KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Glyco: A CHERI-RISC-V nanopass compiler for experimenting with capability-based security features

Constantino Tsarouhas

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Veilige software

Promotor:
Prof. dr. Dominique Devriese

Assessoren:
Prof. dr. Bart Jacobs
Dr. ir. Koen Yskout

Begeleiders:
Ir. Sander Huyghebaert
Dr. Steven Keuchel

Academiejaar 2021 – 2022

Preface

Writing this the day before submitting my thesis text, it is been 620 days since I had my first lecture of my master's degree. My academic life was fun, interesting, yet incredibly serpentine. 10 years ago I would not have imagined ever arriving at this stage, so it is my honour and pleasure to be able to submit this text for the fulfilment of the graduation requirements of the *Master in de ingenieurswetenschappen: computerwetenschappen*.

This thesis deals with two (relatively orthogonal) disciplines in computer science, namely compiler design and capability machines, which align nicely with my interests in compilers, formal systems, and software security. Whilst researching the literature for my thesis, I discovered modern methods that I did not know existed, and at the end of this long road, I am personally convinced they will be increasingly part of modern compiler designs and modern system architectures.

I would like to express my sincere gratitude to my two mentors, Steven Keuchel and Sander Huyghebaert, who guided me almost every week through what is my first large academic research project, as well as my supervisor, Dominique Devriese, for the interesting thesis topic and his helpful advice. I would also like to extend my gratitude to all my friends and colleagues for their support past years. Last but not least, my utmost gratitude goes to my mother Stamatia and aunt Chrisoula who supported me throughout my whole life and without whose support I certainly would not have had any chance at pursuing an academical degree.

I hope you enjoy reading this text as much as I enjoyed writing it.

```
hope(enjoyment, equals: maximum)
```

Constantino Tsarouhas
Brussels, 7 June 2022

Contents

Abstract

CHERI-RISC-V is an extension of the RISC-V instruction set architecture with support for capabilities, a kind of pointer providing authority over a region of memory and for a set for operations. Capabilities allow for fine-grained memory protection and compartmentalisation but also form the basis of several proposed secure calling conventions which provide local state encapsulation and well-bracketed control flow, two security properties that ensure that call frames cannot be accessed from other procedures and enforce the "return-to-caller" principle.

Researchers in the field painstakingly forked the LLVM compiler suite to include support for CHERI-RISC-V targets but found experimenting with capabilities and capability-based security features arduous given that LLVM has a massive codebase and is designed for a broad set of architectures. This thesis explores a design & implementation of a new nanopass compiler for CHERI-RISC-V targets we call *Glyco*. A nanopass transforms a program from one language to another language; a nanopass compiler transforms a program in a source language throughout numerous languages and nanopasses to a target language, or in the case of Glyco, CHERI-RISC-V assembly.

This thesis begins with a basic implementation of Glyco with almost no capability-based security features and gradually extends it with security features. The first feature is a variation on a proposed secure calling convention we call *GHSCC* that guarantees local state encapsulation and a variation of well-bracketed control flow we call *unrepeatable return* that ensures return capabilities can only be used once. We then compare this new calling convention in terms of runtime and memory overhead against a more traditional one using a few test programs. The compiler is then extended with two new features we call *sealed objects* and *sealed closures*. A sealed object's local state can only be accessed from within a method whereas a sealed closure's saved environment can only be accessed from within the closure body. We evaluate the nanopass approach by measuring each new feature's impact on the codebase.

Samenvatting

CHERI-RISC-V breidt de RISC-V-processorarchitectuur uit met ondersteuning voor *capability's*, een soort pointer die een bepaalde verzameling van machtigingen (zoals lezen of schrijven) voor een geheugengebied draagt. Met capability's kan software geheugenbescherming op kleinschalig niveau aanbieden en geheugen compartimentaliseren. Capability's liggen ook ten grondslag van veilige calling conventions, functieoproepafspraken die veiligheidseigenschappen zoals *local state encapsulation* en *well-bracketed control flow* garanderen. Local state encapsulation garandeert dat de interne toestand van een oproep enkel toegankelijk is binnen de functie. Well-bracketed control flow verzekert dat functies enkel naar hun oproeper kunnen terugkeren.

Onderzoekers hebben de LLVM-compilersuite zorgvuldig aan de CHERI-RISC-V-architectuur aangepast maar hebben moeite ondervonden om met capability's en veiligheidsfuncties gebouwd op capability's te experimenteren. LLVM bestaat namelijk uit een aanzienlijke hoeveelheid broncode en is ontworpen voor een brede reeks aan computerarchitecturen. Dit eindwerk onderzoekt daarom het ontwerp en implementatie van een nieuwe nanopasscompiler voor CHERI-RISC-V-systemen die we *Glyco* noemen. Een nanopass is een programmatransformatie (of vertaling) van één taal naar een andere; een nanopasscompiler transformeert (vertaalt) een programma van een brontaal via een reeks talen naar een doeltaal, de CHERI-RISC-V-assembleertaal in het geval van Glyco.

Deze verhandeling begint met een basisontwerp en -implementatie van Glyco met quasi geen veiligheidsfuncties gebaseerd op capability's en breidt dit ontwerp dan geleidelijk uit met nieuwe veiligheidsfuncties. De eerste functie is een variatie op een veilige calling convention voorgesteld in de literatuur die we *GHSCC* noemen. Deze calling convention biedt local state encapsulation en een variatie op well-bracketed control flow genaamd *eenmalige terugkeer* (*unrepeatable return*). Deze laatste eigenschap verzekert een oproeper dat een opgeroepen (of andere) functie hoogstens één keer kan terugkeren naar de oproeper. We vergelijken vervolgens de nieuwe calling convention met een traditionele in termen van uitvoeringstijd en geheugengebruik. Vervolgens voegen we twee functies aan de compiler toe die we *verzegelde objecten* (*sealed objects*) en *verzegelde closures* (*sealed closures*) noemen. De interne toestand van een verzegeld object is enkel toegankelijk in methoden van het object, terwijl de opgeslagen omgeving van een verzegelde closure enkel bereikbaar is in het lichaam van de closure. Tot slot evalueren we de nanopassmethodologie door de impact van elke nieuwe functie op de broncode te meten.

Chapter 1

Introduction

Sir Charles Antony Richard Hoare introduced null references in ALGOL in 1965 as a simple solution to implementing trees using pointers, something he reflected upon almost half a century later [Hoare, 2009]:

> "That led me to suggest that the null pointer was a possible value of every reference variable [...] and it may be perhaps a billion-dollar mistake."

Null pointers have over the decades caused numerous issues but are obviously not the only pointer-related problem. A whole class of memory safety issues are caused by incorrect handling of pointers. Buffer overflow attacks are possible when code does not properly check if a pointer points to memory appropriately allocated for the purpose. Some types of arbitrary code execution are possible when pointers are used to write executable code to memory which is later executed as part of normal program execution or another attack, like a shellcode attack.

A capability is, in the context of this thesis, a pointer that grants authority for a set of operations over a specific range of memory such as an array or object. Capability machines are processors that implement support for capabilities and —more importantly— efficiently enforce the invariants provided by them. When used properly, they can mitigate a wide class of memory safety problems and hence have long been studied academically, e.g., a design for *guarded pointers* to be implemented in hardware [Carter et al., 1994]. Renewed interest has recently emerged in the form of a modern capability machine by Watson et al. [2019], Capability Hardware Enhanced RISC Instructions or **CHERI** for short, and the opportunities it presents for security features in high-level software abstractions. Chapter 2 gives an overview of capabilities on CHERI.

Much research into capability machines and security features based on them is done on theoretical machines like the linear capability machine by Skorstengaard et al. [2019b]. While this allows for rigorous mechanised proofs, it does not offer the same level of hands-on experience that a compiler and emulator do provide. A fork of the LLVM compiler suite[1] (especially the Clang C++ compiler) and CheriBSD,[2] a fork of the FreeBSD operating

[1] The CHERI LLVM project repository is available at `https://github.com/CTSRD-CHERI/llvm-project`.

[2] The project repository is available at `https://github.com/CTSRD-CHERI/cheribsd`.

system, provide a testing ground. However, the LLVM codebase is immense and targets numerous architectures and thus does not lend itself to experimentation with radically new ideas.

This thesis explores a compiler that implements a few capability-based security features and produces executables for CHERI-RISC-V processors, which are RISC-V processors extended with support for CHERI capabilities. This compiler, which we call **Glyco**, is designed, implemented, and evaluated in four stages, following a nanopass approach. A nanopass performs a small program transformation; a nanopass compiler transforms a program in a source language through numerous nanopasses to a target program, which in Glyco's case is CHERI-RISC-V assembly. In this thesis we show how this approach permits experimentation by keeping the changes required to implement new functionality mostly localised to a few nanopasses.

The first version implements basic support for semi-functional programs with very limited capability-based security features. Chapter 3 explores the nanopass approach, which is used in some educational and commercial compilers, then lays out the different parts of the first version of the Glyco compiler. This version is later used as the baseline for evaluating compiler extensions.

Capabilities enable the use of **secure calling conventions**, which conform to a few desirable security properties around procedure calls and their local state. This thesis discusses *local state encapsulation*, a security property that guarantees that a procedure's local state is not accessible from other procedures or calls, and *well-bracketed control flow*, which ensures that called procedures return correctly to their caller. Chapter 4 discusses these properties in more detail before presenting this thesis' first contribution, a variant of a secure calling convention proposed by Georges et al. [2021a] as well as its implementation in Glyco. This calling convention provides local state encapsulation and a weaker variant of well-bracketed control flow we call *unrepeatable return*, which only guarantees that return capabilities are used at most once. We evaluate the nanopass approach by measuring the extension's impact on the compiler codebase and assess the impact of this calling convention on built programs by comparing them to a compilation with a more traditional calling convention.

A second contribution of this thesis is a feature we call **sealed objects**, which are similar to objects in object-oriented programming languages but with additional security properties. A sealed object's local state is only accessible from its methods and this is enforced at the hardware level with sealed capabilities. Chapter 5 examines the design and implementation of sealed objects as well as two additional features that are at the basis, namely **lambdas** and **named types**. We then evaluate the nanopass approach once again by quantifying the feature's impact on the compiler codebase.

A third and final contribution are **sealed closures**, which are anonymous functions that securely capture the environment they're defined in — the environment of a sealed closure cannot be accessed outside of the closure's body. Sealed closures rely on sealed objects and are thus a prime application for them. Chapter 6 explores their design and evaluates the nanopass approach's impact on their implementation.

A note on versioning & syntax This thesis text presents a compiler in four iterations (or milestones), starting with a basic CHERI-RISC-V compiler with almost no security features built on capabilities (Glyco 0.1) and ending with a compiler featuring a secure calling convention (0.2), sealed objects (0.3), and sealed closures (1.0). One of the goals of this thesis is to explore how the nanopass approach performs for designing and implementing a compiler from scratch but also for extending it.

To present the reader with a consistent narrative, we have structured the text so that each chapter limits itself to the feature set and languages of the version being discussed. The first chapter discussing the compiler, Chapter 2, introduces Glyco 0.1 while the following chapters extend this compiler. However, we have chosen to use the final syntax for examples across the text. The syntax of the discussed languages does not change significantly between versions and is of lesser importance to understanding the compiler and its evolution. A single syntax also allows the reader to readily try out the examples in the most refined version of the compiler.

A full grammar of the final compiler's different languages is presented in Appendix A.

Source code The full source code as well as build & usage instructions are available at `https://tsarouhas.eu/glyco/`.

Chapter 2

Background: Ensuring Memory Safety using Capabilities

Most security bugs classified as "serious" in large projects such as the Chromium browser engine are memory safety bugs [The Chromium Authors, 2022] — bugs such as out-of-bounds array or call stack accesses that arise from the use of "memory-unsafe" programming languages like C, the use of memory safety opt-outs (such as Rust's `unsafe`) in "memory-safe" programming languages, bugs in the compiler, bugs in the standard library implementation of a memory-safe programming language, or bugs in the runtime environment such as the Java VM or an ECMAScript interpreter.

Common approaches to mitigating attacks or restricting the attack vector that such a bug can cause include reintroducing compiler-inserted array bounds checks in memory-unsafe programming languages and enforcing page-level permissions in the virtual memory system of the operating system. However, runtime checks introduce a runtime cost, and virtual memory protection doesn't protect against a compromised or vulnerable library such as Apache Log4j accessing potentially sensitive data in the process' address space such as TLS session keys stored in an in-process OpenSSL data structure.

A different approach are **capabilities**, which are unforgeable tokens that carry authority over an entity [Levy, 1984, Section 1.1]. They have been described (without naming them) since at least the 1970s, e.g., in the context of the Multics timesharing system (which later played an important role in the design of Unix) by Saltzer [1974], but renewed research interest has emerged since the development of a design for a new capability machine called CHERI in the early 2010s [Watson et al., 2020, Section A.1 and Chapter 13], which we discuss in the next section.

In this thesis, capabilities are unforgeable pointers that carry authority over a precise region of memory for a specified set of operations. While virtual memory systems provide coarse-grained memory protection, usually at the level of a page, capabilities provide *fine-grained* memory protection. A memory allocator can for instance return a capability that grants access to (and only to) the allocated region of memory. To ensure that authority cannot be created, capabilities can only be derived from a source with more authority, such as an operating system, a more privileged routine, or another capability. Finally, capability support can be provided at the hardware level, removing the runtime cost

associated with software-based bounds & permission checks.

2.1 Capability Hardware Enhanced RISC Instructions

CHERI (**Capability Hardware Enhanced RISC Instructions**; see also Watson et al. [2019]) is a design for a capability machine, extending several existing instruction set architectures (ISAs) such as RISC-V, MIPS, x86-64, and Arm with hardware capability support. One of CHERI's design goals is to provide a viable transition path for mainstream systems. An ecosystem formed around CHERI, such as capability extensions for C & C++, capability support in the LLVM compiler toolchain and QEMU emulator, a FreeBSD fork with capability support called CheriBSD, and several large libraries and systems such as PostgreSQL and WebKit being ported to CheriBSD. Additionally, CHERI supports an execution mode that allows capability-unaware code to work as if they're executed on non-capability hardware.[1]

In 64-bit CHERI-RISC-V, the CHERI extension of the 64-bit RISC-V ISA, 64-bit pointers become 128-bit capabilities. As illustrated in Fig. 2.1, such a capability consists of a 64-bit address, a 27-bit compressed value indicating the capability's bounds relative to the address, a 16-bit permissions mask (specifying permissions such as *permit load*), with the remaining 21 bits reserved for encoding other flags and the capability's object type (used for a CHERI feature called "sealing", cf. infra). An out-of-band 1-bit validity tag is set when the 128-bit datum represents a valid capability; this bit cannot be set in software.[2] The tag is cleared whenever the capability is modified by any instruction not intended to modify capabilities such as `XOR` or `ADD`.

CHERI-RISC-V defines several capability-aware instructions that *can* modify capabilities via guarded manipulation. For instance, `CAndPerm` removes permissions from a given capability by performing a bitwise conjunction with a given permissions mask whereas `CIncOffset` adds an offset to a given capability's address. These limitations are in place to ensure 4 important safety properties of capability machines and CHERI architectures in particular, briefly summarised here below.

Permissions A capability specifies zero or more permissions that the holder of the capability is allowed to exercise. For example, the *permit load* resp. *permit store* permissions allow the holder to load resp. store data in the region of memory specified by the capability. Beyond permissions also found in traditional virtual memory systems, capabilities also support CHERI-specific permissions such as *permit capability load* and *permit capability*

[1] CHERI also supports a hybrid model where compilers can implement return addresses and function pointers using capabilities while implementing other pointers using capability-unaware instructions, thereby improving the security of legacy code without breaking it. Capability-unaware instructions in CHERI-RISC-V evaluate pointers as offsets of the capability in the DDC (Default Data Capability) register, which a capability-aware operating system or runtime can configure before passing control to the capability-unaware program.

[2] CHERI does not specify where validity bits are stored but it requires one such bit for each register and for each capability-aligned memory location that supports storing capabilities. Validity bits for registers may for instance be kept in a bitmap in an internal register, whereas validity bits for memory locations may be stored in tagged memory, "not directly accessible via data loads or stores."

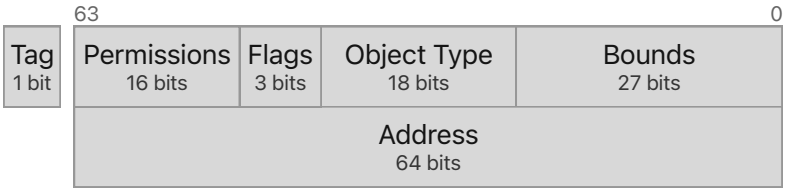


FIGURE 2.1: A simplified layout of a 256-bit CHERI capability compressed into 128 bits using the CHERI Concentrate encoding. CHERI-RISC-V defines one flag bit and reserves the other two bits for future use. The validity tag bit is stored out-of-band, usually in tagged memory not directly accessible by user programs.

store. Performing an action not allowed by the capability, such as storing data using a load-only capability, results in a machine trap.[3]

Bounds A capability specifies a contiguous region of memory where the holder of the capability is allowed exercise actions permitted by the capability's permissions. For example, a capability produced by a secure, capability-aware implementation of `malloc` pointing to a heap-allocated buffer would only permit accesses within that buffer. Performing an action outside of these bounds, i.e., a buffer overflow, results in a machine trap.

Provenance A capability can only be derived from other capabilities. Authority cannot be forged by modifying its in-memory representation; attempting to do so causes the capability's tag bit to be cleared which invalidates the capability. Tag bits cannot be set in software.

Monotonicity A capability cannot have more authority than the capability it is derived from: permissions can only be removed and bounds can only be shrunk. At CPU reset, the hardware provides root capabilities to the bootloader or firmware, which can be iteratively restricted in higher levels of abstraction, as illustrated in Fig. 2.2.

2.2 Sealed Capabilities

CHERI provides a capability protection mechanism called **sealing**, whereby the capability holder is restricted from modifying or dereferencing the capability. The inverse operation, i.e., **unsealing**, is only allowed under specific circumstances. Sealing comes in two forms: sealing using a seal capability or sealing as a sentry capability. The sealed state of a capability is encoded in its **object type** field; the object type of an unsealed capability has the special value -1.

Sealing using a seal capability A capability with the *permit seal* permission, i.e., a **seal capability**, can be used to seal another capability. The object type of the sealed capability

[3] In some specific cases, such as loading a capability using a capability that only allows loading data but not capabilities, merely results in a capability's tag being cleared, thereby deactivating any authority that it otherwise might unintentionally have conferred.

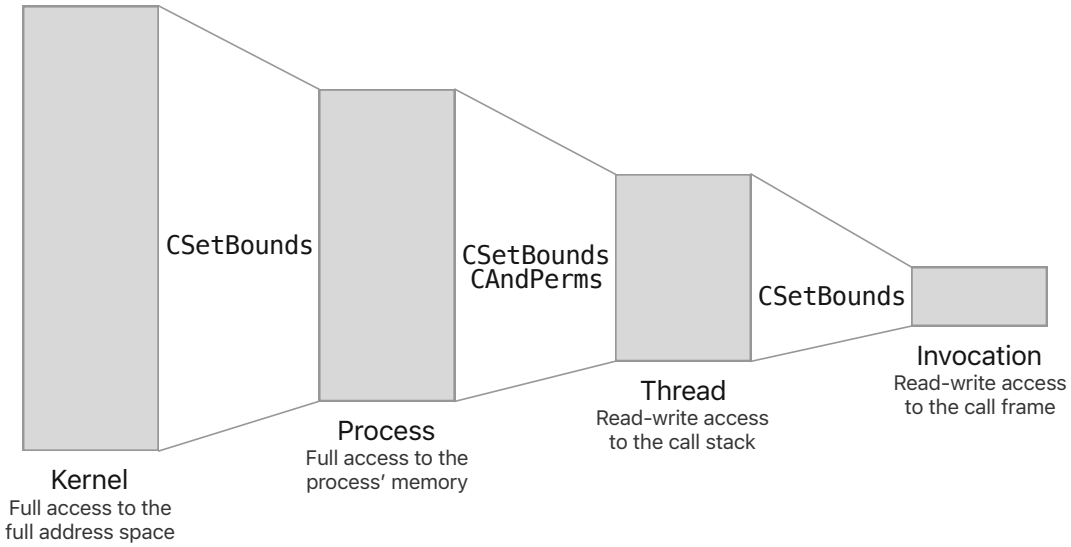


FIGURE 2.2: Provenance and monotonicity in action in protecting process memory, the call stack, and a call frame. The size of the depicted memory regions are not to scale. Similar illustrations can be made about a process' heap and heap-allocated memory.

is set to the seal capability's "address".[4] A capability with the *permit unseal* permission, i.e., an **unseal capability**, can be used to unseal a sealed capability provided that the sealed capability's object type is equal to the unseal capability's "address". A seal capability can also act as an unseal capability if it has both permissions.

As would be expected from normal pointer-like capabilities, the (un)seal capability's address must be within its bounds when (un)sealing; otherwise a machine trap ensues. By carefully controlling access to and bounding capabilities with the *permit seal* and *permit unseal* permissions, sealing can be used for higher-level features such as encapsulation.

Sealing and unsealing can be done using the `CSeal` respectively `CUnseal` instructions. CHERI-RISC-V additionally provides a `CInvoke` instruction that takes a code and data capability pair with matching object type, unseals them, and jumps to the code capability's address. This feature enables secure domain transitions, wherein the caller cannot dereference the code or data capability (which may belong to a closure, for example), but can use them to perform an invocation.

Sealing as a sentry capability An executable capability can be sealed by itself as a **sentry capability** using the `CSealEntry` instruction. CHERI-RISC-V defines a `CJALR` instruction that jumps to the address of a given target capability, unsealing the capability if it is a sentry capability. Similar to RISC-V's `JALR` instruction, the instruction also accepts a link

[4] This "address" does not necessarily need to refer to a valid location in memory. A capability whose address is only used for sealing and unsealing capabilities should therefore omit the *permit load* and *permit store* family of permissions to avoid letting holders of the seal capability access invalid memory locations. Also note that the object type field is 18 bits wide whereas the address field is 64 bits wide: sealing with a seal capability whose address does not fit in 18 bits results in a machine trap. Additionally, object types reserved by the architecture cannot be used for (un)sealing using a seal capability.

register wherein it puts the return address. Unlike `JALR` however, `CJALR` also seals the return capability as a sentry capability, ensuring that the callee can only use it to return to the caller. A sentry capability's object type has the special value −2.

Chapter 3

Glyco: A Nanopass Compiler for CHERI-RISC-V

Glyco[1] is a compiler targeting the CHERI-RISC-V architecture, producing software running on CheriBSD systems and on a Sail-based emulator of the CHERI-RISC-V ISA.

Glyco follows a nanopass compiler design, an iterative approach to designing and implementing a compiler that results in a series of transformations of a program from a high-level language to a low-level language (such as assembly) via several intermediate languages. We briefly discuss this approach in Section 3.1. The languages in Glyco are summarised in Fig. 3.1.

In Section 3.2 we discuss the high-level **Expressions** language (EX), a functional programming language with support for common features such as functions, conditionals, definitions, vectors, and records. These features are implemented in several transformations from EX to the CC language.

The **Calling Convention** language (CC) is a procedural programming language with support for procedures, variables, conditionals, vectors, and records. It is the lowest language in Glyco that supports the notion of a procedure with parameters and results. CC programs are transformed to SV programs by imposing a set of rules governing procedure calls, i.e., a calling convention. We discuss calling conventions and the CC language in Section 3.3.

Structured Values (SV) is a structured imperative programming language with support for variables, conditionals, vectors, and records, and is discussed in Section 3.4. It is so named because it is the lowest abstraction that supports structured values, i.e., vectors and records, as opposed to untyped byte buffers. SV programs are transformed to ID programs by substituting indices and fields with byte offsets.

