint `T0 conforms to Hashable`. This allows the actual key and value types of the dictionary to be determined by the context of the expression. As noted above for first-class polymorphic functions, this immediate opening is valid because an unbound generic type, i.e., one that does not have specified generic arguments, cannot be used except where the generic arguments can be inferred.

13.4 Constraint Solving

The primary purpose of the constraint solver is to take a given set of constraints and determine the most specific type binding for each of the type variables in the constraint system. As part of this determination, the constraint solver also resolves overloaded declaration references by selecting one of the overloads.

Solving the constraint systems generated by the Swift language can, in the worst case, require exponential time. Even the classic Hindley-Milner type inference algorithm requires exponential time, and the Swift type system introduces additional complications, especially overload resolution. However, the problem size for any particular expression is still fairly small, and the constraint solver can employ a number of tricks to improve performance. The *Performance* section describes some tricks that have been implemented or are planned, and it is expected that the solver will be extended with additional tricks going forward.

This section will focus on the basic ideas behind the design of the solver, as well as the type rules that it applies.

13.4.1 Simplification

The constraint generation process introduces a number of constraints that can be immediately solved, either directly (because the solution is obvious and trivial) or by breaking the constraint down into a number of smaller constraints. This process, referred to as *simplification*, canonicalizes a constraint system for later stages of constraint solving. It is also re-invoked each time the constraint solver makes a guess (at resolving an overload or binding a type variable, for example), because each such guess often leads to other simplifications. When all type variables and overloads have been resolved, simplification terminates the constraint solving process either by detecting a trivial constraint that is not satisfied (hence, this is not a proper solution) or by reducing the set of constraints down to only simple constraints that are trivially satisfied.

The simplification process breaks down constraints into simpler constraints, and each different kind of constraint is handled by different rules based on the Swift type system. The constraints fall intofive categories: relational con-

straints, member constraints, type properties, conjunctions, and disjunctions. Only the first three kinds of constraints have interesting simplification rules, and are discussed in the following sections.

Relational Constraints

Relational constraints describe a relationship between two types. This category covers the equality, subtyping, and conversion constraints, and provides the most common simplifications. The simplification of relationship constraints proceeds by comparing the structure of the two types and applying the typing rules of the Swift language to generate additional constraints. For example, if the constraint is a conversion constraint:

```
A -> B <c C -> D
```

then both types are function types, and we can break down this constraint into two smaller constraints `C < A` and `B < D` by applying the conversion rule for function types. Similarly, one can destroy all of the various type constructors—tuple types, generic type specializations, lvalue types, etc.—to produce simpler requirements, based on the type rules of the language[2].

Relational constraints involving a type variable on one or both sides generally cannot be solved directly. Rather, these constraints inform the solving process later by providing possible type bindings, described in the *Type Variable Bindings* section. The exception is an equality constraint between two type variables, e.g., `T0 == T1`. These constraints are simplified by unifying the equivalence classes of `T0` and `T1` (using a basic union-find algorithm), such that the solver need only determine a binding for one of the type variables (and the other gets the same binding).

Member Constraints

Member constraints specify that a certain type has a member of a given name and provide a binding for the type of that member. A member constraint `A.member == B` can be simplified when the type of `A` is determined to be a nominal or tuple type, in which case name lookup can resolve the member name to an actual declaration. That declaration has some type `C`, so the member constraint is simplified to the exact equality constraint‘‘B := C‘‘.

The member name may refer to a set of overloaded declarations. In this case, the type `C` is a fresh type variable (call it `T0`). A disjunction constraint is introduced, each term of which new overload set binds a different declaration’s type to `T0`, as described in the section on *Overloading*.

The kind of member constraint—type or value—also affects the declaration type `C`. A type constraint can only refer to member types, and `C` will be the declared type of the named member. A value constraint, on the other hand, can refer to either a type or a value, and `C` is the type of a reference to that entity. For a reference to a type, `C` will be a metatype of the declared type.

13.4.2 Strategies

The basic approach to constraint solving is to simplify the constraints until they can no longer be simplified, then produce (and check) educated guesses about which declaration from an overload set should be selected or what concrete type should be bound to a given type variable. Each guess is tested as an assumption, possibly with other guesses, until the solver either arrives at a solution or concludes that the guess was incorrect.