Inferred Declarations (ID), **Abstract Locations** (AL), and **Abstract Locations Annotated** (ALA) are three structured imperative languages with support for variables and conditionals; all three are discussed in Section 3.5. ID programs do not need to declare local variables whereas AL programs do need to declare them. The transformation between the two languages performs a basic inference algorithm to discover the names and types of variables. AL programs are then lowered to ALA programs which contain

[1] From the Greek term γλυκὸ meaning "sweet", alluding to the taste of a wild cherry.

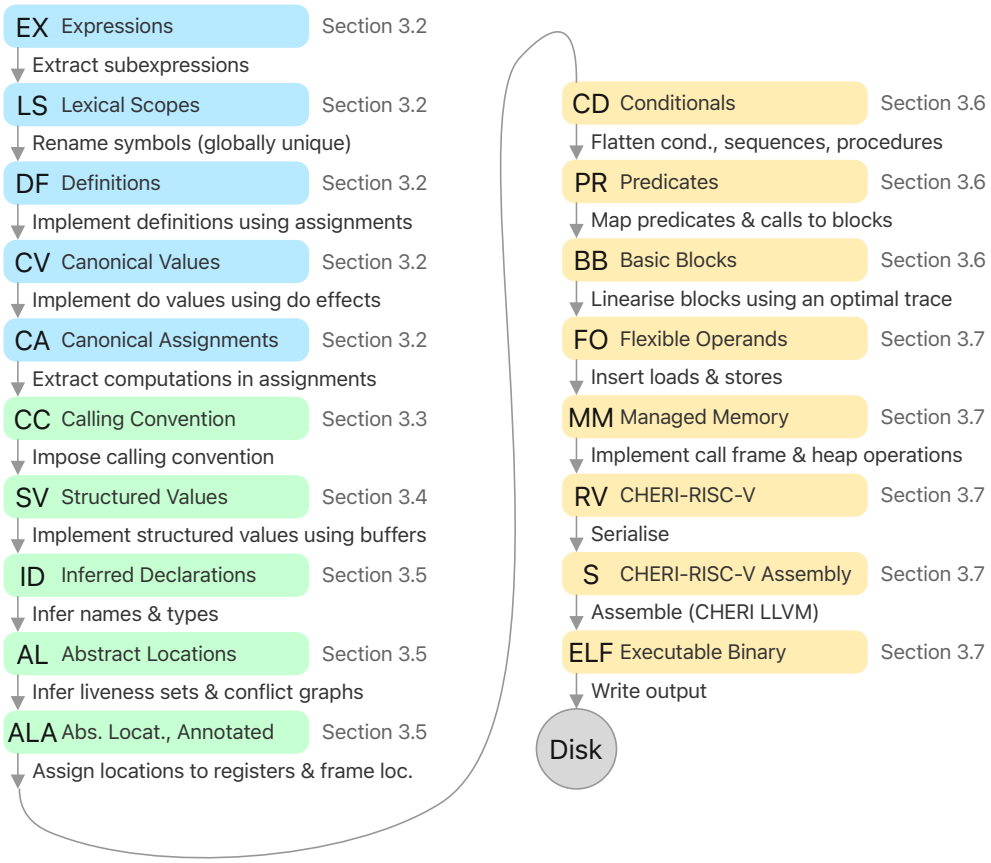


FIGURE 3.1: The Glyco 0.1 pipeline, consisting (from top to bottom) of high-level (blue), mid-level (green), and low-level (orange) languages. The informal partition of the pipeline in three segments is mostly based on how much access to physical locations (such as registers) a language provides.

additional liveness and conflict information. This information is then used to perform register allocation and assign a register or memory location to each variable, resulting in a CD program.

In Section 3.6 we go over **Conditionals** (CD), a structured imperative language with support for conditionals. CD programs are flattened over several passes to **Flexible Operands** (FO) programs, which consist of labelled and unlabelled effects in a similar way to labelled and unlabelled instructions in an assembly program. We then tackle the last few languages and transformations in Section 3.7 that finally produce an ELF executable.

This chapter discusses the feature set and languages of Glyco 0.1,[2] which allows us to focus in each chapter on one set of functionality. It also permits us to evaluate how the nanopass approach performs with extensions in future versions which we discuss in later chapters.

Example programs in this chapter follow the syntax of the final version of the compiler,

[2]The source code is available at `https://tsarouhas.eu/glyco/0.1/`.

Glyco 1.0,[3] and are compiled using the default configuration. The syntax of a program in one compiler version does not significantly change in another version but for consistency we chose to express all programs in the latest compiler's syntax, skipping languages introduced after Glyco 0.1. A full language reference of this syntax can be found in Appendix A.

3.1 Background: Nanopass Compilers

Glyco is built following a *nanopass* compiler design, described in the context of compiler education [Sarkar et al., 2004] and commercial compiler development [Keep and Dybvig, 2013]. A nanopass compiler consists of numerous small passes, so-called **nanopasses**, which translate programs written in one **intermediate language (IL)** to programs written in another. The input of a nanopass is a program in a higher-level language than the one of its output; we therefore call this translation process a **lowering**. A nanopass should be concise and perform a relatively self-contained transformation. There should be no data structures shared across nanopasses except for the program itself.

The nanopass design allows the compiler engineer to design and implement their compiler *by iterated abstraction*. A simplified description follows. The engineer first chooses a target language (usually a machine language such as x86-64 or indeed CHERI-RISC-V) and determines an abstraction over it. The engineer then defines an IL that implements that abstraction as well as a nanopass which transforms programs written in the new IL to the target language. The compiler engineer then repeats this process, this time abstracting over the IL with a new IL and nanopass. This process goes on until a level of abstraction has been reached that can either be directly used as a source language (by human users), or be easily produced by parser actions.

An important benefit of the nanopass approach is each new iteration begins and ends with a working compiler. After each iteration, users can start writing and compiling programs in the new IL and unit tests can be written that ensure that the new nanopass produces the expected transformations. For experimental architectures such as CHERI-RISC-V, this also means that designers can experiment more quickly with new ideas, something that may be harder to do on a full-fledged production compiler such as Clang.

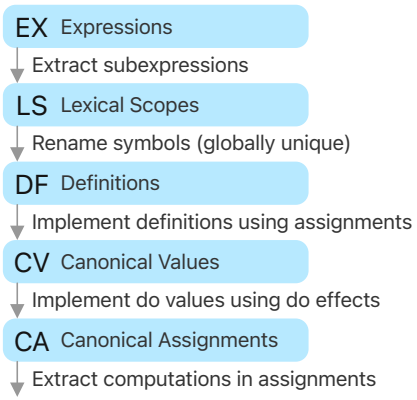
Glyco 0.1 defines 17 ILs,[4] a majority of which are adapted from the student compiler by Bowman et al. [2022] for x86 targets. Each language is usually named after the new abstraction or feature it brings compared to the lower language. We discuss the ILs in reverse chronological order (from high-level to low-level languages) as opposed to the order in which the ILs were defined (from low-level to high-level languages). This allows us to explain the compiler pipeline using example programs, showing how they are manipulated by the compiler's nanopasses.

[3] The source code is available at `https://tsarouhas.eu/glyco/1.0/`.

[4] We don't consider ELF to be an IL in this thesis.

3.2 An Expression Language

This section introduces the EX, LS, DF, CV, and CA languages. The nanopasses between these ILs consecutively transform programs from a functional to an imperative, procedural language.



The highest IL in the first version of Glyco is called **EX** (Expressions), a language with basic support for arithmetic operations, functions, conditionals (`if-then-else`), definitions (`let`-bindings), vectors (fixed-size arrays), and records (collections of key-value pairs).

The following EX program computes the 30th number in the Fibonacci sequence starting with 0 and 1:[5]

```
(
  evaluate(function(fib), 0 1 30),
  functions:
    (fib, takes: (prev, s32) (curr, s32) (iter, s32), returns: s32, in:
      if(relation(iter, le, 1),
        then:  value(curr),
        else:  evaluate(function(fib), curr binary(prev, add, curr) binary(iter, sub, 1))
      )
    )
)
```

This program defines a function `fib` with three signed 32-bit integer parameters `prev`, `curr`, and `iter` which returns a signed 32-bit integer. The function recursively adds the "previous" and "current" sums until the number of iterations drops to 0, then returns the "current" sum. The program itself is defined as the result of `fib` where `prev` is 0, `curr` is 1, and `iter` is 30.

EX is an *intermediate* language and not necessarily a *user-friendly* language; it contains little syntactical sugar. For example, the predicate $iter \leq 1$ is written as `relation(iter, le, 1)`. A significant benefit is that the structure is easy to manipulate as the code goes through the compiler's nanopasses. One can however define a more user-friendly language and write a parser that emits EX.

From EX to LS LS (Lexical Scopes) does not support subexpressions so the nanopass to LS binds all subexpressions to names (temporaries) and uses those names in place of the subexpressions. For instance, the EX value

```
evaluate(function(fib), ... binary(prev, add, curr) ...)
```

is transformed into the equivalent LS value

[5] EX and all other ILs conform to the Sisp format, where a list is written as the list's elements separated by whitespace and where attributes are separated by commas. Sisp is briefly documented in Appendix B.

```
let(
  (arg0, ...) (arg1, let(
    (ex.lhs, source(prev)) (ex.rhs, source(curr)),
    in: binary(ex.lhs, add, ex.rhs)
  )) (arg2, ...),
  in: evaluate(function(fib), arg0 arg1 arg2)
)
```

Some transformations are not strictly necessary, e.g., binding `prev` to `ex.lhs` instead of using `prev` directly. They do not however incur any additional overhead in the final executable: a nanopass further down the compiler pipeline cleans up these redundant temporaries.

From LS to DF DF (Definitions) does not support shadowing; all `let` bindings are in a single namespace. The nanopass from LS to DF implements shadowing by renaming symbols such that each definition uses a globally unique symbol.

From DF to CV CV (Canonical Values) does not support `let` bindings, i.e., definitions. The nanopass implements each definition in CV using a `do` value, i.e., a computed value, a value on which a computation can be attached, with a `set` effect in it. For example, the DF value

```
let((answer, source(42)), in: source(answer))
```

is lowered to the CV value

```
do(set(answer, to: source(42)), then: source(answer))
```

From CV to CA CA (Canonical Assignments) does not support `do` *values*, but does support `do` *effects*, which are sequences of effects. The nanopass implements `do` values by extracting their effects into `do` effects and ending with an additional `set` effect that moves the result to the intended destination. That is, a CV `set` effect

```
set(result, to:
  do(
    /* effects */,
    then: /* result */
  )
)
```

is transformed to a CA `do` effect

```
do(
  /* effects */
  set(result, to: /* result */)
)
```

From CA to CC CA programs perform assignments using `set` effects, whereas CC (Calling Convention) only supports `set` effects with a constant or location source, and provides other effects that perform more complicated tasks such as allocating records. The final purely structural transformation lowers `set` effects in CA using equivalent effects in CC. For example, the CA program

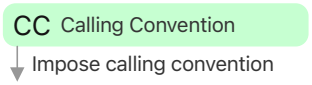
```
(
  do(
    set(a, to: source(1))
    set(b, to: source(a))
    set(c, to: binary(1, add, 2))
    set(d, to: record(
      (
        (name, cap(vector(of: u8, sealed: false)))
        (age, s32)
      ), sealed: false
    ))
    set(e, to: field(name, of: d))
    set(f, to: vector(s32, count: 100))
    set(g, to: element(of: f, at: 50))
    return(g)
  )
)
```

is lowered to the CC program

```
(
  do(
    set(a, to: 1)
    set(b, to: a)
    compute(c, 1, add, 2)
    createRecord(
      ((name, cap(vector(of: u8, sealed: false))) (age, s32)),
      capability: d,
      scoped: false
    )
    getField(name, of: d, to: e)
    createVector(s32, count: 100, capability: f, scoped: false)
    getElement(of: f, index: 50, to: g)
    return(g)
  )
)
```

3.3 A Conventional Calling Convention

This section introduces calling conventions, the CC language, and its associated nanopass.



A **calling convention** is a set of rules imposed by the operating system, instruction set architecture, and/or programming language that specify how procedures are called. A low-level calling convention often specifies

- where parameters and result values are placed (in dedicated registers, in a particular order on the call stack, or a combination of both);
- how large values are passed to the callee or caller (over multiple registers, on the call stack, or in heap memory), if supported;
- the state of the call stack and some registers (like the register keeping the frame pointer) when a procedure starts executing and when it returns to the caller;
- which registers a procedure can use freely but for which it cannot assume that their contents will be preserved across a procedure call (**caller-saved registers**); and
- which registers a procedure can only use after saving their previous contents and if they're restored after use (**callee-saved registers**).

The calling convention used in the base version of Glyco is called **Glyco Conventional Calling Convention (GCCC)** and mimics a traditional calling convention that would be used for programs written in C, with some parts specified by [Waterman and Asanović, 2019, chapter 25].

Call stack The call stack is a region of memory on which information about procedure calls is stored, such as local state and return addresses. The stack grows downward, i.e., from high to low addresses. The `csp` register contains the **stack capability**, a capability that points to the location of the last pushed datum, i.e., the top of the stack, and gives read-write authority over the stack memory. The stack capability's address decreases whenever data are pushed to the call stack and increases whenever data are popped from the stack.

Call frames A call frame (also known as a *stack frame*) is a region of memory on the call stack that belongs to a procedure call and on which the procedure can store its local state. The `cfp` register contains the **frame capability**, a capability derived from the stack capability that points to the base of the call frame. The procedure places local state and expects parameters in locations that are fixed distances removed from this base.

Before accessing its call frame, a procedure pushes a call frame of the desired size to the call stack in three steps. The procedure first pushes the (caller's) frame capability to the call stack, then updates the frame capability to point to this capability, and finally allocates room for its local state by moving the stack capability so that it points to the new call frame's last datum.

A procedure pops its call frame when it is done using it, e.g., when returning to its caller. It does so by popping all data stored in the call frame and restoring the previous frame capability.

Parameters Arguments to parameters are passed to the callee via dedicated argument registers first. When the set of argument registers is exhausted, any remaining arguments are pushed (in order) to the stack where they become part of the caller's call frame for the duration of the call.[6] Since these *frame-resident* arguments reside at higher addresses

[6] This is the only part of another procedure's call frame that a callee is allowed to access directly.

than the callee's frame capability, the callee can access these arguments using positive offsets of the frame capability.

The set of argument parameters in Glyco is customisable via a command-line option. The default set is `a0`, `a1`, `a2`, `a3`, `a4`, `a5`, `a6`, and `a7`.

All of Glyco's supported data types in Glyco fit in a register. Vectors and records are passed by reference as capabilities.

Results A procedure can return a single value via the `a0` register. Vectors and records are returned by reference as capabilities.

Available registers The set of caller-saved and callee-saved registers is customisable via command-line options. The default caller-saved registers are `s1`, `s2`, `s3`, `s4`, `s5`, `s6`, `s7`, `s8`, `s9`, `s10`, and `s11`; the default callee-saved registers are the remaining 9 registers available in CC.

Local state A procedure call's local state comprises saved register data (cf. supra), data for which no register is available, and stack-allocated buffers. These reside at lower addresses than the call's frame capability; the procedure can therefore access them using negative offsets of the frame capability.

Return address The caller provides a return address capability in the `cra` register.[7] The callee can return control to the callee by jumping to this return capability.

Figure 3.2 shows a simple two-frame call stack example with zero registers available for arguments and local variables, to force Glyco to store them in call frames.

From CC to SV CC introduces procedures with parameters and results. It is lowered to its lower language SV (Structured Values) by imposing the calling convention described above. We'll work through the CC to SV nanopass by applying it to the following CC program.

```
1  (
2    do(
3      call(procedure(sum), 19 23, result: the_sum)
4      return(the_sum)
5    ),
6    procedures:
7      (sum, takes: (first, s32) (second, s32), returns: s32, in:
8        do(
9          compute(the_result, first, add, second)
10         return(the_result)
11       )
12     )
13 )
```

[7] `cra` is a callee-saved register, so if the callee needs to call another procedure, it will need to save the contents of `cra` before overwriting the register, except in the special case of a tail-call.

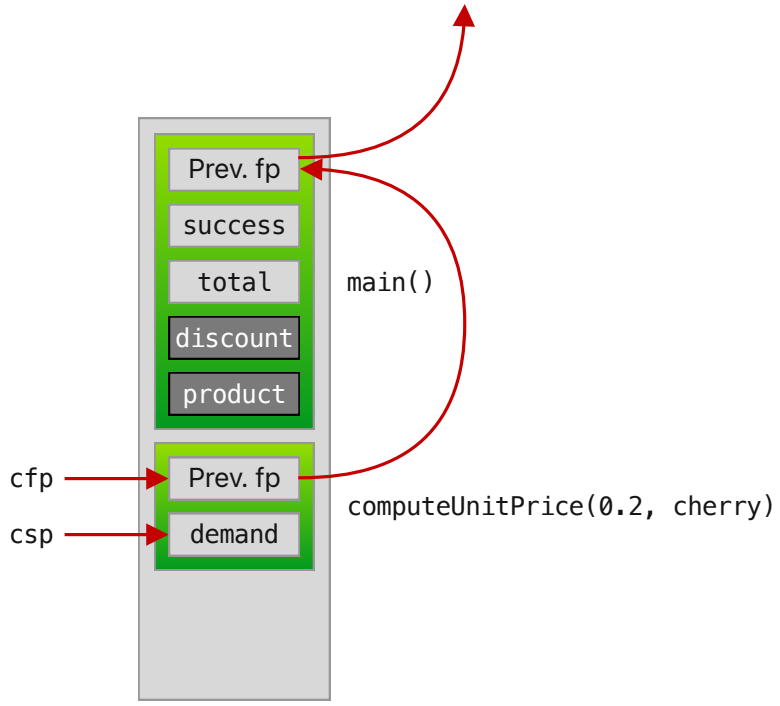


FIGURE 3.2: A call stack containing two call frames in a configuration of GCCC where no registers are available for storing parameters and local variables. The first call frame belongs to the initial procedure `main` which is called by the runtime, a dynamic linker, or the operating system and takes no parameters. Its call frame contains a *previous frame pointer* capability that points to an unspecified location (possibly `null`) as well as two local variables `discount` and `price`. `main` invokes the 2-parameter `computeUnitPrice` procedure, binding the value `0.2` to the `discount` parameter and `cherry` to the `product` parameter. The latter procedure's call frame contains a *previous frame pointer* capability that points to the location of the *previous frame pointer* capability in the previous call frame, as well as a local variable `demand`. This call frame is at the top of the stack and so the `cfp` register's capability points to the *previous frame pointer* capability in `computeUnitPrice`'s call frame. The `csp` register's capability always points to the last datum pushed on the stack, which is `demand` in this example. Note that the stack grows downward.

The main program begins by pushing a new scope, which is essentially the same as pushing a new call frame to the call stack.[8]

```
1 (
2   do(
3     pushScope
```

The program receives a return capability in `cra`, which it can use to return to the runtime (or operating system). To free up this register for other uses, it binds its value to an abstract location.[9]

[8] The term *(location) scope* abstracts from the call frame. ILs above AL do not support accessing the call frame directly but do provide a way of creating and restoring sets of abstract locations, hence the term.

[9] `call` effects redefine the register. The register allocator will notice a conflict and thus assign the saved

```
4       set(abstract(cc.retcap), to: register(ra, cap(code)))
```

The `call` effect is lowered by copying the arguments to the appropriate registers (or onto a newly stack-allocated arguments record for arguments that don't fit in registers), passing control to the callee, and finally (after control is returned) copying the result from `a0` to the result location. The register allocator (discussed in Section 3.5) needs to know what registers are used for passing arguments and are thus live across the call; the `call` effect is therefore annotated with a list of used arguments registers.

```
5       set(register(a0), to: 19)
6       set(register(a1), to: 23)
7       call(capability(to: sum), parameters: a0 a1)
8       set(abstract(the_sum), to: register(a0, s32))
```

The program finishes by restoring the return capability (which it can only do while the current scope is valid), popping the current scope, then returning to the program's caller.

```
9       set(register(a0), to: the_sum)
10      set(register(ra), to: cc.retcap)
11      popScope
12      return(to: register(ra, cap(code)))
13    ),
```

The procedure begins analogously to the program by pushing a new scope and binding the caller's return capability to an abstract location. It does the same for all callee-saved registers, since their contents must be preserved by a procedure before it can use them. The register allocator in a later nanopass cleans up (some) redundant moves.

```
14    procedures: (
15      sum,
16      in: do(
17        pushScope
18        set(abstract(cc.savedS1), to: register(s1, registerDatum))
19        /* same for s2 through s10 */
20        set(abstract(cc.savedS11), to: register(s11, registerDatum))
21        set(abstract(cc.retcap), to: register(ra, cap(code)))
```

It then binds each received argument to a location with its corresponding parameter's name, making those arguments available to the rest of the procedure which expects arguments in those locations. This operation also frees the argument registers for other uses, such as for calls within the procedure.

```
22        set(abstract(first), to: register(a0, s32))
23        set(abstract(second), to: register(a1, s32))
```

The rest of the procedure is left as-is.

```
24        compute(abstract(the_result), first, add, second)
```

return capability to a location on the call frame. For leaf procedures, no such conflict occurs and the register allocator optimises away the binding to `cc.retcap`.

The `return` effect is lowered by copying the result to the result register `a0`, restoring the contents of callee-saved registers and the return capability, popping the current scope, and returning control back to the caller.

```
25        set(register(a0), to: the_result)
26        set(register(s1), to: cc.savedS1)
27        /* same for s2 through s10 */
28        set(register(s11), to: cc.savedS11)
29        set(register(ra), to: cc.retcap)
30        popScope
31        return(to: register(ra, cap(code)))
32      )
33    )
34  )
```

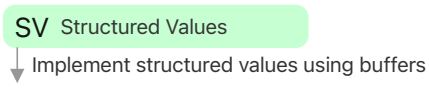
Listing 1 shows this program after a lowering to CHERI-RISC-V assembly (S), although the association between SV and S code may be unclear as there are many nanopasses between these ILs.

```
107 rv.main:        csc cfp, -16(csp)
108                 cincoffsetimm cfp, csp, -16
109                 cincoffsetimm csp, csp, -32
110                 csc cra, -16(cfp)
111                 addi a0, zero, 19
112                 addi a1, zero, 23
113                 cjal cra, sum
114 cd.ret:         clc cra, -16(cfp)
115                 cincoffsetimm csp, cfp, 16
116                 clc cfp, 0(cfp)
117                 cjalr cnull, cra
118 sum:            csc cfp, -16(csp)
119                 cincoffsetimm cfp, csp, -16
120                 cincoffsetimm csp, csp, -16
121 cd.then:        cmove ca4, cs2
122                 cmove ca5, cs3
123                 cmove ca6, cs4
124                 cmove ca7, cs5
125 cd.then$4:      cmove ca2, cs10
126                 cmove ca3, cs11
127 cd.then$5:      add a0, a0, a1
128 cd.then$6:      cmove cs2, ca4
129                 cmove cs3, ca5
130                 cmove cs4, ca6
131                 cmove cs5, ca7
132 cd.then$10:     cmove cs10, ca2
133                 cmove cs11, ca3
134 cd.then$11:     cincoffsetimm csp, cfp, 16
135                 clc cfp, 0(cfp)
136                 cjalr cnull, cra
```

LISTING 1: An assembly excerpt of the 42 program, with the `sum` procedure starting from line 118. The current version of Glyco is unable to delete some redundant moves such as the ones in lines 121–126.

3.4 Structured Values

This section introduces vectors and records (collectively called *structured values*) and the SV language.



SV (Structured Values) provides a structured abstraction over unstructured buffers in the form of vectors and records. A **vector** in SV is a homogeneous[10] fixed-size[11] array and is represented with a capability that points to the vector's first element, similarly to an array in the C programming language. A **record** is a heterogeneous[12] key-value map with fixed keys and is similar to a C `struct`. However, unlike C `structs` and like SV vectors, a record is represented by a capability that points to the record's first value. Vectors and records in SV are always passed by reference and never implicitly copied. Vector and record capabilities are automatically bounded to the region of memory they occupy; indexing a vector with an index that exceeds the vector's length causes a machine trap.[13]

From SV to ID Structured values are implemented in the lower language ID (Inferred Declarations) by translating effects dealing with them into effects dealing with unstructured data buffers. For example, the SV program (with alternating effects highlighted)

```
(
  do(
    pushScope
    createRecord(
      (
        (pi, cap(vector(of: s32, sealed: false)))
        (fib, cap(vector(of: s32, sealed: false)))
      ),
      capability: abstract(sequences), scoped: true
    )
    createVector(s32, count: 5, capability: abstract(pi), scoped: true)
    setField(pi, of: abstract(sequences), to: abstract(pi))
    createVector(s32, count: 7, capability: abstract(fib), scoped: true)
    setField(fib, of: abstract(sequences), to: abstract(fib))
    setElement(of: abstract(fib), index: 0, to: 1)
    setElement(of: abstract(fib), index: 1, to: 1)
    setElement(of: abstract(fib), index: 2, to: 2)
    setElement(of: abstract(fib), index: 3, to: 3)
    setElement(of: abstract(fib), index: 4, to: 5)
    getElement(of: abstract(pi), index: 2, to: register(a0, s32))
    popScope
    return(to: register(ra, cap(code)))
```

[10] All elements are of the same data type.

[11] The number of elements is determined at creation and can only be changed by creating a new vector of the desired size and moving all elements to it.

[12] Values are not necessarily of the same data type.

[13] Due to CHERI-RISC-V bounds compression, a buffer capability may grant a larger region of memory than requested during allocation, meaning that some "out of bounds" indices near the end index may actually be valid and not cause a trap. Glyco compensates for this by allocating the extra memory granted by the buffer capability, thereby ensuring that no other buffer can occupy this extra space accessible by the first capability.

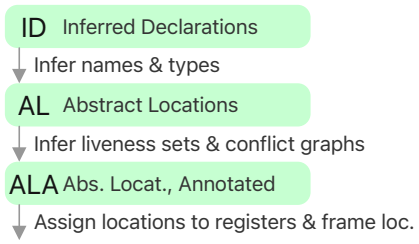
```
  )
)
```

is lowered to the following ID program (with the above highlighted effects' lowerings highlighted):

```
(
  do(
    pushScope
    createBuffer(bytes: 32, capability: abstract(sequences), scoped: true)
    createBuffer(bytes: 20, capability: abstract(pi), scoped: true)
    setElement(cap, of: abstract(sequences), offset: 0, to: pi)
    createBuffer(bytes: 28, capability: abstract(fib), scoped: true)
    setElement(cap, of: abstract(sequences), offset: 16, to: fib)
    compute(abstract(sv.offset), 0, sll, 2)
    setElement(s32, of: abstract(fib), offset: sv.offset, to: 1)
    compute(abstract(sv.offset$1), 1, sll, 2)
    setElement(s32, of: abstract(fib), offset: sv.offset$1, to: 1)
    compute(abstract(sv.offset$2), 2, sll, 2)
    setElement(s32, of: abstract(fib), offset: sv.offset$2, to: 2)
    compute(abstract(sv.offset$3), 3, sll, 2)
    setElement(s32, of: abstract(fib), offset: sv.offset$3, to: 3)
    compute(abstract(sv.offset$4), 4, sll, 2)
    setElement(s32, of: abstract(fib), offset: sv.offset$4, to: 5)
    compute(abstract(sv.offset$5), 2, sll, 2)
    getElement(s32, of: abstract(pi), offset: sv.offset$5, to: register(a0))
    popScope
    return(to: register(ra, cap))
  )
)
```

3.5 Abstract Locations

This section sets out the difference between abstract and physical locations, register allocation, and the ID, AL, and ALA languages.



All programs until now have been able to bind values to names in one form or another, be it by using new named locations in effects or binding values to names with `let`. These names refer to physical locations where data can be stored, but without the programmer explicitly specifying the physical location itself, and are therefore referred to as **abstract locations**. The compiler is responsible for assigning each abstract location a **physical location** like a register or location in the call frame.

Since registers are orders of magnitude faster to access than locations in RAM, the compiler attempts to assign as many abstract locations as it can to registers in a process called **register allocation**. Since the number of registers is limited, the compiler may need to **spill** abstract locations to locations on the call frame. Glyco employs several heuristics to minimise the runtime cost by spilling abstract locations that it thinks will be accessed the least.

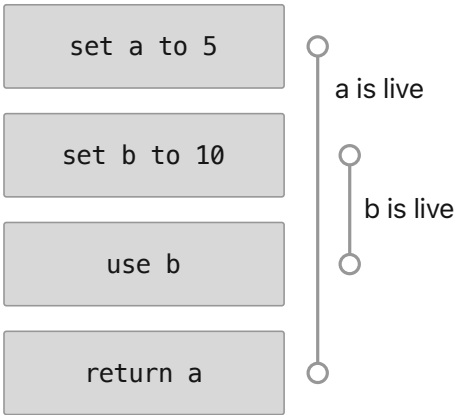


FIGURE 3.3: A liveness example where *a* and *b* have overlapping liveness intervals and thus cannot be assigned to the same register. The conflict graph contains an edge between *a* and *b*.

Glyco's register allocator is spread over three nanopasses which perform the following steps, for the main program and each procedure separately:

1. Discover the abstract locations that are used and the data types that they take.
2. For each abstract location, determine where in the program or procedure it is defined or redefined with a value and where that value is used. The execution path between a definition and that definition's last use is a **liveness interval**, a period of time where the definition's value is considered *live*.
3. Find out which liveness intervals overlap with other liveness intervals. The locations involved in such overlapping intervals are considered to be **in conflict** with each other.
4. Assign registers to abstract locations so that no two abstract locations are assigned the same register if they are in conflict with each other. If no register is available for an abstract location, allocate space on the call frame and assign it that location.

   Prioritise abstract locations with few conflicts when assigning registers as they're likely to maximise the number of abstract locations that can be assigned to a register.

A liveness example is shown in Fig. 3.3.

From ID to AL An ID program is lowered to an AL (Abstract Locations) program by inferring a declaration for each abstract location used in the program and each of the program's procedures. Each declaration associates an abstract location with a data type. For example, the ID program

```
(
  do(
    pushScope
    compute(abstract(three), 1, add, 2)
    createBuffer(bytes: 20, capability: abstract(buffer), scoped: true)
    getElement(s32, of: abstract(buffer), offset: 0, to: abstract(number))
    set(register(a0), to: abstract(number))
    popScope
```

```
    return(to: register(ra, cap))
  )
)
```

is lowered to the following AL program (declarations highlighted):

```
(
  locals: abstract(buffer, cap) abstract(number, s32) abstract(three, s32),
  in: do(
    pushScope
    compute(abstract(three), 1, add, 2)
    createBuffer(bytes: 20, capability: abstract(buffer), scoped: true)
    getElement(s32, of: abstract(buffer), offset: 0, to: abstract(number))
    set(register(a0), to: number)
    popScope
    return(to: register(ra, cap))
  )
)
```

From AL to ALA Similar to how the previous nanopass infers local declarations, the AL to ALA (Abstract Locations, Annotated) lowering infers static[14] liveness and conflict information about the program, and associates it with every effect and predicate in an ALA program. It consists of two data structures: a liveness set and a conflict graph.

An effect's **liveness set at entry** is a set of locations that are **possibly live** right before the effect is executed. Every location that is not in the set is considered **definitely dead**.[15] An effect's **conflict graph at entry** is a undirected graph of locations with an edge connecting every pair of locations that are in conflict right before the effect is executed. An effect's **liveness set at exit** resp. **conflict graph at exit** is the next effect's liveness set at entry resp. conflict graph at entry.

The analysis algorithm derives an effect's liveness set and conflict graph *at entry* from the effect's liveness set and conflict graph *at exit* using the following 2 rules.

- If an effect **defines** a location's value, the location is marked as (definitely) dead at entry and thus removed from the liveness set at entry. We know for certain that the location is not live right before the effect executes since its value is overwritten.

  Additionally, the newly defined location is in conflict with all locations that are in the liveness set at exit. The effect (possibly) starts a liveness interval for the newly defined location in the middle of liveness intervals of each location in the liveness set. These liveness intervals overlap; the conflict graph at entry is therefore updated

[14] As observed from the code itself, without running or simulating the program.

[15] We choose to explicitly qualify liveness with "possibly" or "definitely" because by Rice's theorem it is impossible to ascertain whether a value is actually live in all execution paths. Liveness is a property of a program: a definition's value may not ever be live if the effects using the value are never executed. Compilers (including Glyco) take a conservative approach and *assume* that a value is live if they cannot rule out that it is dead. This heuristic may cause suboptimal register allocation due to excessive liveness intervals, but guarantees correctness by not accidentally assigning two live values to the same register. In contrast, when an effect overwrites a location's value (without using the previous value), we can be certain about the previous value being dead at the point of entering the effect.

with edges between the newly defined location and every location in the liveness set at exit.

- If an effect **uses** a location's value, the location is marked as (possibly) live at entry and is thus added to the liveness set at entry. An effect that defines and uses the same location's value, i.e., an assignment to itself, is considered to keep the location (possibly) live.

Since the liveness set and conflict graph at entry depend on the liveness set and conflict graph at exit, the algorithm traverses the program backwards, i.e., against the direction of execution and towards the beginning of the program or procedure. The algorithm starts with an empty initial liveness set and conflict graph, i.e., a `return` effect's liveness set and conflict graph at exit are empty.

Location coalescing The number of locations in a program can often be reduced by merging locations that are assigned the same value and are not in conflict with each other. For example, `three` and `number` in

```
(
  locals: abstract(number, s32) abstract(three, s32),
  in: do(
    pushScope
    compute(abstract(three), 1, add, 2)
    set(abstract(number), to: abstract(three))
    /* use number */
    popScope
    return(to: register(ra, cap))
  )
)
```

can be coalesced to get

```
(
  locals: abstract(number, s32),
  in: do(
    pushScope
    compute(abstract(number), 1, add, 2)
    /* use number */
    popScope
    return(to: register(ra, cap))
  )
)
```

thereby saving one `set` effect (highlighted in the pre-coalescing example) and making the liveness set and conflict graph smaller.

The location coalescing optimisation algorithm works as follows:

1. Select a candidate pair that is involved in a `set` effect. The algorithm is done if there are no (non-rejected) candidate pairs.

2. If the candidate locations are in conflict with each other, reject the pair and try another.

3. Tentatively form the union of the location vertices in the conflict graph. If the degree of the merged vertex is equal to or larger than the number of available registers, reject the pair and try another. This conservative heuristic by Briggs et al. [1994] ensures that the merged location will not require spilling (to the call frame) after merging if the individual locations themselves do not require spilling.

4. Commit the union of the location vertices in the conflict graph, arbitrarily choose which location to retain, and remove the other location from the program by substituting it everywhere with the retained location. If the pair consists of a physical and an abstract location,[16] retain the physical location.

5. Repeat from Step 1.

From ALA to CD The nanopass from ALA to CD (Conditionals) uses the conflict graph at entry of the program and of each procedure to determine an assignment of abstract locations to physical locations, and substitutes each abstract locations with its assigned physical location.

Register assignment is a graph colouring problem on the conflict graph where the colours are the available registers. No two connected locations in the conflict graph can be coloured with the same colour, i.e., no two locations that are in conflict can be assigned to the same register. The register assignment algorithm works as follows:

1. Select an abstract location L with the lowest degree and without an assigned physical location. The algorithm is done if there is no such location.

2. Select an available register. If any abstract location already assigned to this register is in conflict with L, reject the register and try another. If no (non-rejected) register is available, spill the abstract location by assigning it a free location on the call frame.

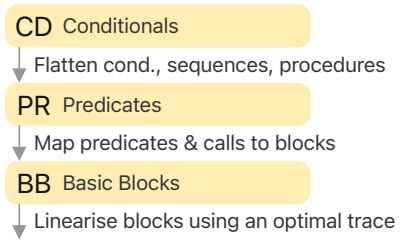
3. Repeat from Step 1.

Unlike most algorithms in the compiler's nanopasses, this algorithm runs in $O(nm)$ time for n locations and m assignable registers,[17] which underlines the importance of minimising the number of abstract locations via optimisations such as location coalescing.

3.6 Structured Jumps & Branches

This section deals with the final major abstractions over straight-line programs: conditionals (`if-then-else` effects), sequences (`do` effects), and procedures. The nanopasses from the CD to BB languages linearise programs.

[16] Usually a result of the CC to SV nanopass which puts arguments and results in dedicated registers.
[17] Assuming that conflict graph queries are done in amortised constant time, as in Glyco.



In CD, a **conditional** consists of a predicate, an affirmative branch, and a negative branch. Whenever the predicate evaluates to true, the affirmative branch is executed, otherwise the negative branch is executed. A **sequence** sequentially executes its contained effects. A **procedure** can be invoked with a `call` effect which updates the `cra` register with the address of the next effect, then jumps to the procedure's code. These three features make CD a block-structured programming language. The upcoming nanopasses get programs to a form that can be readily translated into assembly.

From CD to PR The CD to PR (Predicates) nanopass flattens conditionals, sequences, and procedures into basic blocks. A **basic block** is a labelled sequence of effects that is executed from beginning to end, before jumping or branching to itself or to other blocks. No effect in the block but the first is labelled. A basic block's effects are followed by a **continuation**, which is a jump, branch, or call. No effect except the continuation is a jump, call, or branch. For example, the CD program (code belonging to every even basic block highlighted)

```
(
  do(
    set(cap, register(s1), to: register(ra))
    set(s32, register(a0), to: 42)
    call(capability(to: abs))
    return(to: register(s1))
  ),
  procedures: (abs, in:
    if(
      relation(register(a0), le, 0),
      then: return(to: register(ra)),
      else: do(
        compute(register(a0), register(a0), mul, -1)
        return(to: register(ra))
      )
    )
  )
)
```

is lowered to the PR program (every even basic blocks highlighted)

```
(
 (
  name: rv.main,
  do: set(cap, register(s1), to: register(ra)) set(s32, register(a0), to: 42),
  then: call(capability(to: abs), returnPoint: cd.ret)
 )
 (name: cd.ret, do: , then: return(to: register(s1)))
 (
  name: abs,
  do: ,
  then: branch(if: relation(register(a0), le, 0), then: cd.then, else: cd.else)
```

```
 )
 (name: cd.then, do: , then: return(to: register(ra)))
 (
  name: cd.else,
  do: compute(register(a0), register(a0), mul, -1),
  then: return(to: register(ra))
 )
)
```

This program can be presented as a control-flow graph like in Fig. 3.4 where the edges are transitions specified by continuations and vertices are either basic blocks or (implied) callers. The graph contains one connected subgraph for the main program and for each procedure. Control jumps from one subgraph to another via `call` and `return` continuations. Each `branch` continuation specifies two outgoing edges.

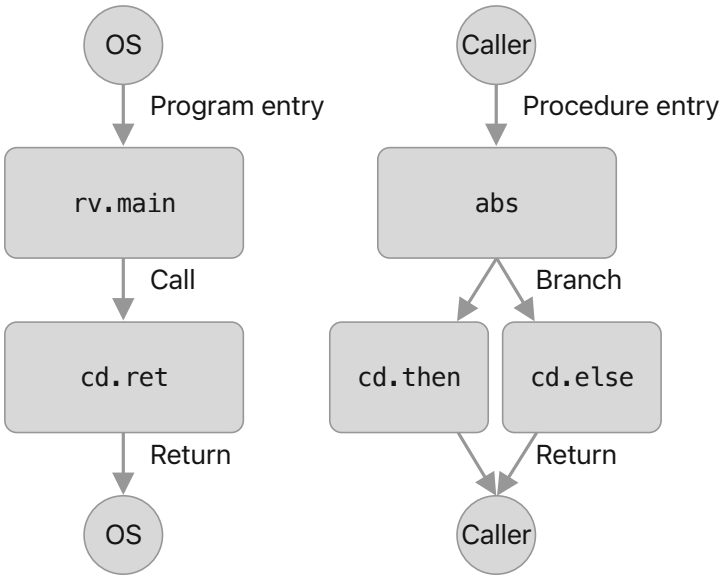


FIGURE 3.4: A control-flow graph of the PR program. The connected graph on the left represents the main program whereas the one on the right represents the `abs` procedure.

From PR to BB The next IL, BB (Basic Blocks), does not support predicates so the PR to BB nanopass maps predicates to appropriate continuations. For example, the above program is lowered to the BB program

```
(
  (
    name: rv.main,
    do: set(cap, register(s1), to: register(ra)) set(s32, register(a0), to: 42),
    then: call(capability(to: abs), returnPoint: cd.ret)
  )
  (name: cd.ret, do: , then: return(to: register(s1)))
  (name: abs, do: , then: branch(register(a0), le, 0, then: cd.then, else: cd.else))
  (name: cd.then, do: , then: return(to: register(ra)))
  (
    name: cd.else,
    do: compute(register(a0), register(a0), mul, -1),
    then: return(to: register(ra))
```

```
    )
)
```

From BB to FO The final structural transformation linearises basic blocks in BB programs to get an FO (Flexible Operands) program with a single list of unlabelled and labelled effects.

The nanopass attempts to get an optimal placement of basic blocks, i.e., an optimal trace, by juxtaposing successive blocks where possible. Unconditional jumps between juxtaposed blocks can be elided thereby avoiding the cost associated with jumps such as redundant instructions and pipeline stalls.

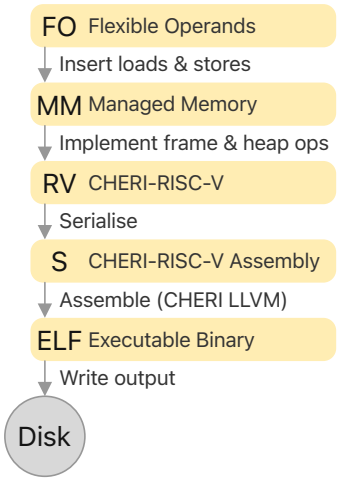
The above BB program becomes the FO program in Listing 2.

```
(
  labelled(rv.main, set(cap, register(s1), to: register(ra)))
  set(s32, register(a0), to: 42)
  call(capability(to: abs))
  labelled(cd.ret, compute(register(zero), register(zero), add, register(zero)))
  return(to: register(s1))
  labelled(abs, compute(register(zero), register(zero), add, register(zero)))
  branch(to: cd.then, register(a0), le, 0)
  labelled(cd.else, compute(register(a0), register(a0), mul, -1))
  return(to: register(ra))
  labelled(cd.then, compute(register(zero), register(zero), add, register(zero)))
  return(to: register(ra))
)
```

LISTING 2: An FO program.

3.7 Basic Abstractions over Assembly

This section briefly discusses the last ILs, which provide some useful basic abstractions over the CHERI-RISC-V instruction set.



From FO to MM FO effects accept input from several kinds of sources such as constants, registers, and (call) frame locations; and can produce results in registers or frame locations. Some source–destination combinations are not available as CHERI-RISC-V instructions. There is for instance no addition instruction that can load operands from memory or store a result in memory. There's also no subtraction instruction that takes an immediate minuend (first term).

To abstract this complexity away in FO and higher ILs, the FO to MM (Managed Memory) nanopass inserts the necessary loads and stores and reorders operands to realise an effect using CHERI-RISC-V instructions in a lower nanopass.

From MM to RV MM introduces the call stack, the heap, and operations on them.

The MM to RV (CHERI-RISC-V) nanopass translates call frame operations such as `pushFrame`, `popFrame`, `load`, and `store` into manipulations of the stack capability register `csp` and frame capability register `cfp`.

Buffer allocation is implemented using a dedicated heap capability register `ctp`. The heap capability points to the next free location on the heap and allocation increases the heap capability address — deallocation is not supported.

From RV to S An RV program consists of CHERI-RISC-V instructions. The RV to S (CHERI-RISC-V Assembly) nanopass serialises these instructions. For example, the `copyWord(destination: a0, source: a1)` instruction is serialised as `mv a0, a1`.

Listing 3 shows an S program derived from the FO program in Listing 2.

```
107 rv.main:        cmove cs1, cra
108                 addi a0, zero, 42
109                 cjal cra, abs
110 cd.ret:         nop
111                 cjalr cnull, cs1
112 abs:            nop
113                 addi t5, zero, 0
114                 ble a0, t5, cd.then
115 cd.else:        addi t4, zero, -1
116                 mul a0, a0, t4
117                 cjalr cnull, cra
118 cd.then:        nop
119                 cjalr cnull, cra
```

LISTING 3: The S program from the FO program in Listing 2, excluding runtime code.

From S to ELF The final step of the compiler pipeline is assembling and linking. These two steps are delegated to the assembler from the CHERI LLVM compiler toolchain.[18] The result is an ELF file that, depending on the compilation target, can be executed in either the Sail-based CHERI-RISC-V emulator or in CheriBSD.

[18] This last step is unrelated to the research problem, hence this dependency on LLVM.

Chapter 4

Glyco Heap-based Secure Calling Convention

The previous chapter discusses the first milestone of the Glyco compiler, namely a nanopass compiler producing fairly traditional executables for CHERI-RISC-V targets, with almost no special considerations for capability machines, except for minor enhancements such as bounded heap-allocated buffers. This chapter explores the first significant upgrade of the Glyco compiler, namely one where procedure calls are implemented using a secure calling convention named **Glyco Heap-based Secure Calling Convention (GHSCC)** and adapted from the heap-based calling convention proposed by Georges et al. [2021a].

We first describe in Section 4.1 the desired security properties of a "secure" calling convention. We then detail in Section 4.2 the low-level implementation of two runtime routines that form the basis of GHSCC. After that, we explore in Section 4.3 the implementation of the calling convention itself. We then evaluate in Section 4.4 GHSCC against GCCC in terms of changes to the compiler and its ILs, generated codesize, and runtime performance. We finish in Section 4.5 by listing a few alternative secure calling conventions described in literature or implemented in existing software.

This chapter discusses the feature set and languages of Glyco 0.2,[1] however just like in the previous chapter, example programs in this chapter follow the syntax of the final version of the compiler, Glyco 1.0. A full language reference of this final compiler can be found in Appendix A. Examples are compiled using the `--cc heap` option.

4.1 Background: Secure Calling Conventions

A traditional calling convention such as GCCC works with a single stack of call frames per thread, with each call frame holding a procedure call's ephemeral state, such as values assigned to local variables. When a procedure is called, a new call frame is pushed on the stack, the caller puts a return address in a designated register or location on the call frame, and passes control to the callee. When returning, the most recent call frame on the stack

[1] The source code is available at `https://tsarouhas.eu/glyco/0.2/`.

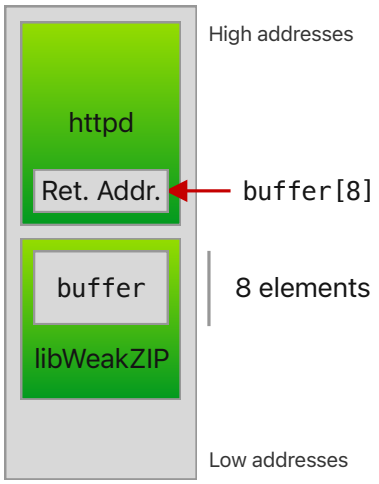


FIGURE 4.1: A vulnerability in a (trusted) libWeakZIP procedure may allow an attacker to overflow the stack-allocated buffer. In this example, the return address is stored at the end of the caller's call frame and is thus overwritten by the attack. Note that the stack grows downward.

is popped and the callee jumps to the caller-provided return address. The exact details and distribution of responsibilities are mandated by the calling convention in use.

A traditional calling convention however provides no security in the face of adversarial code that operates in the same address space. For one, adversarial code has access to the full call stack and thus to other call frames than its own. The adversarial code may be from an external dependency that has not been rigorously vetted, but it may also be first-party code containing an overflow vulnerability on a statically-allocated buffer that is exploited with specially crafted input, as illustrated in Fig. 4.1. **Local state encapsulation** is a security property that makes it **impossible for a procedure to access the local state,** i.e., call frame, **of another procedure,** [Skorstengaard et al., 2019b, Section 1] or for a procedure in one compartment (such as a library) to access the local state of a procedure in another compartment.