Within the implementation, each guess is modeled as an assumption within a new solver scope. The solver scope inherits all of the constraints, overload selections, and type variable bindings of its parent solver scope, then adds one more guess. As such, the solution space explored by the solver can be viewed as a tree, where the top-most node is

[2] As of the time of this writing, the type rules of Swift have not specifically been documented outside of the source code. The constraints-based type checker contains a function `matchTypes` that documents and implements each of these rules. A future revision of this document will provide a more readily-accessible version.

the constraint system generated directly from the expression. The leaves of the tree are either solutions to the type-checking problem (where all constraints have been simplified away) or represent sets of assumptions that do not lead to a solution.

The following sections describe the techniques used by the solver to produce derived constraint systems that explore the solution space.

Overload Selection

Overload selection is the simplest way to make an assumption. For an overload set that introduced a disjunction constraint `T0 := A1 or T0 := A2 or ... or T0 := AN` into the constraint system, each term in the disjunction will be visited separately. Each solver state binds the type variable `T0` and explores whether the selected overload leads to a suitable solution.

Type Variable Bindings

A second way in which the solver makes assumptions is to guess at the concrete type to which a given type variable should be bound. That type binding is then introduced in a new, derived constraint system to determine if the binding is feasible.

The solver does not conjure concrete type bindings from nothing, nor does it perform an exhaustive search. Rather, it uses the constraints placed on that type variable to produce potential candidate types. There are several strategies employed by the solver.

Meets and Joins

A given type variable `T0` often has relational constraints placed on it that relate it to concrete types, e.g., `T0 <c Int` or `Float <c T0`. In these cases, we can use the concrete types as a starting point to make educated guesses for the type `T0`.

To determine an appropriate guess, the relational constraints placed on the type variable are categorized. Given a relational constraint of the form `T0 <? A` (where `<?` is one of `<`, `<t`, or `<c`), where `A` is some concrete type, `A` is said to be "above" `T0`. Similarly, given a constraint of the form `B <? T0` for a concrete type `B`, `B` is said to be "below" `T0`. The above/below terminologies comes from a visualization of the lattice of types formed by the conversion relationship, e.g., there is an edge `A -> B` in the latter if `A` is convertible to `B`. `B` would therefore be higher in the lattice than `A`, and the topmost element of the lattice is the element to which all types can be converted, `protocol<>` (often called "top").

The concrete types "above" and "below" a given type variable provide bounds on the possible concrete types that can be assigned to that type variable. The solver computes[3] the join of the types "below" the type variable, i.e., the most specific (lowest) type to which all of the types "below" can be converted, and uses that join as a starting guess.

Supertype Fallback

The join of the "below" types computed as a starting point may be too specific, due to constraints that involve the type variable but weren't simple enough to consider as part of the join. To cope with such cases, if no solution can be found with the join of the "below" types, the solver creates a new set of derived constraint systems with weaker assumptions, corresponding to each of the types that the join is directly convertible to. For example, if the join was some class `Derived`, the supertype fallback would then try the class `Base` from which `Derived` directly inherits.

[3] More accurately, as of this writing, "will compute". The solver doesn't current compute meets and joins properly. Rather, it arbitrarily picks one of the constraints "below" to start with.

This fallback process continues until the types produced are no longer convertible to the meet of types "above" the type variable, i.e., the least specific (highest) type from which all of the types "above" the type variable can be converted[4].

Default Literal Types

If a type variable is bound by a conformance constraint to one of the literal protocols, "`T0` conforms to `IntegerLiteralConvertible`", then the constraint solver will guess that the type variable can be bound to the default literal type for that protocol. For example, `T0` would get the default integer literal type `Int`, allowing one to type-check expressions with too little type information to determine the types of these literals, e.g., `-1`.

13.4.3 Comparing Solutions

The solver explores a potentially large solution space, and it is possible that it will find multiple solutions to the constraint system as given. Such cases are not necessarily ambiguities, because the solver can then compare the solutions to to determine whether one of the solutions is better than all of the others. To do so, it computes a "score" for each solution based on a number of factors:

- How many user-defined conversions were applied.
- How many non-trivial function conversions were applied.
- How many literals were given "non-default" types.