A different but related problem is that adversarial code may ignore the return capability provided by the caller and instead return to a different location, within or outside the caller. They may, for instance, skip an authentication check by jumping straight into the execution path of a successful authentication, or they might return to the caller's caller as illustrated in Fig. 4.2. They may also store the provided return capability and jump to it more than once. In contrast when **well-bracketed control flow** applies, **a procedure can only call procedures, return to its caller, or diverge** (e.g., get stuck in an infinite loop) [Skorstengaard et al., 2019b, Section 1].

A calling convention proposed by Georges et al. [2021a, section 7.3] provides local state encapsulation by ensuring that call frames always occupy freshly allocated memory to which certainly no dangling capabilities point. However it does not provide well-bracketed control flow. GHSCC adapts a variant of this calling convention that provides local state encapsulation and a weaker variant of well-bracketed control flow.

4.2 Secure Heap Allocation & A Runtime

As the name suggests, Glyco Heap-based Secure Calling Convention uses the heap instead of a call stack. A secure implementation of GHSCC depends on a secure implementation

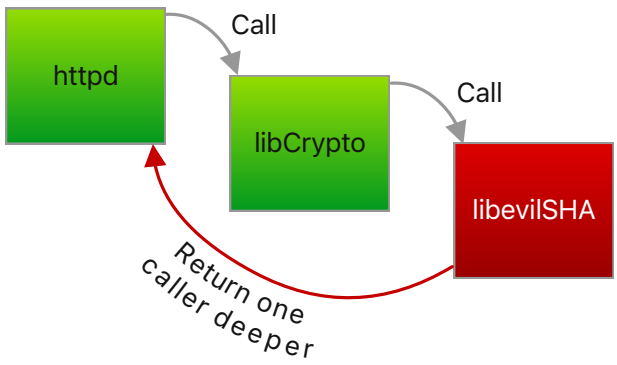


FIGURE 4.2: An example demonstrating one kind of attack when well-bracketed control flow is not guaranteed. libevilSHA returns one caller deeper, skipping libCrypto and leaving it waiting to be returned to.

of heap allocation as described by Georges et al. [2021a, section 7.1]: a region of memory belonging to a heap-allocated buffer must only be accessible to user programs via the capability provided by the allocator when that buffer is allocated.

Glyco implements a simple *bump-pointer* allocator where an allocation is performed by returning an appropriately bounded capability to the first free region in heap memory and offsetting an internal **heap capability** to point to the next free location after the allocated buffer. Deallocation is not supported; the allocator therefore never returns a capability to a region of memory used by a previously allocated buffer.[2]

To implement heap allocation and future functionality securely, Glyco is extended with support for runtime routines, which are procedures provided by the language runtime, with capabilities not afforded to normal procedures, and possibly using a different (often more basic) calling convention than ordinary procedures. Each runtime routine is stored in memory not directly accessible to user programs except via a sentry capability to the routine's entry point, as can be seen in Fig. 4.3.

As Glyco is developed almost entirely in isolation, it cannot rely on an operating system or dynamic linker to provide these routines and therefore provides them itself. It is expected that a production compiler would be able to rely on an OS-provided standard library instead, thereby removing the need to trust the compiler and its runtime.

The first runtime routine is a **heap allocation routine**, which accepts a byte size in `t0` and a return capability in `ct2` and returns a unique capability to an allocated buffer in `ct1`. The heap capability, providing access to the full heap and pointing to the next free location in it, is stored within the allocator's memory. This ensures that, at any point in time, only the allocator has access to unallocated memory and that memory associated with an allocated buffer is only ever accessible to the allocator and whoever holds the capability to that buffer returned by the allocator. Listing 4 shows the assembly code of the heap allocation routine.

[2] As described by Watson et al. [2020, section 2.3.16], capability revocation in a garbage-collected system can be implemented in several ways such as scanning accessible memory to revoke dangling capabilities, or having the memory management unit prevent accesses to deallocated buffers. Revocation and garbage collection are outside this thesis' scope.

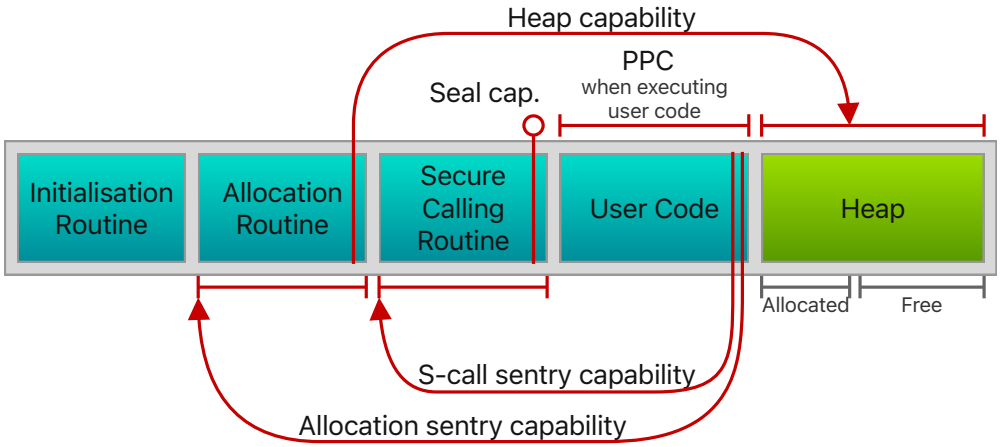


FIGURE 4.3: The process memory layout of a program compiled with GHSCC. The allocation routine contains a heap capability that grants access to the heap and points to the next allocatable location in it. The secure calling routine embeds a seal capability which it uses to seal the `cra–cfp` pair for callees. Both routines are accessible to the user program only via a sentry capability.

```
89  mm.alloc:       addi t2, zero, 15
90                  add t0, t0, t2
91                  xori t2, t2, -1
92                  and t0, t0, t2
93                  cllc ct2, mm.heap_cap
94                  clc ct2, 0(ct2)
95                  csetbounds ct0, ct2, t0
96                  cgetlen t3, ct0
97                  cincoffset ct2, ct2, t3
98                  cllc ct3, mm.heap_cap
99                  csc ct2, 0(ct3)
100                 .4byte 4276224091 # cclear 0, 128
101                 .4byte 4276881499 # cclear 3, 16
102                 cjalr cnull, ct1
103                 .balign 16
104 mm.heap_cap:    .octa 0
105 mm.alloc_end:   .balign 4
```

LISTING 4: Assembly code of the heap allocation routine as inserted in each program. The CHERI LLVM assembler did not support the `cclear` instruction at the time of development so it is encoded directly by Glyco instead. The heap capability is stored at `mm.heap_cap` and is initialised by the initialisation routine (which is part of the runtime).

```
104 mm.scall:       cllc ct0, mm.seal_cap
105                 clc ct1, 0(ct0)
106                 cseal cra, cra, ct1
107                 cseal cfp, cfp, ct1
108                 cincoffsetimm ct1, ct1, 1
109                 csc ct1, 0(ct0)
110                 .4byte 4276191323 # cclear 0, 96
111                 cjalr cnull, ct6
112                 .balign 16
113 mm.seal_cap:    .octa 0
114 mm.scall_end:   .balign 4
```

LISTING 5: Assembly code of the s-call routine as inserted in each program compiled by Glyco 0.2. The CHERI LLVM assembler did not support the `cclear` instruction at the time of development so it is encoded directly by Glyco instead. The next free seal capability is stored at `mm.seal_cap` and is initialised by the initialisation routine (which is part of the runtime).

4.3 Call Frame Encapsulation & Secure Calling

A procedure allocates a buffer for its call frame in its prologue by calling the heap allocation routine with an appropriate byte size, and uses the call frame just as it would with GCCC. The heap allocation routine ensures that only it and the procedure have access to the buffer,[3] thereby ensuring that local state encapsulation holds at that point in time.

A procedure call in GHSCC is realised using a **secure calling (s-call) routine** and is therefore referred to as an **s-call**. This runtime routine expects a target capability in `ct6`, a return capability in `cra`, and a frame capability in `cfp`. When invoked, the routine generates a unique seal capability, seals the frame and return capabilities, and jumps to the target capability's address. Similar to how the heap capability in the allocation routine grants the latter access to the heap and points to the next free location, the s-call routine contains a seal capability that grants it sealing power for all object types and whose address is the next available object type. Listing 5 shows the assembly code of the s-call routine as of Glyco 0.2.[4]

An s-call looks as follows:

1. Just as with GCCC, the caller puts the arguments for the callee in argument registers. However, if there are more parameters than there are argument registers, an arguments record is allocated on the heap (using the runtime routine described above), a capability to it is put in the last argument register, and any arguments that do not fit in registers are stored in the arguments record. The callee does not need to access the caller's call frame to gain access to arguments that did not fit in registers, nor does the caller need access to the callee's call frame to store supplementary arguments.

[3] The runtime is implicitly trusted in our threat model. In a realistic compiler and runtime environment with a heap allocation routine, the program would need to trust at least the dynamic linker and any linked runtime code.

[4] The s-call routine is generalised as a seal creation routine in the next chapter so we show the s-call routine as it was implemented originally instead of its generalisation in Glyco 1.0.

2. The caller initialises a value `returned` to 0 and stores it in its call frame. This value is later used to protect against multiple uses of the return capability.

3. The caller clears all registers except those used for the frame capability and for arguments. This ensures that the callee does not accidentally receive any authority beyond what is explicitly passed to it.

4. The caller invokes the s-call routine, passing to it a target capability to the caller and a return capability to itself. The routine also receives the caller's frame capability via `cfp`.

5. The routine seals the received frame capability and return capability with its seal capability, then increases the seal capability's address to ensure that the next s-call is sealed with a unique object type.

6. The routine clears any registers that it has used itself to ensure it does not accidentally leak new authority, then jumps to the target capability's address (the callee).

7. The callee allocates a call frame using the heap allocation routine. It then stores the caller's (sealed) frame capability in its new call frame, unless the callee is a leaf procedure, i.e., unless it calls other procedures.[5]

8. The callee executes its effect.

9. When the callee is done, it clears all registers except those used for the result and the return capability. This ensures that the caller does not accidentally receive any authority beyond what is explicitly returned to it.

10. The callee returns control to the caller with `cinvoke cra, cfp` where `cra` and `cfp` are the return resp. frame capabilities provided to the callee by the s-call routine. The machine unseals the two capabilities after checking that they have the same object type and that they comply with the instruction's preconditions,[6] then puts the second operand (`cfp`) in `ct6`, and finally jumps to the first operand's address (`cra`).

11. The caller restores its frame capability by moving the capability in `ct6` back to `cfp`.

12. The caller checks whether `returned` from Step 2 is equal to 0. If it is, it is set to 1; otherwise, the program crashes by attempting to load a value using the null capability.

13. The caller resumes execution.

[5] This behaviour emerges from the fact that the register allocator spills the locations containing the caller's sealed frame and return capabilities when a procedure contains a call effect.

[6] Among these preconditions are: the first operand must be valid and executable and the second operand must be valid and nonexecutable.

4.3.1 Lowering CC to SV with the GHSCC Configuration

To demonstrate the changes in the CC to SV nanopass, let us consider once again the CC program from the previous chapter:

```
(
  do(
    call(procedure(sum), 19 23, result: the_sum)
    return(the_sum)
  ),
  procedures:
    (sum, takes: (first, s32) (second, s32), returns: s32, in:
      do(
        compute(the_result, first, add, second)
        return(the_result)
      )
    )
)
```

As before, the main program begins (in SV) by pushing a new scope and binding the caller's (operating system's or runtime's) return capability to a temporary name.

```
(
  do(
    pushScope
    set(abstract(cc.retcap), to: register(ra, cap(code)))
```

The `call` effect is lowered by first passing arguments via registers (as before) and the rest via an argument records (which this example doesn't require). An indicator `returned` value is set to 0 which is used for enforcing the unrepeatable return property. Finally, all registers except used argument registers are cleared and control is passed to the callee.

```
    set(register(a0), to: 19)
    set(register(a1), to: 23)
    set(abstract(cc.returned), to: 0)
    clearAll(except: a0 a1)
    call(capability(to: sum), parameters: a0 a1)
```

Upon returning, the program checks whether the indicator value is still 0. If it is, it sets the value to 1 and continues. If it is not, it raises a machine trap by dereferencing a capability to a 0-sized vector.

```
    if(
      relation(cc.returned, ne, 0),
      then: do(
        createVector(s32, count: 0, capability: abstract(cc.empty), scoped: true)
        getElement(of: abstract(cc.empty), index: 0, to: register(zero))
      ),
      else: set(abstract(cc.returned), to: 1)
    )
```

The rest of the program and the `sum` procedure are lowered as before, except that all registers except the ones used for the result value and for the return capability are cleared before returning.

```
18      set(abstract(the_sum), to: register(a0, s32))
19      set(register(a0), to: the_sum)
20      set(register(ra), to: cc.retcap)
21      clearAll(except: a0 ra)
22      popScope
23      return(to: register(ra, cap(code)))
24    ),
25    procedures: (
26      sum,
27      in: do(
28        pushScope
29        set(abstract(cc.retcap), to: register(ra, cap(code)))
30        set(abstract(first), to: register(a0, s32))
31        set(abstract(second), to: register(a1, s32))
32        compute(abstract(the_result), first, add, second)
33        set(register(a0), to: the_result)
34        set(register(ra), to: cc.retcap)
35        clearAll(except: a0 ra)
36        popScope
37        return(to: register(ra, cap(code)))
38      )
39    )
40  )
```

4.3.2 Local State Encapsulation & Unrepeatable Return

As part of an s-call, the callee receives arguments and a return–frame capability pair sealed with a unique object type for which the callee has no unseal capability and therefore cannot dereference. The callee thus cannot access the caller's local state, thereby preserving local state encapsulation. The capabilities are only unsealed when control is returned to the caller using the atomic `CInvoke` instruction. Since the callee also does not have a seal capability for that object type, it cannot forge a call stack to manipulate the procedure's local state, nor can it forge a return capability to trick `CInvoke` into unsealing the sealed frame capability before invoking adversarial code.

Before invoking the s-call routine, the caller initialises an indicator value to 0 and stores it in its call frame. When the callee returns control to the caller, the caller checks if the value is still 0 and crashes otherwise. It then sets the value to 1 to ensure that subsequent uses of the return capability result in a machine trap, thereby ensuring that the return capability is only usable once. Since local state encapsulation protects the call frame from external access, it also protects the indicator value from the same.

The latter security property, which we call **unrepeatable return**, is a weaker variant of well-bracketed control flow and can be stated as follows: **a return capability provided by a procedure supporting unrepeatable return can be used at most once**. Unlike well-bracketed control flow, it does not disallow a callee to return to another procedure's caller, as illustrated in Fig. 4.4.

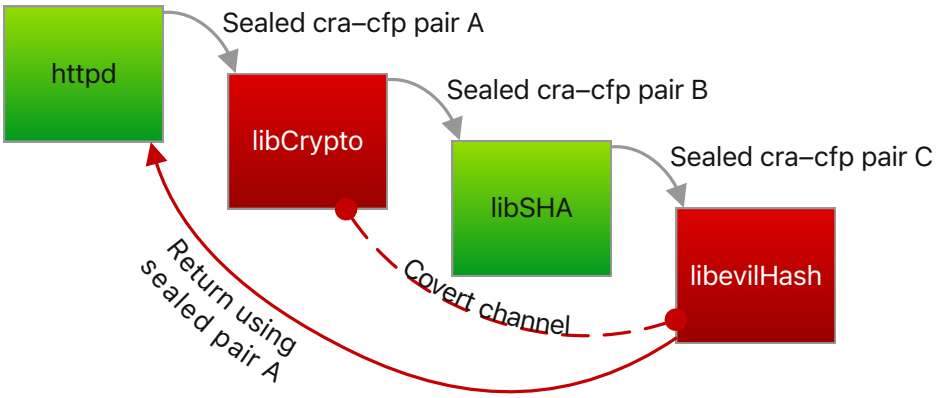


FIGURE 4.4: GHSCC does not support well-bracketed control flow. In this example, libCrypto sends the return–frame capability pair it receives from httpd to libevilHash, which uses it to return directly to httpd, thereby leaving libSHA (and libCrypto) waiting to be returned to.

4.4 Evaluation

4.4.1 Impact on the Codebase

We now evaluate the impact of the implementation of GHSCC in Glyco which can serve as an indicator to the development effort required for adding a new calling convention to an existing nanopass compiler with a similar design. We first present the most significant changes to Glyco's languages, going from low-level to high-level ILs, then quantify the codebase's growth. The changes to the compiler pipeline are also summarised in Fig. 4.5.

CHERI-RISC-V (RV) The low-level language RV is extended with additional grammar for statements such as `padding` and `bssSection`, which can be lowered trivially to their CHERI-RISC-V assembly counterparts. Instructions are now merely one kind of statement, and programs are redefined as a list of statements. This additional grammar is required by MM which now explicitly manages regions of memory for use by the call stack, the heap, or runtime routines.

Canonical Effects (CE) To avoid overloading MM's nanopass with responsibilities, a new IL is defined above RV grouping related instructions. Instructions such as `copyWord` and `copyCapability` are for instance exposed under a single `copy` effect in CE, which MM inherits.

Runtime (RT) To facilitate calling runtime routines (which behave differently than ordinary procedures), a new IL RT is defined above CE providing a `callRuntimeRoutine` effect. This effect accepts a label to a capability located in memory accessible to the user program, loads the capability, and jumps to its address. These capabilities are sentry capabilities to runtime memory; user programs cannot dereference these capabilities but they can jump to their address. This prevents a user program from accessing the runtime's internal data structures like the heap capability.

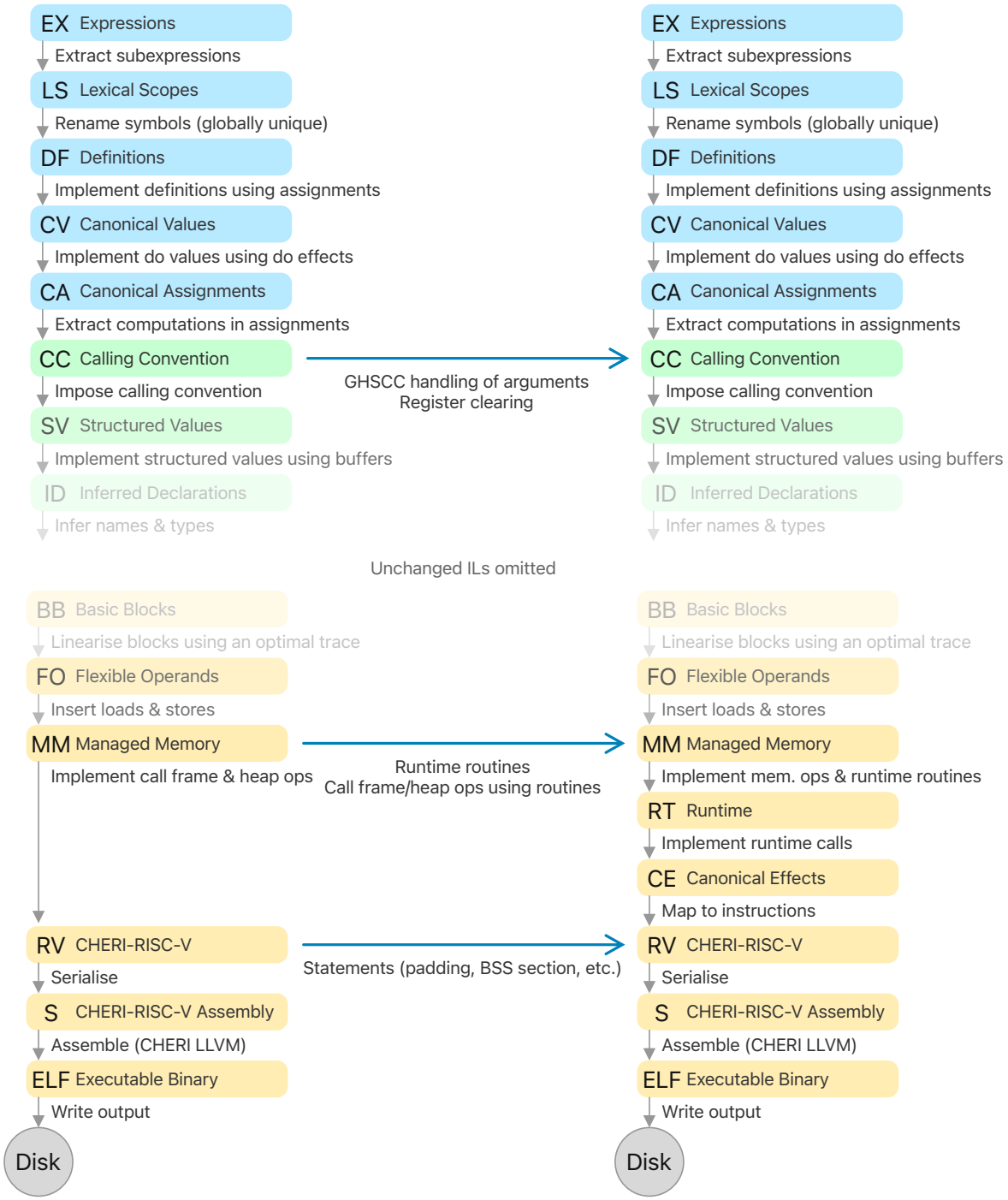


FIGURE 4.5: The Glyco 0.1 (left) and 0.2 (right) pipelines and the main differences between them.

Managed Memory (MM) MM moves above RT but its grammar remains mostly the same as in the previous version. The MM (to RT) nanopass however changes significantly. When a `Program` is lowered, MM now includes runtime initialisation code, explicit ELF sections for the call stack and heap, and sentry capabilities to the allocation and s-call runtime routines. The lowering of the `createBuffer` effect is updated to invoke the allocation routine instead of directly manipulating the heap capability (previously kept in register `ctp`). The `pushFrame` effect now allocates a call frame on the heap.

Calling Convention (CC) The `call` effect in CC is updated to allocate an arguments record for arguments that do not fit in the available argument registers, and to clear registers before calling the procedure.

Codebase growth The implementation of GHSCC introduces 2 new ILs (RT and CC) bringing the total in Glyco 0.2 to 19 ILs. Using cloc,[7] we measured a growth in the codebase from 5941 SLOC in Glyco 0.1 to 6921 SLOC in Glyco 0.2, representing a net growth of +16%.[8]

The compiler's implementation does not rely on a framework such as one described by Keep and Dybvig [2013] that provides a compact syntax for deriving a new IL given an existing IL. New ILs are instead derived by duplicating parts of the lower language's source files and adapting them for the new IL. The SLOC metric is therefore not as informative as the number of added ILs or changes per IL.

4.4.2 Impact on Build Products

We now evaluate GHSCC itself by benchmarking a few test programs against the previous calling convention.

Increased binary footprint For measuring the binary footprint, we decided to measure the number of statements and instructions in the assembly of a few test programs as opposed to measuring the byte size of the executables themselves. The latter metric is less informative since the space taken up by ELF headers and structures dominates the space occupied by the code of interest.

We compile the test programs in Glyco 0.2 using a default configuration targeting the CHERI-RISC-V Sail emulator and using the GCCC and GHSCC calling conventions. The default configuration enables all optimisations provided by Glyco.

The minimalistic test program 42 immediately evaluates to the number 42. The fib test program computes the 30th or 300th number of the Fibonacci sequence using a recursive function with accumulator parameters to avoid redundant computations. The fibvec test program does the same but by computing the sequence in a vector.[9] The Euclid test program computes the greatest common divisor of 20 and 50 or of 441 and 520 using a

[7] cloc is available at `https://github.com/AlDanial/cloc`.

[8] The source line of code metric (SLOC) excludes empty lines and comments but does not perform other normalisation. For instance, multiple statements on a single line still count as 1 SLOC.

[9] Vectors are stack-allocated in the GCCC configuration in Glyco 0.2.

simple subtraction algorithm. As shown in Table 4.1, we can observe a visible increase in binary footprint.

	GCCC				GHSCC			Δ		
Program	Size	Time	Stack	Heap	Size	Time	Heap	Size	Time	S + H
Runtime	75	—	—	—	110	—	—	+47%	—	—
42	45	107 t	32 B	0 B	66	207 t	96 B	+47%	+93%	+200%
fib(0, 1, 30)	53	705 t	1.50 kB	0 B	114	2.41 kt	2.51 kB	+115%	+241%	+67%
fib(0, 1, 300)	53	1.61 kt	14.4 kB	0 B	114	22.1 kt	15.4 kB	+115%	+1278%	+7%
fibvec(0, 1, 30)	106	1.34 kt	1.58 kB	0 B	198	3.00 kt	2.91 kB	+87%	+124%	+84%
fibvec(0, 1, 300)	106	12.9 kt	15.6 kB	0 B	198	28.9 kt	15.6 kB	+87%	+123%	+0%
Euclid(20, 50)	62	170 t	208 B	0 B	151	441 t	1.36 kB	+144%	+159%	+554%
Euclid(441, 520)	62	478 t	880 B	0 B	151	1.49 kt	2.08 kB	+144%	+212%	+136%

TABLE 4.1: Benchmarking test programs in GCCC and GHSCC in Glyco 0.2. The program size is measured in number of assembly statements and is split between the runtime and the rest of the program. Time is measured in ticks (or kiloticks), with one instruction being executed per tick. Stack and heap sizes are maxima reached during execution. 1 kB = 1000 B.

Increased runtime and memory overhead For measuring runtime overhead, we execute the same test programs in the CHERI-RISC-V Sail emulator, counting the number of steps to completion. For memory overhead, we determine the distance between the highest and lowest address of the stack capability and heap capability (after runtime initialisation). As shown in Table 4.1, we can observe a significant degradation in performance when using GHSCC in a function call-heavy program.

4.5 Related Work: Alternative Secure Calling Conventions

The heap-based calling convention proposed in this chapter and implemented in Glyco is but one calling convention providing local state encapsulation, albeit without a rigorous proof. Several alternatives have been proposed in literature or are implemented in CHERI software, some of which also provide well-bracketed control flow.

To ensure local state encapsulation with a stack-based calling convention, a caller can restrict the stack capability before calling a procedure so that the callee only gets access to the unused part of the stack. However, this approach requires a way for the caller to restore the previous (less restricted) stack capability without the callee being able to do the same. Additionally, as illustrated in Fig. 4.6, an adversary can save a copy of their (restricted) stack capability in a first call. If the adversary is called a second time when their stack capability is lower in the stack (due to the stack containing more call frames than during the first call), it can use the saved stack capability to gain access to memory used by other call frames. They can similarly store the return capability and use it more than once, breaking well-bracketed control flow.

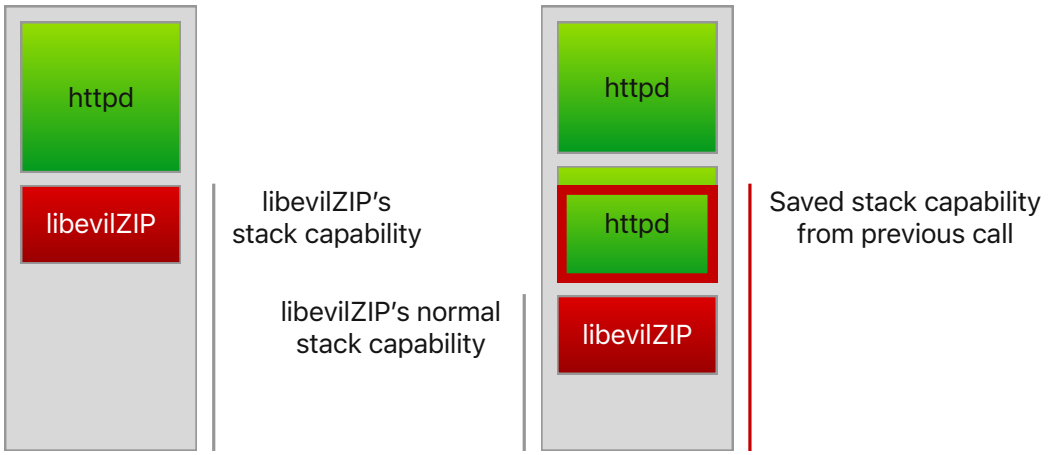


FIGURE 4.6: The call stack in a first invocation (left) and a second invocation (right) of the adversary, where the second invocation's call frame is at a lower address than the first invocation's call frame. During the first invocation, the adversary saves their stack capability for use during a later invocation. Even though the stack capability in the second invocation provides authority over a smaller part of the call stack, the previously saved stack capability continues to provide the adversary with authority over the larger region, which now includes memory occupied by other call frames. Note that the stack grows downward.

Using local capabilities to revoke copies of the stack capability A calling convention proposed by Skorstengaard et al. [2019a] relies on **local capabilities**, a type of capability that can be only kept in registers and stored using capabilities that allow storing local capabilities. Local capabilities in CHERI are capabilities without the *Global* permission, whereas capabilities that can be used to store local capabilities have the *Store local capability* permission.

Just before a procedure is called, the caller pushes a restoration routine[10] on the call stack, sets the return capability to point to that routine, and seals it as a sentry capability. The restoration routine restores the caller's stack capability and returns to the caller; the stack and return capabilities restored by the routine are therein embedded.

To prevent the callee from saving their return and stack capabilities somewhere, the two capabilities are marked as local. However, they still need to be saved in the restoration routine. Since this routine is part of the call stack, the stack capability is given the *Store local capability* permission. This permission is not given to capabilities to any other region of memory, ensuring that return and stack capabilities cannot be saved elsewhere. Since the adversary can still save these capabilities somewhere in stack memory, e.g., far from any call frames where it is unlikely to be overwritten, the unoccupied part of the stack must be cleared before calling or returning to a possible adversary.

The calling convention guarantees both local state encapsulation and well-bracketed control flow since an adversary cannot retain either a stack or return capability after it returns. However, it requires clearing of potentially large regions of unoccupied stack

[10] Skorstengaard et al. [2019a] uses the term *activation record* instead, but since it is commonly also a synonym for *call frame* and *record* has a different meaning in Glyco, we decided against reusing this terminology in this thesis.

memory which is inefficient if no hardware support is available. Furthermore, since the call stack needs to hold executable restoration routines, the stack capability needs the *Permit execute* permission, and thus special care must be taken to avoid accidentally executing code stored in input buffers.

Even though this calling convention's security properties have been proven using a mechanised proof, we did not consider implementing this calling convention in Glyco because no emulator supports efficient stack clearing and manual stack clearing would dominate the test programs' runtime.

Using local uninitialised capabilities to revoke copies of the stack capability A calling convention proposed by Huyghebaert [2020], Georges et al. [2021b] builds on the aforementioned calling convention but removes the requirement for clearing large regions of stack memory. It instead relies on **uninitialised capabilities**, a type of capability proposed by Huyghebaert [2020] only allowing loads on an initialised part of its bounds. Storing a datum immediately after the initialised region extends the capability's initialised region to include the newly stored datum.

In this calling convention, the restoration routine still restores the caller's stack capability except that the stack capability is now an uninitialised capability. This saved stack capability has a smaller initialised region, effectively revoking access to the unoccupied stack memory's contents. The next procedure called by this procedure will need to overwrite any unoccupied stack memory before accessing it.

A callee still needs to clear the stack memory occupied by their own call frame before returning to a potential adversary since the caller may have prepared a separate stack capability granting it access to the callee's call frame.[11] However, the memory occupied by a single call frame is orders of magnitude smaller than the full unoccupied stack memory that needs to be otherwise cleared at every call or return.

Similarly to the previous calling convention, it is rigorously proven for an abstract machine with local and uninitialised capabilities. Uninitialised capabilities are however as of writing not fully specified for CHERI-RISC-V in an authoritative document such as by Watson et al. [2020].

Using linear capabilities to restrict copies of the stack capability The *StkTokens* calling convention by Skorstengaard et al. [2019b] relies on **linear capabilities**, a type of capability that can only be moved, not copied. Before calling a procedure, the caller performs a splitting operation which splits the stack capability into two disjoint capabilities. The stack capability over the caller's call frame as well as the callee's return capability are sealed together with a unique seal capability. The caller then hands the sealed pair as well as the second (still unsealed) stack capability over to the callee.

The callee returns control to the caller by passing the unsealed stack capability to the caller (e.g., via the dedicated stack capability register) and invoking the sealed stack–return capability pair. The machine unseals both capabilities and jumps to the caller. The caller merges the two stack capabilities back to get its original stack capability, over both its call frame and the unused part of the stack.

[11] The caller still has a call frame where it can validly store any local capability.

Local state encapsulation is guaranteed by the stack capability's linearity: the callee must yield its unsealed stack capability to the caller before the caller can reconstruct its previous stack capability. In addition, the callee only gets a sealed stack capability to the caller's call frame and thus is not granted access to the call frame, assuming the caller uses a unique seal capability that it does not hand to the callee.

Well-bracketed control flow is guaranteed by the return capability's linearity: the callee can only return to the caller by invoking the stack–return capability pair, losing access to both capabilities in the process. The callee also cannot return to a caller deeper in the stack as that (deeper) caller is unable to reconstruct their stack capability without its callee's cooperation.

Although this calling convention is delightfully straightforward in its design, linear capabilities are as of writing not authoritatively specified for CHERI-RISC-V.

Using one call stack per compartment As described by Watson et al. [2015], CheriBSD supports compartmentalising software components (such as libraries) whereby each component gets its own call stack, with the operating system managing a central stack per thread. This approach however requires one call stack per component per thread, one central stack per thread, and one syscall per security domain transition, i.e., when a compartment's procedure calls or returns into another compartment's procedure. Furthermore, it does not provide complete local state encapsulation since it only protects components from other components; a vulnerability in a procedure within a library may compromise all call frames from that library in the same thread. Similarly, well-bracketed control flow across components can be enforced by the system since returns across components are mediated by the OS, but the property does not apply within a single component.

Chapter 5

Sealed Objects

The previous chapter discusses a first extension of the Glyco compiler, namely a secure calling convention. This chapter treats a second extension, a feature we call *sealed objects*.

Sealed objects depend on two new features; we begin hence by describing lambdas in Section 5.1 and named & nominal types in Section 5.2 before defining the semantics of objects and methods in Section 5.3. We then list in Section 5.3.1 a few security properties afforded by sealed objects. We finish this chapter by evaluating the changes to the compiler in Section 5.4.

This chapter discusses the feature set and languages of Glyco 0.3.[1] A full language reference can be found in Appendix A.

5.1 Lambdas

A first addition to Glyco is the **lambda**, i.e., an anonymous function, in a new IL called Λ (Lambdas) above EX. Lambdas allow the programmer to define functions (λ values) at the point of use and to pass them around as values. For example, the Λ program in Listing 6 defines a lambda that computes the sum of its two parameters and immediately applies it on 1080 and −80.

```
1  (evaluate(
2    λ(takes: (first, s32) (second, s32), returns: s32, in:
3      value(binary(first, add, second))
4    ), 1080 -80
5  ))
```

LISTING 6: A Λ program evaluating to 1000.

The scope of a lambda definition extends beyond the λ value itself to the `let` value defining it, in a similar way to `letRec` values in some functional programming languages and unlike definitions of other kinds of values. This enables mutual recursion such as the (inefficient) program in Listing 7 determining the parity of 420.

[1] The source code is available at https://tsarouhas.eu/glyco/0.3/.

```
(let(
  (even, λ(takes: (n, s32), returns: s32, in:
    if(
      relation(n, le, 0),
      then: value(1),
      else: evaluate(odd, binary(n, sub, 1))
    )
  ))
  (odd, λ(takes: (n, s32), returns: s32, in:
    if(
      relation(n, le, 0),
      then: value(0),
      else: evaluate(even, binary(n, sub, 1))
    )
  )), in:
  evaluate(even, 420)
))
```

LISTING 7: A Λ program featuring mutual recursion.

Finally, since λ values with support for mutual recursion provide the same expressive power as programs with global functions, Λ removes support for global functions. The reader should bear in mind however that lambdas do not *increase* the (practical) expressive power of programs but merely make it easier to define functions. Lambda values evaluate to code capabilities and do not carry an environment, i.e., lambdas in Λ are not closures.[2]

From Λ to EX The nanopass from Λ to EX extracts the functions defined by λ values into the global scope (with either an auto-generated name or a name derived from the lambda definition) and replaces the λ values by code capabilities to those functions. The program in Listing 6 is thus lowered to the EX program

```
(
  evaluate(function(l.anon), 1080 -80),
  functions: (
    l.anon,
    takes: (first, s32) (second, s32),
    returns: s32,
    in: value(binary(first, add, second))
  )
)
```

The nanopass also injects all λ values in the same `let` value in the lambda body to enable (mutual) recursion. The parity program in Listing 7 is thus lowered as

```
(
  let((even, function(l.even)) (odd, function(l.odd)), in: evaluate(even, 420)),
  functions: (
    l.even,
    takes: (n, s32),
    returns: s32,
```

[2] Spoiler alert: Chapter 6 introduces closures.

```
    in: let(
      (even, function(l.even)) (odd, function(l.odd)),
      in: if(relation(n, le, 0), then: value(1), else: evaluate(odd, binary(n, sub, 1)))
    )
  )
  (
    l.odd,
    takes: (n, s32),
    returns: s32,
    in: let(
      (even, function(l.even)) (odd, function(l.odd)),
      in: if(
        relation(n, le, 0),
        then: value(0),
        else: evaluate(even, binary(n, sub, 1))
      )
    )
  )
)
```

5.2 Alias & Nominal Types

A second new feature is support for defining new (**named**) **types** in a new IL **NT** (Named Types) above Λ. A value in NT can be

- an (8-bit) byte, a `u8`;
- a (32-bit) signed integer, an `s32`;
- a capability to a vector of `T`s, a `cap(vector(T, sealed: S))`;
- a capability to a record, a `cap(record((name, Type) ..., sealed: S))`;
- a capability to a function, a `cap(function(takes: p ..., returns: r))`;
- a seal capability, a `cap(seal(sealed: S))`; or
- a value of a type named `T`

where `S` is `true` if the capability is sealed and `false` otherwise. A type can be defined using a **`letType` value** in one of two ways.

Alias types An **alias type definition** creates an **alias type**, a type that is equivalent to the type it is defined as. An alias type can have a shorter or semantically meaningful type name that the type it is defined as.

For instance, the following NT program defines an alias type named `Sequence` and uses it in a simple computation that evaluates to 4. The value `pi` created by `vector(0, count: 3)` is typed `cap(vector(s32, sealed: false))` but is accepted by the function which takes an argument typed `Sequence`.

```
(value(letType(
  alias(Sequence, cap(vector(of: s32, sealed: false))), in:
  let(
    (sumOfFirstAndSecond, λ(takes: (sequence, Sequence), returns: s32, in:
      value(binary(element(of: sequence, at: 0), add, element(of: sequence, at: 1)))
    ))
    (pi, vector(0, count: 3)),
```

```
    in: do(
      setElement(of: pi, at: 0, to: 3)
      setElement(of: pi, at: 1, to: 1)
      setElement(of: pi, at: 2, to: 4), then:
      evaluate(sumOfFirstAndSecond, pi)
    )
  )
)))
```

Nominal types A **nominal type definition** creates a **nominal type**, a type which is only equivalent to itself. A nominal type is effectively a new type by itself and can be used to ensure that values of two representationally identical but semantically different types cannot accidentally be mixed. However, a value can be explicitly `casted` to a value of a different type that has the same representation.

For example, the following NT program defines two nominal types `Kelvin` and `Celsius` and uses them to convert 500 °C to K and back to °C.

```
(value(letType(
  nominal(Kelvin, s32) nominal(Celsius, s32), in:
  let(
    (toKelvin, λ(takes: (c, Celsius), returns: Kelvin, in:
      value(cast(binary(c, add, 273), as: Kelvin))
    ))
    (toCelsius, λ(takes: (k, Kelvin), returns: Celsius, in:
      value(cast(binary(k, sub, 273), as: Celsius))
    )),
    in: evaluate(toCelsius, evaluate(toKelvin, 500))
  )
)))
```

However, the following program is not valid since the second invocation of `toCelsius`, which expects a value in `Kelvin`, is given a value in `Celsius`. Nominal type rules prevent the accidental conversion from `Kelvin` to `Celsius`, even though both types are represented by `s32`.

```
(value(letType(
  nominal(Kelvin, s32) nominal(Celsius, s32), in:
  let(
    (toKelvin, λ(takes: (c, Celsius), returns: Kelvin, in:
      value(cast(binary(c, add, 273), as: Kelvin))
    ))
    (toCelsius, λ(takes: (k, Kelvin), returns: Celsius, in:
      value(cast(binary(k, sub, 273), as: Celsius))
    )),
    in: evaluate(toCelsius, evaluate(toCelsius, 500))
  )
)))
```

```
Error:  evaluate(toCelsius, 500) is of type named(Celsius) and thus cannot
be used for (k, Kelvin)
```

Implicit type casts With some exceptions, the type checker rejects any uses of values of a structural type such as `s32` in contexts where a nominal type is expected and vice versa. This restriction can be bypassed using a `cast`. For ergonomic reasons, the type-checker performs an implicit cast on

- an argument of structural type that is passed to a parameter of nominal type, like in `evaluate(toKelvin, 500)` above;
- a record value of nominal type in a `field` value;
- a vector value of nominal type in an `element` value;
- a seal value of nominal type in a `sealed` value;
- an operand of structural type in a `binary` value when the other operand is of nominal type — the result of the `binary` value is in that case of nominal type;
- a lambda's result value of structural type when the result type is nominal; and
- a program's result value of nominal type (to `s32`), like the result in `Celsius` above.

From NT to Λ The nanopass from NT to Λ replaces named types by their Λ equivalents and checks if the nominal typing rules hold.

5.3 Objects & Methods

Objects in object-oriented programming languages are constructs that consist of state and behaviour. An object owns a region of memory where it holds its state while its behaviour manifests through its methods, which are functions that access and modify the object's state as part of their operation.

Objects usually provide encapsulation, a technique that discourages or prohibits code outside of an object's method from accessing or modifying the object's state, thereby promoting the decoupling of disparate parts of a codebase. To illustrate this, consider the following NT program.

```
(
  value(
    letType(nominal(Counter, cap(record(((value, s32)), sealed: false))),
      in: let(
        (increaseCounter, λ(takes: (counter, Counter), returns: s32, in:
          let((newValue, binary(field(value, of: counter), add, 1)),
            in: do(
              setField(value, of: counter, to: newValue),
              then: value(newValue)
            )
          )
        ))
        (getCounterValue, λ(takes: (counter, Counter), returns: s32, in:
          value(field(value, of: counter))
        ))
        (counter, record((value, 32)))
        (ignored, evaluate(increaseCounter, counter))
        (ignored, evaluate(increaseCounter, counter))
        (ignored, evaluate(increaseCounter, counter)),
        in: evaluate(getCounterValue, counter)
      )
```

```
22      )
23    )
24  )
```

This program implements a `Counter` type, which is a record consisting of a single integer value, and two lambdas that return resp. increase the counter's value. It then creates a `Counter` with an initial value of 32 and increases it three times.

`counter` is here an "object" and `increaseCounter` and `getCounterValue` are the "methods" which access or modify the `counter`'s state. However, nothing prevents from external code from directly accessing the `value` field, for example to set it to zero by doing `setField(value, of: counter, to: 0)` anywhere where `counter` is accessible. That is, `Counter` "objects" in the example above are not encapsulated.

Sealed objects & methods A **sealed object** (or simply **object** in this chapter) is an encapsulated record (the object's **state**) on which a predetermined set of methods can be invoked. A **method** is a function that (among other parameters) takes a capability to an object's state. We refer to the object on which a method is called as the method's **receiver**.

Object types & initialisers An object belongs to an **object type**, which determines the record type and methods of all objects of that type. Object types in Glyco are in this respect similar to classes in programming languages such as Java. Object types however do not support inheritance and record fields cannot be made visible outside of methods.

Object types can be defined in a new IL above NT called **OB** (Objects) using an **object type definition** in a `letType` value. A definition consists of

- a name, like any other type definition;
- a record type specifying the structure of each object's state;
- an initialiser effect applied on all new objects accepting zero or more parameters; and
- zero or more methods.

An **initialiser** is a pseudo-method automatically invoked on every new object immediately after its state is allocated that ensures that the object has the right initial state. An object can be created using an `object` value, which accepts a type name and arguments to the initialiser's parameters. Methods can only be defined when the object type is being defined. They can be invoked using a `message` value, which takes an object, method name, and arguments to the method's parameters. Initialisers and methods alike get a capability to the object's state, i.e., the record composing it, through the `self` value.

The OB program in Listing 8 reimplements the `Counter` type as an object type, creates a counter with an initial value of 32, increases the count three times, and evaluates to the counter's final value (35).

5.3.1 Object State Encapsulation & Unique Seals

The first important security property afforded by sealed objects, the **encapsulation property**, is that an object's state is **only accessible within the initialiser or a method defined**

```
(value(
  letType(
    object((
      Counter,
      initialiser:
        (takes: (initialValue, s32), in:
          record((value, initialValue))
        ),
      methods:
        (increase, takes:, returns: s32, in:
          let((newValue, binary(field(value, of: self), add, 1)), in:
            do(
              setField(value, of: self, to: newValue),
              then: value(newValue)
            )
          )
        )
        (getCount, takes:, returns: s32, in:
          value(field(value, of: self))
        )
    )), in:
    let(
      (counter, object(Counter, 32))
      (ignored, evaluate(message(counter, increase),))
      (ignored, evaluate(message(counter, increase),))
      (ignored, evaluate(message(counter, increase),)), in:
      evaluate(message(counter, getCount),)
    )
  )
))
```

LISTING 8: An OB program implementing a counter and counting from 32 to 35. The `evaluate(message(counter, increase),)` syntax is valid in Glyco 1.0 and is introduced in the next chapter. The equivalent syntax in this chapter's version (Glyco 0.3) is `message(counter, increase,)`. Additionally, initialisers in Glyco 0.3 receive a preallocated record in `self` while initialisers in the final version (shown here) allocate themselves a record.

during object type definition in `letType`.[3] A second security property is that a method or initialiser **can only be invoked on an object of the type defining that method or initialiser**.[4]

Both properties are achieved by sealing capabilities to objects and methods using unique seal capabilities provided by a runtime routine. Sealing capabilities using a seal capability is a power which was originally bestowed upon the runtime in the design and implementation of GHSCC where an s-call seals a return–frame capability pair before passing control to the callee.

[3] This property does not hold if the initialiser or a method leaks an unsealed capability to the state record. Our threat model assumes trust in an object type's implementation.

[4] This property similarly does not hold if an unsealed code capability to the method is leaked by the type owner.

An **object capability** is a sealed capability to a record embodying an object. A **method capability** is a sealed code capability to a method. Due to their sealed nature, object capabilities cannot be used in an `getField` value or `setField` effect and similarly method capabilities cannot be used in an `evaluate` value without a receiver of the right object type.[5]

Object and method capabilities belonging to the same object type are sealed using the same seal capability; a method capability can therefore be invoked on an object capability using the `CInvoke` CHERI-RISC-V instruction. A method capability cannot be invoked on an object capability belonging to a different object type; the hardware ensures this invariant when the `CInvoke` instruction is executed, with a violation causing a machine trap.

5.3.2 Object Types Are Objects

When an object type is defined in a `letType` value, a unique seal capability is created and used for sealing the new type's method capabilities. This capability must be protected from adversaries as it allows for additional methods to be defined. With access to the seal capability, the adversary merely needs to seal a method capability of its own and invoke the adversarial method on the sealed object to get access to the object's state, thereby breaking the encapsulation property.

An object type's seal capability is also needed for sealing new object capabilities, i.e., for creating new objects of the object type. To allow an adversary to create new objects of a type it does not own, Glyco introduces the concept of an **object type object**, an object representing an object type, and the notion of a **metatype**, the type of a type object. The metatype provides a `createObject` method with the same parameters as those of the type's initialiser and which produces an object capability. The seal capability needed for sealing the capability to the initialised state of the new object is stored within the type object and only accessible from within the `createObject` method by relying on the security properties afforded by objects.