Solutions with smaller scores are considered better solutions. When two solutions have the same score, the type variables and overload choices of the two systems are compared to produce a relative score:

- If the two solutions have selected different type variable bindings for a type variable where a "more specific" type variable is a better match, and one of the type variable bindings is a subtype of the other, the solution with the subtype earns +1.
- If an overload set has different selected overloads in the two soluions, the overloads are compared. If the type of the overload picked in one solution is a subtype of the type of the overload picked in the other solution, then first solution earns +1.

The solution with the greater relative score is considered to be better than the other solution.

13.5 Solution Application

Once the solver has produced a solution to the constraint system, that solution must be applied to the original expression to produce a fully type-checked expression that makes all implicit conversions and resolved overloads explicit. This application process walks the expression tree from the leaves to the root, rewriting each expression node based on the kind of expression:

Declaration references Declaration references are rewritten with the precise type of the declaraion as referenced. For overloaded declaration references, the `Overload*Expr` node is replaced with a simple declaration reference expression. For references to polymorphic functions or members of generic types, a `SpecializeExpr` node is introduced to provide substitutions for all of the generic parameters.

Member references References to members are similar to declaration references. However, they have the added constraint that the base expression needs to be a reference. Therefore, an rvalue of non-reference type will be materialized to produce the necessary reference.

[4] Again, as of this writing, the solver doesn't actually compute meets and joins, so the solver continues until it runs out of supertypes to enumerate.

Literals Literals are converted to the appropriate literal type, which typically involves introducing calls to the witnesses for the appropriate literal protocols.

Closures Since the closure has acquired a complete function type, the body of the closure is type-checked with that complete function type.

The solution application step cannot fail for any type checking rule modeled by the constraint system. However, there are some failures that are intentionally left to the solution application phase, such as a postfix '!' applied to a non-optional type.

13.5.1 Locators

During constraint generation and solving, numerous constraints are created, broken apart, and solved. During constraint application as well as during diagnostics emission, it is important to track the relationship between the constraints and the actual expressions from which they originally came. For example, consider the following type checking problem:

```
struct X {
  // user-defined conversions
  func [conversion] __conversion () -> String { /* ... */ }
  func [conversion] __conversion () -> Int { /* ... */ }
}

func f(i : Int, s : String) { }

var x : X
f(10.5, x)
```

This constraint system generates the constraints "`T(f)` ==Fn `T0 -> T1`" (for fresh variables `T0` and `T1`), "`(T2, X)` <c `T0`" (for fresh variable `T2`) and "`T2 conforms to ``FloatLiteralConvertible`". As part of the solution, after `T0` is replaced with `(i : Int, s : String)`, the second of these constraints is broken down into "`T2 <c ``Int`" and "`X` <c `String`". These two constraints are interesting for different reasons: the first will fail, because `Int` does not conform to `FloatLiteralConvertible`. The second will succeed by selecting one of the (overloaded) conversion functions.

In both of these cases, we need to map the actual constraint of interest back to the expressions they refer to. In the first case, we want to report not only that the failure occurred because `Int` is not `FloatLiteralConvertible`, but we also want to point out where the `Int` type actually came from, i.e., in the parameter. In the second case, we want to determine which of the overloaded conversion functions was selected to perform the conversion, so that conversion function can be called by constraint application if all else succeeds.

Locators address both issues by tracking the location and derivation of constraints. Each locator is anchored at a specific expression, i.e., the function application `f(10.5, x)`, and contains a path of zero or more derivation steps from that anchor. For example, the "`T(f)` ==Fn `T0 -> T1`" constraint has a locator that is anchored at the function application and a path with the "apply function" derivation step, meaning that this is the function being applied. Similarly, the "`(T2, X)` <c `T0` constraint has a locator anchored at the function application and a path with the "apply argument" derivation step, meaning that this is the argument to the function.

When constraints are simplified, the resulting constraints have locators with longer paths. For example, when a conversion constraint between two tuples is simplified conversion constraints between the corresponding tuple elements, the resulting locators refer to specific elements. For example, the `T2 <c Int` constraint will be anchored at the function application (still), and have two derivation steps in its path: the "apply function" derivation step from its parent constraint followed by the "tuple element 0" constraint that refers to this specific tuple element. Similarly, the `X <c String` constraint will have the same locator, but with "tuple element 1" rather than "tuple element 0". The `ConstraintLocator` type in the constraint solver has a number of different derivation step kinds (called "path elements" in the source) that describe the various ways in which larger constraints can be broken down into smaller ones.