Each object type has an associated type object and metatype. The metatype (the type object's type) does not define an initialiser; instead, the singleton type object's state is statically allocated and initialised at the point of the object type definition and the metatype's seal capability is discarded after use.

5.3.3 Lowering OB to NT

We'll work through the OB to NT nanopass by applying it on the `Counter` example in Listing 8.

```
1 (
2   value(
```

[5] A `getField` value is eventually lowered to a load instruction in the CHERI-RISC-V language. A `setField` effect is eventually lowered to a store instruction. An `evaluate` value is eventually lowered to either a `CJALR` or `CInvoke` instruction. Loading or storing using a sealed capability, jumping to a sealed code capability using `CJALR`, or invoking a mismatched code–data capability pair using `CInvoke` results in a machine trap.

An object type definition (in a `letType`) is lowered to NT by

1. defining a nominal type definition;

```
3       letType(
4         nominal(Counter, cap(record(((value, s32)), sealed: true))),
5         in: let(
```

2. creating unique seal capabilities for the metatype and the new type;

```
6           (ob.tseal, seal)
7           (ob.oseal, seal)
```

3. allocating and initialising a type object with the type's seal capability;
4. sealing the capability to this object with the metatype's seal capability;

```
8           (
9             ob.Counter.type,
10            let(
11              (ob.typeobj, record((seal, ob.oseal))),
12              in: sealed(ob.typeobj, with: ob.tseal)
13            )
14          )
```

5. implementing the metatype's `createObject` method using a lambda that allocates a state record, executes the initialiser's effect, seals a capability to the state record using the seal capability stored in the type object, and returns this sealed capability to the method's caller;
6. sealing the capability for this method with the metatype's seal capability;

```
15          (
16            ob.ob.Counter.Type.createObject.m,
17            sealed(
18              λ(
19                takes: (
20                  ob.self,
21                  cap(record(((seal, cap(seal(sealed: false)))), sealed: true)),
22                  sealed: true
23                )
24                (initialValue, s32),
25                returns: cap(record(((value, s32)), sealed: true)),
26                in: let(
27                  (ob.seal, field(seal, of: ob.self)),
28                  in: value(sealed(record((value, initialValue)), with: ob.seal))
29                )
30              ),
31              with: ob.tseal
32            )
33          )
```

7. implementing each method of the type using a lambda that takes a sealed parameter `ob.self` representing the receiver (and which the `self` value maps to) plus any parameters defined on the method; and finally,
8. sealing the capability of each of the type's methods.

```
34          (
35            ob.Counter.increase.m,
36            sealed(
```

```
37              λ(
38                takes: (
39                  ob.self,
40                  cap(record(((value, s32)), sealed: true)),
41                  sealed: true
42                ),
43                returns: s32,
44                in: let(
45                  (newValue, binary(field(value, of: ob.self), add, 1)),
46                  in: do(
47                    setField(value, of: ob.self, to: newValue),
48                    then: value(newValue)
49                  )
50                )
51              ),
52              with: ob.oseal
53            )
54          )
55          (
56            ob.Counter.getCount.m,
57            sealed(
58              λ(
59                takes: (
60                  ob.self,
61                  cap(record(((value, s32)), sealed: true)),
62                  sealed: true
63                ),
64                returns: s32,
65                in: value(field(value, of: ob.self))
66              ),
67              with: ob.oseal
68            )
69          ),
```

Each method invocation is lowered by evaluating the method capability with the receiver as the first argument, followed by any provided arguments. Object construction in `object` values is implemented in NT by calling the metatype's `createObject` method capability on the type object.

```
70        in: let(
71          (counter, evaluate(ob.ob.Counter.Type.createObject.m, ob.Counter.type 32))
72          (ignored, evaluate(ob.Counter.increase.m, counter))
73          (ignored, evaluate(ob.Counter.increase.m, counter))
74          (ignored, evaluate(ob.Counter.increase.m, counter)),
75          in: evaluate(ob.Counter.getCount.m, counter)
76        )
77      )
78    )
79  )
80)
```

5.3.4 Seal Creation Routine

Glyco 0.3 generalises the s-call runtime routine introduced in Chapter 4 (cf. Listing 5) to a **seal capability creation routine**, which is invoked by the `seal` effect. Procedure calls in the GHSCC configuration now create a unique seal, seal the frame–return capability pair, and pass control directly to the callee, instead of delegating these tasks to an s-call routine. The two security properties provided by GHSCC (local state encapsulation and unrepeatable return) are unaffected by this change since the seal capability created by the seal creation routine is kept (or can be kept) private by the caller.

An assembly excerpt of the seal creation routine is given in Listing 9.

```
107 mm.cseal:       cllc ct1, mm.cseal_seal_cap
108                 clc ct6, 0(ct1)
109                 cincoffsetimm ct6, ct6, 1
110                 csc ct6, 0(ct1)
111                 csetboundsimm ct6, ct6, 1
112                 .4byte 4276158555 # cclear 0, 64
113                 cjalr cnull, ct0
114                 .balign 16
115 mm.cseal_seal_cap: .octa 0
116 mm.cseal_end:   .balign 4
```

LISTING 9: Assembly code of the seal creation routine as inserted in each program. The CHERI LLVM assembler did not support the `cclear` instruction at the time of development so it is encoded directly by Glyco instead. The seal capability is stored at `mm.cseal_seal_cap` and is initialised by the initialisation routine (which is part of the runtime).

5.4 Evaluation: Impact on the Codebase

Similarly to what we did in Section 4.4, we evaluate the impact of the implementation of sealed objects in Glyco. The changes to the compiler pipeline are also summarised in Fig. 5.1.

Managed Memory (MM) The s-call runtime routine is generalised to a seal allocation routine, given that both s-calls and object type definitions only need a runtime routine to securely provide a unique seal capability for the next invocation resp. object type definition. The language now provides new `createSeal` and `seal` effects. The `call` effect's lowering under GHSCC is modified to invoke the seal allocation routine and use the returned seal capability to seal the return–frame capability pair.

A `callSealed` effect is also added to the IL's syntax which performs roughly the same as the `call` effect, except that it accepts a sealed code–data capability pair, as required for method invocation.

Flexible Operands (FO) The `createSeal`, `seal`, and `callSealed` effects from MM are propagated to FO. Just like for other FO effects, the FO to MM nanopass inserts any required loads and stores around the lowered effect.

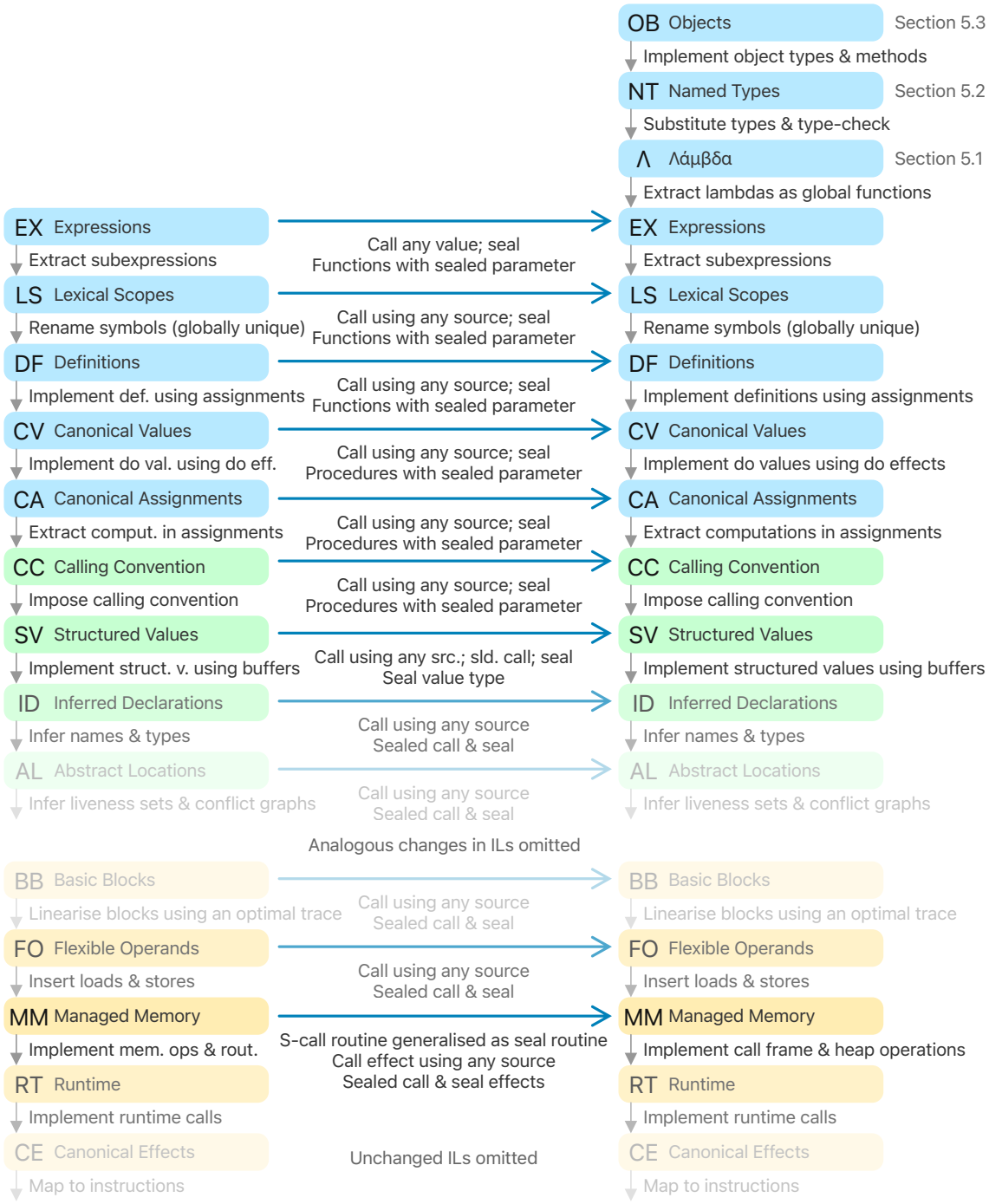


FIGURE 5.1: The Glyco 0.2 (left) and 0.3 (right) pipelines and the main differences between them.

The `call` effect is updated to accept any kind of source, not just labels. This is needed to enable calling code capability values, which is how lambdas are represented across the compiler pipeline.

Basic Blocks (BB) and Predicates (PR) The `createSeal`, `seal`, and updated `call` effects from FO are propagated to BB and PR. A new `callSealed` continuation is added.

Conditionals (CD) through Structured Values (SV) The 3 new effects and the updated `call` effect are propagated through these 5 ILs, usually with trivial lowerings.

Structured Values (SV) An additional seal capability value type is added. The `createSeal` effect creates capabilities of this type while the `seal` effect uses them.

Calling Convention (CC) The `createSeal`, `seal`, and updated `call` effects from SV are propagated to CC but the `callSealed` effect is not. Instead, individual parameters can now be marked as being sealed. The `call` effect is lowered using a `call` resp. `callSealed` effect in SV when the target procedure has zero resp. one sealed parameter(s). CC does not support calling procedures with two or more sealed parameters.[6]

Canonical Assignments (CA) The `createSeal` and `seal` effects from CC are propagated to CA as sources. The `call` effect is updated to accept any kind of source for its procedure operand instead of only accepting labels.

Computed Values (CV) through Expressions (EX) The `createSeal` and `seal` sources from CA as propagated to CV, DF, LS, and EX as values. The `evaluate` value is similarly updated to accept any kind of source for its procedure operand.

Lambdas (Λ) through Objects (OB) These new ILs are described in the preceding sections.

Codebase growth The implementation of lambdas, named types, and sealed objects introduces 3 new ILs (Λ, NT, and OB) bringing the total in Glyco 0.3 to 22 ILs. Using cloc again, we measured a growth in the codebase from 6921 SLOC in Glyco 0.2 to 9360 SLOC in Glyco 0.3, representing a net growth of +35%. However, as noted in Section 4.4, a significant part of this growth is due to code duplication.

[6] One way to implement support for such procedures is to transform each procedure taking multiple sealed parameters to first call itself multiple times, each time unsealing a different argument.

Chapter 6

Sealed Closures

The preceding chapter introduces sealed objects, objects whose state can only be accessed via dedicated methods. This chapter presents an application of sealed objects, a new feature we call *sealed closures*.

Section 6.1 describes closures and set outs the problems associated with traditional implementations of closures. Section 6.2 then presents a sealed object-based solution and a new IL. Finally, Section 6.3 lists the changes in the compiler and evaluates the nanopass approach's impact on the codebase.

This chapter discusses the feature set and languages of the final version of Glyco, version 1.0.[1] A full language reference can be found in Appendix A.

6.1 Traditional Closures

A **closure** is a function that can use the environment[2] wherein the closure is defined during its operation, i.e., names (variables) valid at the point of the closure definition can also be used in the closure body. The closure is said to **capture** or **save** the environment or to **capture** variables from the **outer scope**.

The following Swift program uses a closure to print all numbers between 1 and 100, increased by `offset`.

```
1 let offset = 5
2 let numbers = (1...100)
3   .map { number in number + offset }
4 print(numbers)    // 6, 7, 8, 9, 10, ...
```

The closure takes one parameter `number` and captures the constant `offset` from the outer scope. The `map` method invokes the closure for every number between 1 and 100

[1] The source code is available at https://tsarouhas.eu/glyco/1.0/.

[2] An environment, and more specifically a *value environment*, can be seen as a mapping from names to values — each definition adds or updates entries in it and the environment is consulted whenever a name is mentioned in the program. That is of course not what happens in an ahead-of-time compiler such as Glyco where the environment consists of registers and memory locations, and the mapping is mostly implemented by the register allocator, but it remains a useful software model nevertheless.

inclusive, binding each number to the `number` parameter. The closure also gets access to `offset` even though `map` is not even aware of it.

Closures are usually implemented by pairing a function with an environment. When the pair is invoked, the function is invoked with the environment as a (hidden) parameter. For instance, the following OB program implements a closure `closure` that takes a single parameter `n` and captures an environment with a single captured name `offset`.

```
(
  let(
    (offset, 5)
    (closure, record(
      (function,
        λ(
          takes: (n, s32) (env, cap(record(((offset, s32))))),
          returns: s32,
          in: value(binary(n, add, field(offset, of: env)))
        )
      )
      (env, record((offset, offset)))
    )), in:
    evaluate(field(function, of: closure), 600 field(env, of: closure))
  )
)
```

The closure here is implemented as a capability to a record containing a code capability `function` and a capability `env` to a record containing fields for each captured value (here just `offset`). This closure capability (to the function–environment record) can be passed around the program and eventually invoked as shown above — `600` is in this example the argument to the closure.

The contents of a closure's saved environment are established at the point of closure definition. In most programmer models, the captured value `offset` from the example above should remain fixed at `5`, which is the value of `offset` at closure definition. A user of a closure should not be able to execute the closure with a different value for `offset` than the closure's creator intended. They should only be able to influence the closure's result by providing different arguments, and possibly by influencing any global state that the user has access to and the closure relies on.

In this OB example, it is almost trivially simple to change the value of the `offset` field, e.g., by writing `do(setField(offset, of: env, to: 10), then: evaluate(...))`.

6.2 Sealed Closures

The encapsulation problem presented in the previous section is solved in **sealed closures**, which are closures whose **captured environment cannot be accessed from outside the closure body**. Sealed closures are presented in a new IL **CL** (Closures), built on top of OB. CL does not provide any new syntax but instead increases the expressive power provided by λ values, which can now mention names defined outside of the lambda definition. The OB closure example above can now be expressed more elegantly as

```
(
  let(
    (offset, 5)
    (closure, λ(takes: (n, s32), returns: s32, in:
      value(binary(n, add, offset))
    )),
    in: evaluate(closure, 600)
  )
)
```

From CL to OB The CL to OB nanopass defines a unique object type for each closure definition and instantiates a single object of that new type. The closure object's state consists of one record entry per captured name. The object is defined with a single method `invoke` that binds the captured names to the corresponding values in `self` and then performs the closure body. The above CL program is thus lowered to the OB program

```
(
  let(
    (offset, 5)
    (
      closure,
      letType(
        object(
          (
            cl.Closure,
            initialiser: (takes: (offset, s32), in: record((offset, offset))),
            methods: (
              invoke,
              takes: (n, s32),
              returns: s32,
              in: let(
                (offset, field(offset, of: self)),
                in: value(binary(n, add, offset))
              )
            )
          )
        ),
        in: message(object(cl.Closure, offset), invoke)
      )
    ),
    in: evaluate(closure, 600)
  )
)
```

6.3 Evaluation: Impact on the Codebase

Similarly to what we did in Sections 4.4 and 5.4, we evaluate the impact of the implementation of sealed closures in Glyco by describing the new ILs and changes to existing ILs before quantifying the codebase's growth. The changes to the compiler pipeline are also summarised in Fig. 6.1.

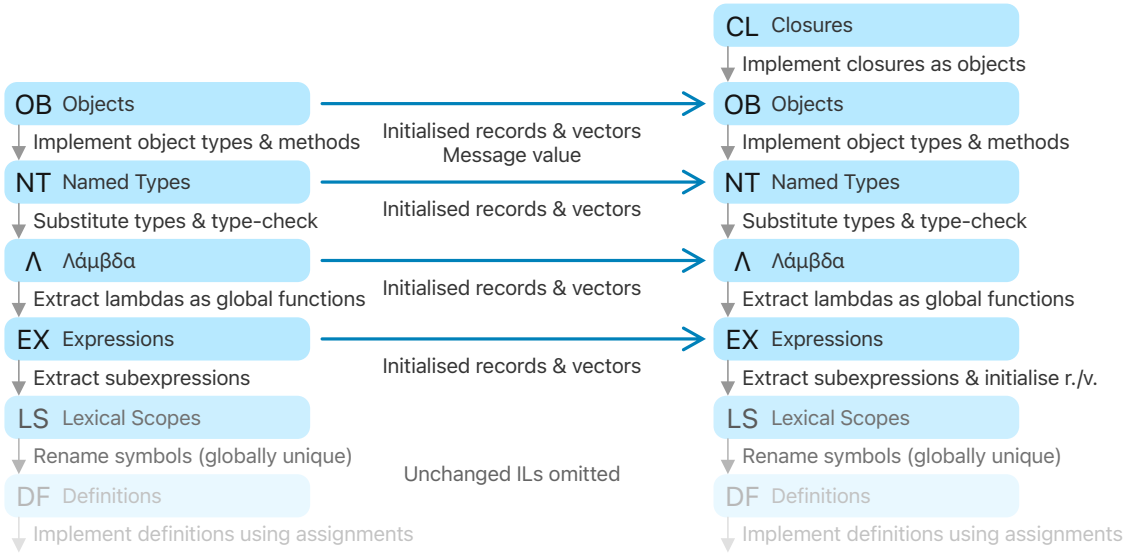


FIGURE 6.1: The Glyco 0.3 (left) and 1.0 (right) pipelines and the main differences between them.

Expressions (EX) through Objects (OB) The `record` value is redesigned to accept record entries instead of a record type. The EX to LS nanopass is updated to initialise the record besides only allocating it, thereby reducing the amount of code needed to create records in higher ILs like OB and CL.

The `vector` value is redesigned in a similar vein to accept a fill value. The EX to LS nanopass statically initialises the allocated vector, except when the fill value is `0` since all allocated buffers are already zero-initialised in the emulator, and should be in a more realistic CHERI-RISC-V runtime environment.

Objects (OB) The `message` value is repurposed to *create* a message, i.e., a bound object–method pair, as opposed to *sending* one. Messages are of a new `message` capability type and can be sent in an `evaluate` value, which continues to accept normal lambdas. They're represented in NT as a record consisting of an object and method capability, and provide type erasure as required from closures (which are implemented as a unique object type per closure definition).

To reduce the costs of allocating (and initialising) messages, the OB to NT nanopass recognises the form `evaluate(message(receiver, method), arguments)` and lowers it by immediately invoking the associated method.

Initialisers are redesigned to return an unsealed capability to a record, instead of initialising an already allocated record provided in `self`. The object type no longer declares a state record type, but is now instead inferred from the initialiser's result type. In combination with the `record` value redesign, this reduces the amount of code needed to define an object type.

Closures (CL) This new IL introduces no new syntax. The CL to OB nanopass determines captured names and creates unique object types for each closure. As an optimisation, the nanopass leaves lambdas which do not capture any names as-is.

Codebase growth The implementation of sealed closures introduces 1 new IL bringing the total in Glyco 1.0 to 23 ILs. Using cloc again, we measured a growth in the codebase from 9360 SLOC in Glyco 0.3 to 10357 SLOC in Glyco 1.0, representing a net growth of +11%. However, as noted before, a significant part of this growth is due to code duplication.

Chapter 7

Conclusion

The primary goal of this thesis is to explore a design and implementation of a compiler using a nanopass approach for experimenting with security features built on CHERI-RISC-V capabilities. A second goal is to evaluate the approach itself in terms of how well it lends itself to experimentation with (radically) new ideas. This second goal motivates the structure of the thesis text: Chapters 3 to 6 each present a milestone of the compiler.

The text begins in Chapter 2 with a brief introduction to capability machines and the CHERI-RISC-V architecture.

Chapter 3 then explores the basic version of a nanopass compiler, which we named *Glyco*. The design of this first version of the compiler is adapted from an existing x86 student compiler by Bowman et al. [2022]. The basic version of the compiler provides almost no features specifically enabled by capability support in hardware and is intended as a baseline for extensions. The compiler implements procedure calls using a calling convention named GCCC that looks like most traditional calling conventions used in the field.

The first upgrade of Glyco is the addition of a secure calling convention named *GHSCC*, a variant of a calling convention proposed by Georges et al. [2021a], which is discussed in Chapter 4. Unlike GCCC, GHSCC organises call frames in a linked list in heap-allocated memory. As part of this extension, we also improve the heap allocator by implementing it as a runtime routine and encapsulating the heap capability in it, ensuring that user programs cannot derive capabilities to arbitrary regions of the heap while still allowing them to get access to heap memory allocated by them. GHSCC provides local state encapsulation as described by Skorstengaard et al. [2019b, Section 1], a security property guaranteeing that the local state of a procedure call cannot be accessed from other procedures/calls. Additionally, GHSCC provides a security property we call *unrepeatable return* which guarantees that a return capability can be used at most once, a weaker variant of the well-bracketed control flow security property also described by Skorstengaard et al. [2019b, Section 1]. These properties continue to hold even if a Glyco-built program were to interoperate with untrusted code written in assembly.

The second addition to Glyco, discussed in Chapter 5, is a feature we call *sealed objects*, which are similar to objects in object-oriented languages but whose state is inaccessible outside methods defined during type definition. This form of encapsulation is enforced

using sealed capabilities, meaning that such code could securely interoperate with code written in assembly.

The third and final addition to Glyco is an application of sealed objects, *sealed closures*, which are closures whose captured environment cannot be accessed outside of the closure body, even if the closure is passed to untrusted code. We discuss these in Chapter 6.

7.1 Nanopass Approach

The nanopass approach allows a compiler designer to manage complexity by representing programs in several abstractions and performing actions at a suitable abstraction. For instance, the calling convention is imposed at an abstraction where structured values, abstract locations, and registers are available. Structured values are introduced in SV and are used for creating arguments records when not all arguments fit in registers. Abstract locations are introduced in ALA and are used for binding the contents of callee-saved register to names. Not all callee-saved registers need to be saved so the register allocator in a later nanopass cleans up (some) redundant moves. It does so purely using liveness information; it has no knowledge about the calling convention. As a compiler designer, we also want to limit complexity by removing low-level features such as direct register access in as many languages as we can. CC, i.e., the language introducing the calling convention, is therefore introduced right above ALA and SV. It is also the lowest language without syntax for accessing registers. A single intermediate language such as LLVM IR brings all abstractions together and compiler passes are loosely sequenced, as opposed to the rigidly ordered nanopasses in Glyco and the student compiler mentioned above.

Working with tens of ILs is nevertheless also a challenge, especially since it can cause significant code duplication. An IL usually differs little from its lower language, so nanopass compilers commonly use a framework (such as one described by Sarkar et al. [2004] for educational compilers). We chose to write the compiler in Swift, a strongly typed programming language with data-flow analysis, exhaustive pattern match checking, result builders, and custom operators. The first two features as well as strong typing ensure that each syntactical element in an IL is properly handled while result builders and custom operators make it possible to define a domain-specific language within Swift, which in turn allows us to write nanopasses in a more natural way. However, we did not find a nanopass framework written in Swift, which means that our codebase contains a lot of duplication. While we do use code generation tools such as Sourcery,[1] nanopasses still need to be written exhaustively. The Swift compiler's aforementioned static analysis features reduce the development time but this isn't reflected in the SLOC metric of our evaluation. A nanopass framework allows one to omit unchanged syntactical elements and trivial nanopasses and would make SLOC metrics more meaningful.

7.2 Future Directions

More calling conventions Glyco currently provides two calling conventions, one of which is "secure." An obvious future direction is to add more secure calling conventions,

[1] Sourcery is available at `https://github.com/krzysztofzablocki/Sourcery`.

especially ones proposed in the literature which provide full well-bracketed control flow.

Continuations Heap-allocated call frames (GHSCC), return capabilities, and closures can be naturally generalised to first-class support for continuations.

One possible approach is to introduce continuations at a relatively low level in CC (and ILs near it) by introducing continuation values, which are similar to procedure values, except that they do not return. Just like procedures, continuations accept arguments and can be invoked using `call` effects. Every procedure has an implicit return continuation; the `return` effect is repurposed to invoke this implicit return continuation. The `call` effect is also extended with support to perform a call-with-current-continuation (known as `call/cc` in Scheme): any omitted arguments to a parameter of continuation type are filled in with a continuation following the call.

Another approach is to build continuations at the very top of the current compiler pipeline by relying on existing object and closure infrastructure, in a new IL CO (and probably a few more ILs under it). CO introduces `evaluateWithCurrentContinuation` values; one or more nanopasses extract continuations from these values as lambdas. Continuations can be executed using a `continue` tail effect. Continuations can accept arguments and do not return.

More generic programming The current implementation of Glyco uses some generic programming to reduce code duplication (such as the `Name`, `Codable`, `Language`, and `ComposableEffect` protocols) but a future direction could be to use it much more extensively.

One concrete example would be to define a range of protocols specialising `Language`, with each subprotocol declaring some common syntactical elements provided by two or more ILs. All ILs above CC could for instance conform to a `ValueLanguage` protocol which would declare a `Value` associated type. Syntactical element types would accept a type parameter constrained to one or more language protocols. `Result` could for instance accept a type conforming to `ValueLanguage`. `ValueLanguage` could then provide a `Result` type-alias that is defined as `Result<Self>`. There would no longer be a need to duplicate `Result` since occurrences of `Value` would automatically be bound to the right concrete `Value` type for that language. The same would be done for the majority of syntactical elements appearing in more than one IL.

Basic blocks The student compiler by Bowman et al. [2022] has a few ILs relating to basic blocks which Glyco adapts. However, neither compiler defines optimisations such as common subexpression elimination that are commonly done on basic blocks. As a possible improvement, we could either implement a few of such optimisations, or simplify the codebase by omitting the basic block ILs (BB and PR) altogether.

Appendix A

Language Reference

This chapter presents all ILs in the final version of Glyco, from high-level to low-level languages. The Swift source files for these ILs can be found under `https://tsarouhas.eu/glyco/ils/`.

Common grammar The following grammar applies to all ILs. For any *N*:

[N] := ϵ | *N [N]*

Bool := **false** | **"false"** | **true** | **"true"**

Int := *digits* | **-** *digits*

digits := *digit* | *digit digits*

digit := **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

String := *id* | zero or more printable characters enclosed in double-quotes, with every double-quote character in the string content doubled

id := *idstart* | *idstart idtail*

idstart := a letter from Unicode General Category L* or M* | _ | **$** | **%** | **.**

idtail := *idchar* | *idchar idtail*

idchar := an alphanumeric character from Unicode General Category L*, M*, or N* | _ | **$** | **%** | **.**

CL (Closures) A language that introduces closures, i.e., anonymous functions with an environment.

CL.Program ::= (*Result*)

CL.RecordType ::= (*[Field]*)

CL.Field ::= (*Name*, *ValueType*)

CL.ObjectType ::= (*TypeName*, **initialiser:** *Initialiser*, **methods:** *[Method]*)

CL.Effect ::= **do(***[Effect]***)** | **let(***[Definition]*, **in:** *Effect***)** | **setField(***Field.Name*, **of:** *Value*, **to:** *Value***)** | **setElement(of:** *Value*, **at:** *Value*, **to:** *Value***)**

CL.Definition ::= **(***Symbol*, *Value***)**

CL.Initialiser ::= **(takes:** *[Parameter]*, **in:** *Value***)**

CL.Result ::= **value(***Value***)** | **evaluate(***Value*, *[Value]***)** | **if(***Predicate*, **then:** *Result*, **else:** *Result***)** | **let(***[Definition]*, **in:** *Result***)** | **do(***[Effect]*, **then:** *Result***)**

CL.RecordEntry ::= **(***Field.Name*, *Value***)**

CL.Predicate ::= *Bool* | **constant(***Bool***)** | **relation(***Value*, *BranchRelation*, *Value***)** | **if(***Predicate*, **then:** *Predicate*, **else:** *Predicate***)** | **let(***[Definition]*, **in:** *Predicate***)**

CL.Parameter ::= **(***Symbol*, *ValueType***)**

CL.Method ::= **(***Symbol*, **takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result***)**

CL.CapabilityType ::= **vector(of:** *ValueType***)** | **record(***RecordType***)** | **object(***TypeName***)** | **function(takes:** *[Parameter]*, **returns:** *ValueType*, **closure:** *Bool***)** | **message(takes:** *[Parameter]*, **returns:** *ValueType***)** | **seal**

CL.TypeDefinition ::= **alias(***TypeName*, *ValueType***)** | **nominal(***TypeName*, *ValueType***)** | **object(***ObjectType***)**

CL.Value ::= *Int* | *Symbol* | **self** | **constant(***Int***)** | **named(***Symbol***)** | **record(***[RecordEntry]***)** | **field(***Field.Name*, **of:** *Value***)** | **vector(***Value*, **count:** *Int***)** | **element(of:** *Value*, **at:** *Value***)** | λ**(takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result***)** | **object(***TypeName*, *[Value]***)** | **binary(***Value*, *BinaryOperator*, *Value***)** | **evaluate(***Value*, *[Value]***)** | **message(***Value*, *Method.Name***)** | **if(***Predicate*, **then:** *Value*, **else:** *Value***)** | **let(***[Definition]*, **in:** *Value***)** | **letType(***[TypeDefinition]*, **in:** *Value***)** | **do(***[Effect]*, **then:** *Value***)**

CL.ValueType ::= *TypeName* | **named(***TypeName***)** | **u8** | **s32** | **cap(***CapabilityType***)**

OB (Objects) A language that introduces objects, i.e., encapsulated values with methods.

OB.Program ::= **(***Result***)**

OB.Predicate ::= *Bool* | **constant(***Bool***)** | **relation(***Value*, *BranchRelation*, *Value***)** | **if(***Predicate*, **then:** *Predicate*, **else:** *Predicate***)** | **let(***[Definition]*, **in:** *Predicate***)**

OB.Parameter ::= **(***Symbol*, *ValueType***)**

OB.Initialiser ::= **(takes:** *[Parameter]*, **in:** *Value***)**

OB.RecordType ::= **(***[Field]***)**

OB.Field ::= **(***Name*, *ValueType***)**

OB.Effect ::= **do(***[Effect]***)** | **let(***[Definition]*, **in:** *Effect***)** | **setField(***Field.Name*, **of:** *Value*, **to:** *Value***)** | **setElement(of:** *Value*, **at:** *Value*, **to:** *Value***)**

OB.Value ::= *Int* | *Symbol* | **self** | **constant(***Int***)** | **named(***Symbol***)** | **record(***[RecordEntry]***)** | **field(***Field.Name*, **of:** *Value***)** | **vector(***Value*, **count:** *Int***)** | **element(of:** *Value*, **at:** *Value***)** | λ**(takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result***)** | **object(***TypeName*, *[Value]***)** | **binary(***Value*, *BinaryOperator*, *Value***)** | **evaluate(***Value*, *[Value]***)** | **message(***Value*, *Method.Name***)** | **if(***Predicate*, **then:** *Value*, **else:** *Value***)** | **let(***[Definition]*, **in:** *Value***)** | **letType(***[TypeDefinition]*, **in:** *Value***)** | **do(***[Effect]*, **then:** *Value***)**

OB.ObjectType ::= (*TypeName*, **initialiser:** *Initialiser*, **methods:** *[Method]*)

OB.Result ::= **value**(*Value*) | **evaluate**(*Value*, *[Value]*) | **if**(*Predicate*, **then:** *Result*, **else:** *Result*) | **let**(*[Definition]*, **in:** *Result*) | **do**(*[Effect]*, **then:** *Result*)

OB.ValueType ::= *TypeName* | **named**(*TypeName*) | **u8** | **s32** | **cap**(*CapabilityType*)

OB.CapabilityType ::= **vector(of:** *ValueType*) | **record**(*RecordType*) | **object**(*TypeName*) | **function(takes:** *[Parameter]*, **returns:** *ValueType*) | **message(takes:** *[Parameter]*, **returns:** *ValueType*) | **seal**

OB.Method ::= (*Symbol*, **takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result*)

OB.RecordEntry ::= (*Field.Name*, *Value*)

OB.Definition ::= (*Symbol*, *Value*)

OB.TypeDefinition ::= **alias**(*TypeName*, *ValueType*) | **nominal**(*TypeName*, *ValueType*) | **object**(*ObjectType*)

NT (Named Types) A language that introduces named alias and nominal types.

NT.Program ::= (*Result*)

NT.Effect ::= **do**(*[Effect]*) | **let**(*[Definition]*, **in:** *Effect*) | **setField**(*Field.Name*, **of:** *Value*, **to:** *Value*) | **setElement(of:** *Value*, **at:** *Value*, **to:** *Value*)

NT.RecordEntry ::= (*Field.Name*, *Value*)

NT.CapabilityType ::= **vector(of:** *ValueType*, **sealed:** *Bool*) | **record**(*RecordType*, **sealed:** *Bool*) | **function(takes:** *[Parameter]*, **returns:** *ValueType*) | **seal(sealed:** *Bool*)

NT.Result ::= **value**(*Value*) | **evaluate**(*Value*, *[Value]*) | **if**(*Predicate*, **then:** *Result*, **else:** *Result*) | **let**(*[Definition]*, **in:** *Result*) | **do**(*[Effect]*, **then:** *Result*)

NT.TypeDefinition ::= **alias**(*TypeName*, *ValueType*) | **nominal**(*TypeName*, *ValueType*)

NT.Value ::= *Int* | *Symbol* | **constant**(*Int*) | **named**(*Symbol*) | **record**(*[RecordEntry]*) | **field**(*Field.Name*, **of:** *Value*) | **vector**(*Value*, **count:** *Int*) | **element(of:** *Value*, **at:** *Value*) | λ**(takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result*) | **seal** | **sealed**(*Value*, **with:** *Value*) | **binary**(*Value*, *BinaryOperator*, *Value*) | **evaluate**(*Value*, *[Value]*) | **cast**(*Value*, **as:** *ValueType*) | **if**(*Predicate*, **then:** *Value*, **else:** *Value*) | **let**(*[Definition]*, **in:** *Value*) | **letType**(*[TypeDefinition]*, **in:** *Value*) | **do**(*[Effect]*, **then:** *Value*)

NT.Definition ::= (*Symbol*, *Value*)

NT.RecordType ::= (*[Field]*)

NT.Field ::= (*Name*, *ValueType*)

NT.TypeName ::= *String*

NT.ValueType ::= *TypeName* | **named**(*TypeName*) | **u8** | **s32** | **cap**(*CapabilityType*)

NT.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Value*, *BranchRelation*, *Value*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **let**(*[Definition]*, **in:** *Predicate*)

NT.Parameter ::= (*Symbol*, *ValueType*, **sealed:** *Bool*)

Λ (Lambdas) A language that moves functions to value position.

Λ.Program ::= (*Result*)

Λ.Value ::= *Int* | *Symbol* | **constant**(*Int*) | **named**(*Symbol*) | **record**(*[RecordEntry]*) | **field**(*Field.Name*, **of:** *Value*) | **vector**(*Value*, **count:** *Int*) | **element**(**of:** *Value*, **at:** *Value*) | λ(**takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result*) | **seal** | **sealed**(*Value*, **with:** *Value*) | **binary**(*Value*, *BinaryOperator*, *Value*) | **evaluate**(*Value*, *[Value]*) | **if**(*Predicate*, **then:** *Value*, **else:** *Value*) | **let**(*[Definition]*, **in:** *Value*) | **do**(*[Effect]*, **then:** *Value*)

Λ.RecordEntry ::= (*Field.Name*, *Value*)

Λ.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Value*, *BranchRelation*, *Value*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **let**(*[Definition]*, **in:** *Predicate*)

Λ.Effect ::= **do**(*[Effect]*) | **let**(*[Definition]*, **in:** *Effect*) | **setField**(*Field.Name*, **of:** *Value*, **to:** *Value*) | **setElement**(**of:** *Value*, **at:** *Value*, **to:** *Value*)

Λ.Result ::= **value**(*Value*) | **evaluate**(*Value*, *[Value]*) | **if**(*Predicate*, **then:** *Result*, **else:** *Result*) | **let**(*[Definition]*, **in:** *Result*) | **do**(*[Effect]*, **then:** *Result*)

Λ.Definition ::= (*Symbol*, *Value*)

EX (Expressions) A language that introduces expression semantics for values, thereby abstracting over computation effects.

EX.Program ::= (*Result*, **functions:** *[Function]*)

EX.Value ::= *Int* | *Symbol* | **constant**(*Int*) | **named**(*Symbol*) | **record**(*[RecordEntry]*) | **field**(*Field.Name*, **of:** *Value*) | **vector**(*Value*, **count:** *Int*) | **element**(**of:** *Value*, **at:** *Value*) | **function**(*Label*) | **seal** | **sealed**(*Value*, **with:** *Value*) | **binary**(*Value*, *BinaryOperator*, *Value*) | **evaluate**(*Value*, *[Value]*) | **if**(*Predicate*, **then:** *Value*, **else:** *Value*) | **let**(*[Definition]*, **in:** *Value*) | **do**(*[Effect]*, **then:** *Value*)

EX.Definition ::= (*Symbol*, *Value*)

EX.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Value*, *BranchRelation*, *Value*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **let**(*[Definition]*, **in:** *Predicate*)

EX.Function ::= (*Label*, **takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result*)

EX.RecordEntry ::= (*Field.Name*, *Value*)

EX.Effect ::= **do**(*[Effect]*) | **let**(*[Definition]*, **in:** *Effect*) | **setField**(*Field.Name*, **of:** *Value*, **to:** *Value*) | **setElement**(**of:** *Value*, **at:** *Value*, **to:** *Value*)

EX.Result ::= **value**(*Value*) | **evaluate**(*Value*, *[Value]*) | **if**(*Predicate*, **then:** *Result*, **else:** *Result*) | **let**(*[Definition]*, **in:** *Result*) | **do**(*[Effect]*, **then:** *Result*)

LS (Lexical Scopes) A language that introduces lexical scopes of definitions, thereby removing name clashes.

LS.Program ::= (*Result*, **functions:** *[Function]*)

LS.Symbol ::= *String*

LS.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Source*, *BranchRelation*, *Source*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **let**(*[Definition]*, **in:** *Predicate*)

LS.Parameter ::= (*Symbol*, *ValueType*, **sealed:** *Bool*)

LS.ValueType ::= **u8** | **s32** | **cap**(*CapabilityType*)

LS.Value ::= **source**(*Source*) | **binary**(*Source*, *BinaryOperator*, *Source*) | **record**(*RecordType*) | **field**(*Field.Name*, **of:** *Symbol*) | **vector**(*ValueType*, **count:** *Int*) | **element**(**of:** *Symbol*, **at:** *Source*) | **seal** | **sealed**(*Symbol*, **with:** *Symbol*) | **evaluate**(*Source*, *[Source]*) | **if**(*Predicate*, **then:** *Value*, **else:** *Value*) | **let**(*[Definition]*, **in:** *Value*) | **do**(*[Effect]*, **then:** *Value*)

LS.Function ::= (*Label*, **takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result*)

LS.CapabilityType ::= **vector**(**of:** *ValueType*, **sealed:** *Bool*) | **record**(*RecordType*, **sealed:** *Bool*) | **function**(**takes:** *[Parameter]*, **returns:** *ValueType*) | **seal**(**sealed:** *Bool*)

LS.Source ::= *Int* | *Symbol* | **constant**(*Int*) | **named**(*Symbol*) | **function**(*Label*)

LS.Definition ::= (*Symbol*, *Value*)

LS.Effect ::= **do**(*[Effect]*) | **let**(*[Definition]*, **in:** *Effect*) | **setField**(*Field.Name*, **of:** *Symbol*, **to:** *Source*) | **setElement**(**of:** *Symbol*, **at:** *Source*, **to:** *Source*)

LS.Result ::= **value**(*Value*) | **evaluate**(*Source*, *[Source]*) | **if**(*Predicate*, **then:** *Result*, **else:** *Result*) | **let**(*[Definition]*, **in:** *Result*) | **do**(*[Effect]*, **then:** *Result*)

LS.RecordType ::= (*[Field]*)

LS.Field ::= (*Name*, *ValueType*)

DF (Definitions) A language that introduces definitions with function-wide namespacing.

DF.Program ::= (*Result*, **functions:** *[Function]*)

DF.Effect ::= **do**(*[Effect]*) | **let**(*[Definition]*, **in:** *Effect*) | **setField**(*Field.Name*, **of:** *Location*, **to:** *Source*) | **setElement**(**of:** *Location*, **at:** *Source*, **to:** *Source*)

DF.Result ::= **value**(*Value*) | **if**(*Predicate*, **then:** *Result*, **else:** *Result*) | **evaluate**(*Source*, *[Source]*) | **let**(*[Definition]*, **in:** *Result*) | **do**(*[Effect]*, **then:** *Result*)

DF.Definition ::= (*Location*, *Value*)

DF.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Source*, *BranchRelation*, *Source*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **let**(*[Definition]*, **in:** *Predicate*)

DF.Value ::= **source**(*Source*) | **binary**(*Source*, *BinaryOperator*, *Source*) | **record**(*RecordType*) | **field**(*Field.Name*, **of:** *Location*) | **vector**(*ValueType*, **count:** *Int*) | **element**(**of:** *Location*, **at:** *Source*) | **seal** | **sealed**(*Location*, **with:** *Location*) | **evaluate**(*Source*, *[Source]*) | **if**(*Predicate*, **then:** *Value*, **else:** *Value*) | **let**(*[Definition]*, **in:** *Value*) | **do**(*[Effect]*, **then:** *Value*)

DF.Function ::= (*Label*, **takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Result*)

CV (Computed Values) A language that allows a computation to be attached to a value.

CV.Program ::= (*Effect*, **procedures:** *[Procedure]*)

CV.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Source*, *BranchRelation*, *Source*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **do**(*[Effect]*, **then:** *Predicate*)

CV.Effect ::= **do**(*[Effect]*) | **set**(*Location*, **to:** *Value*) | **setField**(*Field.Name*, **of:** *Location*, **to:** *Source*) | **setElement**(**of:** *Location*, **at:** *Source*, **to:** *Source*) | **if**(*Predicate*, **then:** *Effect*, **else:** *Effect*) | **return**(*Source*)

CV.Procedure ::= (*Label*, **takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Effect*)

CV.Value ::= **source**(*Source*) | **binary**(*Source*, *BinaryOperator*, *Source*) | **record**(*RecordType*) | **field**(*Field.Name*, **of:** *Location*) | **vector**(*ValueType*, **count:** *Int*) | **element**(**of:** *Location*, **at:** *Source*) | **seal** | **sealed**(*Location*, **with:** *Location*) | **evaluate**(*Source*, *[Source]*) | **if**(*Predicate*, **then:** *Value*, **else:** *Value*) | **do**(*[Effect]*, **then:** *Value*)

CA (Canonical Assignments) A language that groups all effects that write to a location under one canonical assignment effect.

CA.Program ::= (*Effect*, **procedures:** *[Procedure]*)

CA.Procedure ::= (*Label*, **takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Effect*)

CA.Effect ::= **do**(*[Effect]*) | **set**(*Location*, **to:** *Value*) | **setField**(*Field.Name*, **of:** *Location*, **to:** *Source*) | **setElement**(**of:** *Location*, **at:** *Source*, **to:** *Source*) | **call**(*Source*, *[Source]*, **result:** *Location*) | **if**(*Predicate*, **then:** *Effect*, **else:** *Effect*) | **return**(*Source*)

CA.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Source*, *BranchRelation*, *Source*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **do**(*[Effect]*, **then:** *Predicate*)

CA.Value ::= **source**(*Source*) | **binary**(*Source*, *BinaryOperator*, *Source*) | **record**(*RecordType*) | **field**(*Field.Name*, **of:** *Location*) | **vector**(*ValueType*, **count:** *Int*) | **element**(**of:** *Location*, **at:** *Source*) | **seal** | **sealed**(*Location*, **with:** *Location*)

CC (Calling Convention) A language that introduces parameters & result values in procedures via the low-level Glyco calling convention.

CC.Program ::= (*Effect*, **procedures:** *[Procedure]*)

CC.CapabilityType ::= **vector**(**of:** *ValueType*, **sealed:** *Bool*) | **record**(*RecordType*, **sealed:** *Bool*) | **procedure**(**takes:** *[Parameter]*, **returns:** *ValueType*) | **seal**(**sealed:** *Bool*)

CC.RecordType ::= (*[Field]*)

CC.Field ::= (*Name*, *ValueType*)

CC.Parameter ::= (*Location*, *ValueType*, **sealed:** *Bool*)

CC.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Source*, *BranchRelation*, *Source*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **do**(*[Effect]*, **then:** *Predicate*)

CC.Source ::= *Int* | *Location* | **constant**(*Int*) | **location**(*Location*) | **procedure**(*Label*)

CC.Effect ::= **do**(*[Effect]*) | **set**(*Location*, **to:** *Source*) | **compute**(*Location*, *Source*, *BinaryOperator*, *Source*) | **createRecord**(*RecordType*, **capability:** *Location*, **scoped:** *Bool*) | **getField**(*Field.Name*, **of:** *Location*, **to:** *Location*) | **setField**(*Field.Name*, **of:** *Location*, **to:** *Source*) | **createVector**(*ValueType*, **count:** *Int*, **capability:** *Location*, **scoped:** *Bool*) | **getElement**(**of:** *Location*, **index:** *Source*, **to:** *Location*) | **setElement**(**of:** *Location*, **index:** *Source*, **to:** *Source*) | **createSeal**(**in:** *Location*) | **seal**(**into:** *Location*, **source:** *Location*, **seal:** *Location*) | **destroyScopedValue**(**capability:** *Source*) | **if**(*Predicate*, **then:** *Effect*, **else:** *Effect*) | **call**(*Source*, *[Source]*, **result:** *Location*) | **return**(*Source*)

CC.ValueType ::= **u8** | **s32** | **cap**(*CapabilityType*)

CC.Procedure ::= (*Label*, **takes:** *[Parameter]*, **returns:** *ValueType*, **in:** *Effect*)

SV (Structured Values) A language that introduces structured values, i.e., vectors and records.