Overload Choices

Whenever the solver creates a new overload set, that overload set is associated with a particular locator. Continuing the example from the parent section, the solver will create an overload set containing the two user-defined conversions. This overload set is created while simplifying the constraint `X <c String`, so it uses the locator from that constraint extended by a "conversion member" derivation step. The complete locator for this overload set is, therefore:

```
function application -> apply argument -> tuple element #1 -> conversion member
```

When the solver selects a particular overload from the overload set, it records the selected overload based on the locator of the overload set. When it comes time to perform constraint application, the locator is recreated based on context (as the bottom-up traversal walks the expressions to rewrite them with their final types) and used to find the appropriate conversion to call. The same mechanism is used to select the appropriate overload when an expression refers directly to an overloaded function. Additionally, when comparing two solutions to the same constraint system, overload sets present in both solutions can be found by comparing the locators for each of the overload choices made in each solution. Naturally, all of these operations require locators to be uniqued, which occurs in the constraint system itself.

Simplifying Locators

Locators provide the derivation of location information that follows the path of the solver, and can be used to query and recover the important decisions made by the solver. However, the locators determined by the solver may not directly refer to the most specific expression for the purposes of identifying the corresponding source location. For example, the failed constraint "`Int` conforms to `FloatLiteralConvertible`" can most specifically by centered on the floating-point literal `10.5`, but its locator is:

```
function application -> apply argument -> tuple element #0
```

The process of locator simplification maps a locator to its most specific expression. Essentially, it starts at the anchor of the locator (in this case, the application `f(10.5, x)`) and then walks the path, matching derivation steps to subexpressions. The "function application" derivation step extracts the argument (`(10.5, x)`). Then, the "tuple element #0" derivation extracts the tuple element 0 subexpression, `10.5`, at which point we have traversed the entire path and now have the most specific expression for source-location purposes.

Simplification does not always exhaust the complete path. For example, consider a slight modification to our example, so that the argument to `f` is provided by another call, we get a different result entirely:

```
func f(i : Int, s : String) { }
func g() -> (f : Float, x : X) { }

f(g())
```

Here, the failing constraint is `Float <c Int`, with the same locator:

```
function application -> apply argument -> tuple element #0
```

When we simplify this locator, we start with `f(g())`. The "apply argument" derivation step takes us to the argument expression `g()`. Here, however, there is no subexpression for the first tuple element of `g()`, because it's simple part of the tuple returned from `g`. At this point, simplification ceases, and creates the simplified locator:

```
function application of g -> tuple element #0
```

13.6 Performance

The performance of the type checker is dependent on a number of factors, but the chief concerns are the size of the solution space (which is exponential in the worst case) and the effectiveness of the solver in exploring that solution space. This section describes some of the techniques used to improve solver performance, many of which can doubtless be improved.

13.6.1 Constraint Graph

The constraint graph describes the relationships among type variables in the constraint system. Each vertex in the constraint graph corresponds to a single type variable. The edges of the graph correspond to constraints in the constraint system, relating sets of type variables together. Technically, this makes the constraint graph a *multigraph*, although the internal representation is more akin to a graph with multiple kinds of edges: each vertex (node) tracks the set of constraints that mention the given type variable as well as the set of type variables that are adjacent to this type variable. A vertex also includes information about the equivalence class corresponding to a given type variable (when type variables have been merged) or the binding of a type variable to a specific type.

The constraint graph is critical to a number of solver optimizations. For example, it is used to compute the connected components within the constraint graph, so that each connected component can be solved independently. The partial results from all of the connected components are then combined into a complete solution. Additionally, the constraint graph is used to direct simplification, described below.

13.6.2 Simplification Worklist

When the solver has attempted a type variable binding, that binding often leads to additional simplifications in the constraint system. The solver will query the constraint graph to determine which constraints mention the type variable and will place those constraints onto the simplification worklist. If those constraints can be simplified further, it may lead to additional type variable bindings, which in turn adds more constraints to the worklist. Once the worklist is exhausted, simplification has completed. The use of the worklist eliminates the need to reprocess constraints that could not have changed because the type variables they mention have not changed.

13.6.3 Solver Scopes

The solver proceeds through the solution space in a depth-first manner. Whenever the solver is about to make a guess—such as a speculative type variable binding or the selection of a term from a disjunction—it introduces a new solver scope to capture the results of that assumption. Subsequent solver scopes are nested as the solver builds up a set of assumptions, eventually leading to either a solution or an error. When a solution is found, the stack of solver scopes contains all of the assumptions needed to produce that solution, and is saved in a separate solution data structure.

The solver scopes themselves are designed to be fairly cheap to create and destroy. To support this, all of the major data structures used by the constraint solver have reversible operations, allowing the solver to easily backtrack. For example, the addition of a constraint to the constraint graph can be reversed by removing that same constraint. The constraint graph tracks all such additions in a stack: pushing a new solver scope stores a marker to the current top of the stack, and popping that solver scope reverses all of the operations on that stack until it hits the marker.

13.6.4 Online Scoring

As the solver evaluates potential solutions, it keeps track of the score of the current solution and of the best complete solution found thus far. If the score of the current solution is ever greater than that of the best complete solution, it abandons the current solution and backtracks to continue its search.

The solver makes some attempt at evaluating cheaper solutions before more expensive solutions. For example, it will prefer to try normal conversions before user-defined conversions, prefer the "default" literal types over other literal types, and prefer cheaper conversions to more expensive conversions. However, some of the rules are fairly ad hoc, and could benefit from more study.

13.6.5 Arena Memory Management

Each constraint system introduces its own memory allocation arena, making allocations cheap and deallocation essentially free. The allocation arena extends all the way into the AST context, so that types composed of type variables (e.g., `T0 -> T1`) will be allocated within the constraint system's arena rather than the permanent arena. Most data structures involved in constraint solving use this same arena.

13.7 Diagnostics

The diagnostics produced by the type checker are currently terrible. We plan to do something about this, eventually. We also believe that we can implement some heroics, such as spell-checking that takes into account the surrounding expression to only provide well-typed suggestions.

CHAPTER 14

Debugging the Swift Compiler

Contents

- *Debugging the Swift Compiler*
  - *Abstract*
  - *Printing the Intermediate Representations*
    - *Debugging on SIL Level*
      - *Options for Dumping the SIL*
      - *Dumping the SIL and other Data in LLDB*
      - *Other Utilities*
      - *Using Breakpoints*
      - *LLDB Scripts*
  - *Debugging Swift Executables*
    - *Determining the mangled name of a function in LLDB*

14.1 Abstract

This document contains some useful information for debugging the swift compiler and swift compiler output.

14.2 Printing the Intermediate Representations

The most important thing when debugging the compiler is to examine the IR. Here is how to dump the IR after the main phases of the swift compiler (assuming you are compiling with optimizations enabled):

1. **Parser**. To print the AST after parsing:

```
swiftc -dump-ast -O file.swift
```

2. **SILGen**. To print the SIL immediately after SILGen:

```
swiftc -emit-silgen -O file.swift
```

3. **Mandatory SIL passes**. To print the SIL after the mandatory passes:

```
swiftc -emit-sil -Onone file.swift
```

   Well, this is not quite true, because the compiler is running some passes for -Onone after the mandatory passes, too. But for most purposes you will get what you want to see.

1. **Performance SIL passes**. To print the SIL after the complete SIL oprimization pipeline:

```
swiftc -emit-sil -O file-swift
```

2. **IRGen**. To print the LLVM IR after IR generation:

```
swiftc -emit-ir -Xfrontend -disable-llvm-optzns -O file.swift
```

4. **LLVM passes**. To print the LLVM IR afer LLVM passes:

```
swiftc -emit-ir -O file.swift
```

5. **Code generation**. To print the final generated code:

```
swiftc -S -O file.swift
```

Compilation stops at the phase where you print the output. So if you want to print the SIL *and* the LLVM IR, you have to run the compiler twice. The output of all these dump options (except `-dump-ast`) can be redirected with an additional `-o <file>` option.

14.2.1 Debugging on SIL Level

Options for Dumping the SIL

Often it is not sufficient to dump the SIL at the begin or end of the optimization pipeline. The SILPassManager supports useful options to dump the SIL also between pass runs.

The option `-Xllvm -sil-print-all` dumps the whole SIL module after all passes. Although it prints only functions which were changed by a pass, the output can get *very* large.

It is useful if you identified a problem in the final SIL and you want to check which pass did introduce the wrong SIL.

There are several other options available, e.g. to filter the output by function names (`-Xllvm -sil-print-only-function/s`) or by pass names (`-Xllvm -sil-print-before/after/around`). For details see `PassManager.cpp`.

Dumping the SIL and other Data in LLDB

When debugging the swift compiler with LLDB (or Xcode, of course), there is even a more powerful way to examine the data in the compiler, e.g. the SIL. Following LLVM's dump() convention, many SIL classes (as well as AST

classes) provide a dump() function. You can call the dump function with LLDB's `expression --` or `print` or `p` command.

For example, to examine a SIL instruction:

```
(lldb) p Inst->dump()
%12 = struct_extract %10 : $UnsafeMutablePointer<X>, #UnsafeMutablePointer._rawValue /
↪/ user: %13
```

To dump a whole function at the beginning of a function pass:

```
(lldb) p getFunction()->dump()
```

SIL modules and even functions can get very large. Often it is more convenient to dump their contents into a file and open the file in a separate editor. This can be done with:

```
(lldb) p getFunction()->dump("myfunction.sil")
```

You can also dump the CFG (control flow graph) of a function:

```
(lldb) p Func->viewCFG()
```

This opens a preview window containing the CFG of the function. To continue debugging press <CTRL>-C on the LLDB prompt. Note that this only works in Xcode if the PATH variable in the scheme's environment setting contains the path to the dot tool.

Other Utilities

To view the CFG of a function (or code region) in a SIL file, you can use the script `swift/utils/viewcfg`. It also works for LLVM IR files. The script reads the SIL (or LLVM IR) code from stdin and displays the dot graph file. Note: .dot files should be associated with the Graphviz app.

Using Breakpoints

LLDB has very powerful breakpoints, which can be utilized in many ways to debug the compiler and swift executables. The examples in this section show the LLDB command lines. In Xcode you can set the breakpoint properties by clicking 'Edit breakpoint'.

Let's start with a simple example: sometimes you see a function in the SIL output and you want to know where the function was created in the compiler. In this case you can set a conditional breakpoint in SILFunction constructor and check for the function name in the breakpoint condition:

```
(lldb) br set -c 'hasName("_TFC3nix1Xd")' -f SILFunction.cpp -l 91
```

Sometimes you want to know which optimization does insert, remove or move a certain instruction. To find out, set a breakpoint in `ilist_traits<SILInstruction>::addNodeToList` or `ilist_traits<SILInstruction>::removeNodeFromList`, which are defined in `SILInstruction.cpp`. The following command sets a breakpoint which stops if a `strong_retain` instruction is removed:

```
(lldb) br set -c 'I->getKind() == ValueKind::StrongRetainInst' -f SILInstruction.cpp -
↪l 63
```

The condition can be made more precise e.g. by also testing in which function this happens:

```
(lldb) br set -c 'I->getKind() == ValueKind::StrongRetainInst &&
          I->getFunction()->hasName("_TFC3nix1Xd")'
          -f SILInstruction.cpp -l 63
```

Let's assume the breakpoint hits somewhere in the middle of compiling a large file. This is the point where the problem appears. But often you want to break a little bit earlier, e.g. at the entrance of the optimization's `run` function.

To achieve this, set another breakpoint and add breakpoint commands:

```
(lldb) br set -n GlobalARCOpts::run
Breakpoint 2
(lldb) br com add 2
> p int $n = $n + 1
> c
> DONE
```

Run the program (this can take quite a bit longer than before). When the first breakpoint hits see what value $n has:

```
(lldb) p $n
(int) $n = 5
```

Now remove the breakpoint commands from the second breakpoint (or create a new one) and set the ignore count to $n minus one:

```
(lldb) br delete 2
(lldb) br set -i 4 -n GlobalARCOpts::run
```

Run your program again and the breakpoint hits just before the first breakpoint.

Another method for accomplishing the same task is to set the ignore count of the breakpoint to a large number, i.e.:

```
(lldb) br set -i 9999999 -n GlobalARCOpts::run
```

Then whenever the debugger stops next time (due to hitting another breakpoint/crash/assert) you can list the current breakpoints:

```
(lldb) br list
1: name = 'GlobalARCOpts::run', locations = 1, resolved = 1, hit count = 85 Options:␣
↪ignore: 1 enabled
```

which will then show you the number of times that each breakpoint was hit. In this case, we know that `GlobalARCOpts::run` was hit 85 times. So, now we know to ignore swift_getGenericMetadata 84 times, i.e.:

```
(lldb) br set -i 84 -n GlobalARCOpts::run
```

LLDB Scripts

LLDB has powerful capabilities of scripting in python among other languages. An often overlooked, but very useful technique is the -s command to lldb. This essentially acts as a pseudo-stdin of commands that lldb will read commands from. Each time lldb hits a stopping point (i.e. a breakpoint or a crash/assert), it will run the earliest command that has not been run yet. As an example of this consider the following script (which without any loss of generality will be called test.lldb):

```
env DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib
break set -n swift_getGenericMetadata
break mod 1 -i 83
```

```
process launch -- --stdlib-unittest-in-process --stdlib-unittest-filter
↪"DefaultedForwardMutableCollection<OpaqueValue<Int>>.Type.subscript(_: Range)/Set/
↪semantics"
break set -l 224
c
expr pattern->CreateFunction
break set -a $0
c
dis -f
```

TODO: Change this example to apply to the swift compiler instead of to the stdlib unittests.

Then by running `lldb test -s test.lldb`, lldb will:

1. Enable guard malloc.
2. Set a break point on swift_getGenericMetadata and set it to be ignored for 83 hits.
3. Launch the application and stop at swift_getGenericMetadata after 83 hits have been ignored.
4. In the same file as swift_getGenericMetadata introduce a new breakpoint at line 224 and continue.
5. When we break at line 224 in that file, evaluate an expression pointer.
6. Set a breakpoint at the address of the expression pointer and continue.
7. When we hit the breakpoint set at the function pointer's address, disassemble the function that the function pointer was passed to.

Using LLDB scripts can enable one to use complex debugger workflows without needing to retype the various commands perfectly everytime.

14.3 Debugging Swift Executables

One can use the previous tips for debugging the swift compiler with swift executables as well. Here are some additional useful techniques that one can use in Swift executables.

14.3.1 Determining the mangled name of a function in LLDB

One problem that often comes up when debugging swift code in LLDB is that LLDB shows the demangled name instead of the mangled name. This can lead to mistakes where due to the length of the mangled names one will look at the wrong function. Using the following command, one can find the mangled name of the function in the current frame:

```
(lldb) image lookup -va $pc
Address: CollectionType3[0x0000000100004db0] (CollectionType3.__TEXT.__text + 16000)
Summary: CollectionType3`ext.CollectionType3.CollectionType3.MutableCollectionType2<A␣
↪where A: CollectionType3.MutableCollectionType2>.(subscript.materializeForSet :␣
↪(Swift.Range<A.Index>) -> Swift.MutableSlice<A>).(closure #1)
Module: file = "/Volumes/Files/work/solon/build/build-swift/validation-test-macosx-
↪x86_64/stdlib/Output/CollectionType.swift.gyb.tmp/CollectionType3", arch = "x86_64"
Symbol: id = {0x0000008c}, range = [0x0000000100004db0-0x00000001000056f0), name="ext.
↪CollectionType3.CollectionType3.MutableCollectionType2<A where A: CollectionType3.
↪MutableCollectionType2>.(subscript.materializeForSet : (Swift.Range<A.Index>) ->␣
↪Swift.MutableSlice<A>).(closure #1)", mangled="_TFFeRq_
↪15CollectionType322MutableCollectionType2_S_S0_m9subscriptFGVs5Rangeqq_
↪s16MutableIndexable5Index_GVs12MutableSliceq__U_FTBpRBBRQPS0_MS4__T_"
```