SV.Program ::= (*Effect*, **procedures:** *[Procedure]*)

SV.ValueType ::= **u8** | **s32** | **cap**(*CapabilityType*) | **registerDatum**

SV.Source ::= *Int* | *AbstractLocation* | **constant**(*Int*) | **abstract**(*AbstractLocation*) | **register**(*Register*, *ValueType*) | **frame**(*Frame.Location*) | **capability**(**to:** *Label*)

SV.Effect ::= **do**(*[Effect]*) | **set**(*Location*, **to:** *Source*) | **compute**(*Location*, *Source*, *BinaryOperator*, *Source*) | **createRecord**(*RecordType*, **capability:** *Location*, **scoped:** *Bool*) | **getField**(*Field.Name*, **of:** *Location*, **to:** *Location*) | **setField**(*Field.Name*, **of:** *Location*, **to:** *Source*) | **createVector**(*ValueType*, **count:** *Int*, **capability:** *Location*, **scoped:** *Bool*) | **getElement**(**of:** *Location*, **index:** *Source*, **to:** *Location*) | **setElement**(**of:** *Location*, **index:** *Source*, **to:** *Source*) | **destroyScopedValue**(**capability:** *Source*) | **createSeal**(**in:** *Location*) | **seal**(**into:** *Location*, **source:** *Location*, **seal:** *Location*) | **if**(*Predicate*, **then:** *Effect*, **else:** *Effect*) | **pushScope** | **popScope** | **clearAll**(**except:** *[Register]*) | **call**(*Source*, **parameters:** *[Register]*) | **callSealed**(*Source*, **data:** *Source*, **unsealedParameters:** *[Register]*) | **return**(**to:** *Source*)

SV.CapabilityType ::= **vector**(**of:** *ValueType*, **sealed:** *Bool*) | **record**(*RecordType*, **sealed:** *Bool*) | **code** | **seal**(**sealed:** *Bool*)

SV.RecordType ::= (*[Field]*)

SV.Field ::= (*Name*, *ValueType*)

SV.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Source*, *BranchRelation*, *Source*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **do**(*[Effect]*, **then:** *Predicate*)

SV.Procedure ::= (*Label*, **in:** *Effect*)

ID (Inferred Declarations) A language that infers declarations from definitions.

ID.Program ::= (*Effect*, **procedures:** *[Procedure]*)

ID.Procedure ::= (*Label*, **in:** *Effect*)

ID.Effect ::= **do**(*[Effect]*) | **set**(*Location*, **to:** *Source*) | **compute**(*Location*, *Source*, *BinaryOperator*, *Source*) | **createBuffer**(**bytes:** *Int*, **capability:** *Location*, **scoped:** *Bool*) | **destroyBuffer**(**capability:** *Source*) | **getElement**(*DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Source*) | **setElement**(*DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Source*) | **createSeal**(**in:** *Location*) | **seal**(**into:** *Location*, **source:** *Location*, **seal:** *Location*) | **if**(*Predicate*, **then:** *Effect*, **else:** *Effect*) | **pushScope** | **popScope** | **clearAll**(**except:** *[Register]*) | **call**(*Source*, **parameters:** *[Register]*) | **callSealed**(*Source*, **data:** *Source*, **unsealedParameters:** *[Register]*) | **return**(**to:** *Source*)

ID.Predicate ::= *Bool* | **constant(***Bool***)** | **relation(***Source*, *BranchRelation*, *Source***)** | **if(***Predicate*, **then:** *Predicate*, **else:** *Predicate***)** | **do(***[Effect]*, **then:** *Predicate***)**

AL (Abstract Locations) A language that introduces abstract locations, i.e., locations whose physical locations are not specified by the programmer.

AL.Program ::= **(locals:** *Declarations*, **in:** *Effect*, **procedures:** *[Procedure]***)**

AL.Predicate ::= *Bool* | **constant(***Bool***)** | **relation(***Source*, *BranchRelation*, *Source***)** | **if(***Predicate*, **then:** *Predicate*, **else:** *Predicate***)** | **do(***[Effect]*, **then:** *Predicate***)**

AL.Procedure ::= **(***Label*, **locals:** *Declarations*, **in:** *Effect***)**

AL.Effect ::= **do(***[Effect]***)** | **set(***Location*, **to:** *Source***)** | **compute(***Location*, *Source*, *BinaryOperator*, *Source***)** | **createBuffer(bytes:** *Int*, **capability:** *Location*, **scoped:** *Bool***)** | **destroyBuffer(capability:** *Source***)** | **getElement(***DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Location***)** | **setElement(***DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Source***)** | **createSeal(in:** *Location***)** | **seal(into:** *Location*, **source:** *Location*, **seal:** *Location***)** | **if(***Predicate*, **then:** *Effect*, **else:** *Effect***)** | **pushScope** | **popScope** | **clearAll(except:** *[Register]***)** | **call(***Source*, **parameters:** *[Register]***)** | **callSealed(***Source*, **data:** *Source*, **unsealedParameters:** *[Register]***)** | **return(to:** *Source***)**

ALA (Abstract Locations, Analysed) A language that introduces abstract locations, annotated with liveness and conflict information.

ALA.Program ::= **(locals:** *Declarations*, **in:** *Effect*, **procedures:** *[Procedure]***)**

ALA.ConflictGraph ::= **(***[Conflict]***)**

ALA.Conflict ::= **(***Location*, *Location***)**

ALA.Declarations ::= **(***[Declaration]***)**

ALA.AbstractLocation ::= *String*

ALA.Predicate ::= **constant(***Bool*, **analysisAtEntry:** *Analysis***)** | **relation(***Source*, *BranchRelation*, *Source*, **analysisAtEntry:** *Analysis***)** | **if(***Predicate*, **then:** *Predicate*, **else:** *Predicate*, **analysisAtEntry:** *Analysis***)** | **do(***[Effect]*, **then:** *Predicate*, **analysisAtEntry:** *Analysis***)**

ALA.Source ::= *Int* | *AbstractLocation* | **constant(***Int***)** | **abstract(***AbstractLocation***)** | **register(***Register*, *DataType***)** | **frame(***Frame.Location***)** | **capability(to:** *Label***)**

ALA.Declaration ::= **abstract(***AbstractLocation*, *DataType***)** | **frame(***Frame.Location*, *DataType***)**

ALA.Effect ::= **do(***[Effect]*, **analysisAtEntry:** *Analysis***)** | **set(***Location*, **to:** *Source*, **analysisAtEntry:** *Analysis***)** | **compute(***Location*, *Source*, *BinaryOperator*, *Source*, **analysisAtEntry:** *Analysis***)** | **createBuffer(bytes:** *Int*, **capability:** *Location*, **scoped:** *Bool*, **analysisAtEntry:** *Analysis***)** | **destroyBuffer(capability:** *Source*, **analysisAtEntry:** *Analysis***)** | **getElement(***DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Location*, **analysisAtEntry:** *Analysis***)** | **setElement(***DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Source*, **analysisAtEntry:** *Analysis***)** | **createSeal(in:** *Location*, **analysisAtEntry:** *Analysis***)** | **seal(into:** *Location*, **source:** *Location*, **seal:** *Location*, **analysisAtEntry:** *Analysis***)** | **if(***Predicate*, **then:** *Effect*, **else:** *Effect*, **analysisAtEntry:** *Analysis***)** | **pushScope(analysisAtEntry:** *Analysis***)** | **popScope(analysisAtEntry:** *Analysis***)** | **clearAll(except:** *[Register]*, **analysisAtEntry:** *Analysis***)** | **call(***Source*, **parameters:** *[Register]*, **analysisAtEntry:** *Analysis***)** | **callSealed(***Source*, **data:** *Source*, **unsealedParameters:** *[Register]*, **analysisAtEntry:** *Analysis***)** | **return(to:** *Source*, **analysisAtEntry:** *Analysis***)**

ALA.Location ::= **abstract(***AbstractLocation***)** | **register(***Register***)** | **frame(***Frame.Location***)**

ALA.Procedure ::= (*Label*, **locals:** *Declarations*, **in:** *Effect*)

ALA.Analysis ::= (**conflicts:** *ConflictGraph*, **possiblyLiveLocations:** *Set<Location>*)

CD (Conditionals) A language that introduces conditionals in effects and predicates, thereby abstracting over blocks (and jumps).

CD.Program ::= (*Effect*, **procedures:** *[Procedure]*)

CD.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Source*, *BranchRelation*, *Source*) | **if**(*Predicate*, **then:** *Predicate*, **else:** *Predicate*) | **do**(*[Effect]*, **then:** *Predicate*)

CD.Effect ::= **do**(*[Effect]*) | **set**(*DataType*, *Location*, **to:** *Source*) | **compute**(*Location*, *Source*, *BinaryOperator*, *Source*) | **createBuffer**(**bytes:** *Int*, **capability:** *Location*, **onFrame:** *Bool*) | **destroyBuffer**(**capability:** *Source*) | **getElement**(*DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Location*) | **setElement**(*DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Source*) | **createSeal**(**in:** *Location*) | **seal**(**into:** *Location*, **source:** *Location*, **seal:** *Location*) | **if**(*Predicate*, **then:** *Effect*, **else:** *Effect*) | **pushFrame**(*Frame*) | **popFrame** | **clearAll**(**except:** *[Register]*) | **call**(*Source*) | **callSealed**(*Source*, **data:** *Source*) | **return**(**to:** *Source*)

CD.Procedure ::= (*Label*, **in:** *Effect*)

PR (Predicates) A language that introduces predicates in branches.

PR.Program ::= (*[Block]*)

PR.Block ::= (**name:** *Label*, **do:** *[Effect]*, **then:** *Continuation*)

PR.Continuation ::= **continue**(**to:** *Label*) | **branch**(**if:** *Predicate*, **then:** *Label*, **else:** *Label*) | **call**(*Source*, **returnPoint:** *Label*) | **callSealed**(*Source*, **data:** *Source*, **returnPoint:** *Label*) | **return**(**to:** *Source*)

PR.Predicate ::= *Bool* | **constant**(*Bool*) | **relation**(*Source*, *BranchRelation*, *Source*)

BB (Basic Blocks) A language that groups effects into blocks of effects where blocks can only be entered at a single entry point and exited at a single exit point.

BB.Program ::= (*[BB.Block]*)

BB.Effect ::= **set**(*DataType*, *Location*, **to:** *Source*) | **compute**(*Location*, *Source*, *BinaryOperator*, *Source*) | **createBuffer**(**bytes:** *Int*, **capability:** *Location*, **onFrame:** *Bool*) | **destroyBuffer**(**capability:** *Source*) | **getElement**(*DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Location*) | **setElement**(*DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Source*) | **createSeal**(**in:** *Location*) | **seal**(**into:** *Location*, **source:** *Location*, **seal:** *Location*) | **pushFrame**(*Frame*) | **popFrame** | **clearAll**(**except:** *[Register]*)

BB.Continuation ::= **continue**(**to:** *Label*) | **branch**(*Source*, *BranchRelation*, *Source*, **then:** *Label*, **else:** *Label*) | **call**(*Source*, **returnPoint:** *Label*) | **callSealed**(*Source*, **data:** *Source*, **returnPoint:** *Label*) | **return**(**to:** *Source*)

BB.Block ::= (**name:** *Label*, **do:** *[Effect]*, **then:** *Continuation*)

FO (Flexible Operands) A language that introduces flexible operands in instructions, i.e., instructions that can take frame locations in all operand positions.

FO.Program ::= (*[Effect]*)

FO.Source ::= *Int* | **constant**(*Int*) | **register**(*Register*) | **frame**(*Frame.Location*) | **capability**(**to:** *Label*)

FO.Effect ::= **set**(*DataType*, *Location*, **to:** *Source*) | **compute**(*Location*, *Source*, *BinaryOperator*, *Source*) | **createBuffer**(**bytes:** *Int*, **capability:** *Location*, **onFrame:** *Bool*) | **destroyBuffer**(**capability:** *Source*) | **getElement**(*DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Location*) | **setElement**(*DataType*, **of:** *Location*, **offset:** *Source*, **to:** *Source*) | **createSeal**(**in:** *Location*) | **seal**(**into:** *Location*, **source:** *Location*, **seal:** *Location*) | **pushFrame**(*Frame*) | **popFrame** | **clearAll**(**except:** *[Register]*) | **branch**(**to:** *Label*, *Source*, *BranchRelation*, *Source*) | **jump**(**to:** *Label*) | **call**(*Source*) | **callSealed**(*Source*, **data:** *Source*, **returnPoint:** *Label*) | **return**(**to:** *Source*) | **labelled**(*Label*, *Effect*)

FO.Location ::= **register**(*Register*) | **frame**(*Frame.Location*)

FO.Register ::= **zero** | **ra** | **s1** | **a0** | **a1** | **a2** | **a3** | **a4** | **a5** | **a6** | **a7** | **s2** | **s3** | **s4** | **s5** | **s6** | **s7** | **s8** | **s9** | **s10** | **s11** | **invocationData**

MM (Managed Memory) A language that introduces a runtime, call stack, heap, and operations on them.

MM.Program ::= (*[Effect]*)

MM.Effect ::= **copy**(*DataType*, **into:** *Register*, **from:** *Register*) | **compute**(**destination:** *Register*, *Register*, *BinaryOperator*, *Source*) | **load**(*DataType*, **into:** *Register*, **from:** *Frame.Location*) | **store**(*DataType*, **into:** *Frame.Location*, **from:** *Register*) | **createBuffer**(**bytes:** *Source*, **capability:** *Register*, **onFrame:** *Bool*) | **destroyBuffer**(**capability:** *Register*) | **loadElement**(*DataType*, **into:** *Register*, **buffer:** *Register*, **offset:** *Source*) | **storeElement**(*DataType*, **buffer:** *Register*, **offset:** *Source*, **from:** *Register*) | **deriveCapability**(**in:** *Register*, **to:** *Label*) | **createSeal**(**in:** *Register*) | **seal**(**into:** *Register*, **source:** *Register*, **seal:** *Register*) | **pushFrame**(*Frame*) | **popFrame** | **permit**(*[Permission]*, **destination:** *Register*, **source:** *Register*) | **clearAll**(**except:** *[Register]*) | **branch**(**to:** *Label*, *Register*, *BranchRelation*, *Register*) | **jump**(**to:** *Target*) | **call**(*Target*) | **callSealed**(*Register*, **data:** *Register*, **returnPoint:** *Label*) | **return**(**to:** *Target*) | **labelled**(*Label*, *Effect*)

MM.Register ::= **zero** | **ra** | **s1** | **a0** | **a1** | **a2** | **a3** | **a4** | **a5** | **a6** | **a7** | **s2** | **s3** | **s4** | **s5** | **s6** | **s7** | **s8** | **s9** | **s10** | **s11** | **t4** | **t5** | **invocationData**

MM.Target ::= *Label* | **label**(*Label*) | **register**(*Register*)

MM.Source ::= *Int* | **constant**(*Int*) | **register**(*Register*)

MM.Frame ::= (**allocatedByteSize:** *Int*)

RT (Runtime) A language that introduces a runtime system and runtime routines.

RT.Program ::= (*[Statement]*)

RT.Effect ::= **copy**(*DataType*, **into:** *Register*, **from:** *Register*)
- | **compute**(**destination:** *Register*, *Register*, *BinaryOperator*, *Source*)
- | **load**(*DataType*, **destination:** *Register*, **address:** *Register*, **offset:** *Int*)
- | **store**(*DataType*, **address:** *Register*, **source:** *Register*, **offset:** *Int*)
- | **deriveCapabilityFromPCC**(**destination:** *Register*, **upperBits:** *UInt*)
- | **deriveCapabilityFromLabel**(**destination:** *Register*, **label:** *Label*)
- | **offsetCapability**(**destination:** *Register*, **source:** *Register*, **offset:** *Source*)

| **getCapabilityLength**(**destination:** *Register*, **source:** *Register*)
| **setCapabilityBounds**(**destination:** *Register*, **base:** *Register*, **length:** *Source*)
| **getCapabilityAddress**(**destination:** *Register*, **source:** *Register*)
| **setCapabilityAddress**(**destination:** *Register*, **source:** *Register*, **address:** *Register*)
| **getCapabilityDistance**(**destination:** *Register*, **cs1:** *Register*, **cs2:** *Register*)
| **seal**(**destination:** *Register*, **source:** *Register*, **seal:** *Register*)
| **sealEntry**(**destination:** *Register*, **source:** *Register*)
| **permit**(*[Permission]*, **destination:** *Register*, **source:** *Register*, **using:** *Register*)
| **clear**(*[Register]*)
| **branch**(**to:** *Label*, *Register*, *BranchRelation*, *Register*)
| **jump**(**to:** *Target*, **link:** *Register*)
| **invoke**(**target:** *Register*, **data:** *Register*)
| **callRuntimeRoutine**(**capability:** *Label*, **link:** *Register*)

RT.Statement ::= **effect**(*Effect*) | **padding**(**alignment:** *DataType*) | **data**(**type:** *DataType*, **value:** *Int*, **count:** *Int*) | **bssSection** | **labelled**(*Label*, *Statement*)

CE (Canonical Effects) A language grouping related instructions under a single effect.

CE.Program ::= (*[Statement]*)

CE.Statement ::= **effect**(*Effect*) | **padding**(**alignment:** *DataType*) | **data**(**type:** *DataType*, **value:** *Int*, **count:** *Int*) | **bssSection** | **labelled**(*Label*, *Statement*)

CE.Permission ::= **global** | **execute** | **load** | **store** | **loadCapability** | **storeCapability** | **storeLocalCapability** | **seal** | **invoke** | **unseal** | **setCID**

CE.DataType ::= **u8** | **s32** | **cap**

CE.Source ::= *Int* | **constant**(*Int*) | **register**(*Register*)

CE.Target ::= *Label* | **label**(*Label*) | **register**(*Register*)

CE.Effect ::= **copy**(*DataType*, **into:** *Register*, **from:** *Register*)
| **compute**(**destination:** *Register*, *Register*, *BinaryOperator*, *Source*)
| **load**(*DataType*, **destination:** *Register*, **address:** *Register*, **offset:** *Int*)
| **store**(*DataType*, **address:** *Register*, **source:** *Register*, **offset:** *Int*)
| **deriveCapabilityFromPCC**(**destination:** *Register*, **upperBits:** *UInt*)
| **deriveCapabilityFromLabel**(**destination:** *Register*, **label:** *Label*)
| **offsetCapability**(**destination:** *Register*, **source:** *Register*, **offset:** *Source*)
| **getCapabilityLength**(**destination:** *Register*, **source:** *Register*)
| **setCapabilityBounds**(**destination:** *Register*, **base:** *Register*, **length:** *Source*)
| **getCapabilityAddress**(**destination:** *Register*, **source:** *Register*)
| **setCapabilityAddress**(**destination:** *Register*, **source:** *Register*, **address:** *Register*)
| **getCapabilityDistance**(**destination:** *Register*, **cs1:** *Register*, **cs2:** *Register*)
| **seal**(**destination:** *Register*, **source:** *Register*, **seal:** *Register*)
| **sealEntry**(**destination:** *Register*, **source:** *Register*)
| **permit**(*[Permission]*, **destination:** *Register*, **source:** *Register*, **using:** *Register*)
| **clear**(*[Register]*)
| **branch**(**to:** *Label*, *Register*, *BranchRelation*, *Register*)
| **jump**(**to:** *Target*, **link:** *Register*)
| **invoke**(**target:** *Register*, **data:** *Register*)

RV (CHERI-RISC-V) A language that maps directly to CHERI-RISC-V assembly statements, i.e., labels, instructions, and directives.

RV.Program ::= (*[Statement]*)

RV.Register ::= **zero** | **ra** | **sp** | **gp** | **tp** | **t0** | **t1** | **t2** | **fp** | **s1** | **a0** | **a1** | **a2** | **a3** | **a4** | **a5** | **a6** | **a7** | **s2** | **s3** | **s4** | **s5** | **s6** | **s7** | **s8** | **s9** | **s10** | **s11** | **t3** | **t4** | **t5** | **t6**

RV.Statement ::= **instruction**(*Instruction*) | **padding**(**byteAlignment:** *Int*) | **data**(**value:** *Int*, **datumByteSize:** *Int*, **count:** *Int*) | **bssSection** | **labelled**(*Label*, *Statement*)

RV.Label ::= *String*

RV.Instruction ::= **copyWord**(**destination:** *Register*, **source:** *Register*)
| **copyCapability**(**destination:** *Register*, **source:** *Register*)
| **computeWithRegister**(**operation:** *BinaryOperator*, **rd:** *Register*, **rs1:** *Register*, **rs2:** *Register*)
| **computeWithImmediate**(**operation:** *BinaryOperator*, **rd:** *Register*, **rs1:** *Register*, **imm:** *Int*)
| **loadByte**(**destination:** *Register*, **address:** *Register*, **offset:** *Int*)
| **loadSignedWord**(**destination:** *Register*, **address:** *Register*, **offset:** *Int*)
| **loadCapability**(**destination:** *Register*, **address:** *Register*, **offset:** *Int*)
| **storeByte**(**source:** *Register*, **address:** *Register*, **offset:** *Int*)
| **storeSignedWord**(**source:** *Register*, **address:** *Register*, **offset:** *Int*)
| **storeCapability**(**source:** *Register*, **address:** *Register*, **offset:** *Int*)
| **deriveCapabilityFromLabel**(**destination:** *Register*, **label:** *Label*)
| **deriveCapabilityFromPCC**(**destination:** *Register*, **upperBits:** *UInt*)
| **offsetCapability**(**destination:** *Register*, **source:** *Register*, **offset:** *Register*)
| **offsetCapabilityWithImmediate**(**destination:** *Register*, **source:** *Register*, **offset:** *Int*)
| **getCapabilityLength**(**destination:** *Register*, **source:** *Register*)
| **setCapabilityBounds**(**destination:** *Register*, **base:** *Register*, **length:** *Register*)
| **setCapabilityBoundsWithImmediate**(**destination:** *Register*, **base:** *Register*, **length:** *Int*)
| **getCapabilityAddress**(**destination:** *Register*, **source:** *Register*)
| **setCapabilityAddress**(**destination:** *Register*, **source:** *Register*, **address:** *Register*)
| **getCapabilityDistance**(**destination:** *Register*, **cs1:** *Register*, **cs2:** *Register*)
| **seal**(**destination:** *Register*, **source:** *Register*, **seal:** *Register*)
| **sealEntry**(**destination:** *Register*, **source:** *Register*)
| **permit**(**destination:** *Register*, **source:** *Register*, **mask:** *Register*)
| **clear**(**quarter:** *Int*, **mask:** *UInt8*)
| **branch**(**rs1:** *Register*, **relation:** *BranchRelation*, **rs2:** *Register*, **target:** *Label*)
| **jump**(**target:** *Label*, **link:** *Register*)
| **jumpWithRegister**(**target:** *Register*, **link:** *Register*)
| **invoke**(**target:** *Register*, **data:** *Register*)

RV.BranchRelation ::= **eq** | **ne** | **lt** | **le** | **gt** | **ge**

RV.BinaryOperator ::= **add** | **sub** | **mul** | **and** | **or** | **xor** | **sll** | **srl** | **sra**

S (CHERI-RISC-V Assembly) The ground language as provided to Clang for assembly and linking.

S.Program ::= *String*

Appendix B

Sisp

Sisp (from "Swifty Lisp") is a data interchange format (comparable to JSON) developed as part of the Glyco compiler and used for textually representing programs. It draws inspiration from both the Swift programming language (in which Glyco is developed) and the S-expression syntax from the Lisp family of programming languages. It attempts to be concise by removing where possible syntactical elements such as list delimiters or quotes around strings, but at the same time improve clarity with features such as labelled attributes.

This chapter briefly introduces the Sisp format and its mapping to Swift types. The language reference presented in Appendix A is sufficient to read and write Glyco programs; knowledge of Sisp is only relevant for understanding and modifying the implementation of the compiler itself. This chapter assumes some working knowledge in Swift and the `Encoder` and `Decoder` protocols provided by the standard library, as documented by Ferber et al. [2017].

Sisp value

A Sisp value is one of the following:

- A positive or negative integer, e.g., `-13`, `5`, and `+5`. The sign is optional for nonnegative integers.
- A string value, e.g., `large` or `"my text"`. The string does not need to be quoted when
  - it is nonempty;
  - it contains only alphanumerical characters (Unicode General Category L*, M*, and N*), _, $, ., and %; and
  - it begins with a letter (Unicode General Category L* or M*), _, $, ., or %.

  Quotation characters within a quoted string can be escaped by doubling them, e.g., `"this is a ""quoted"" string"`.
- A list of values, written consecutively and separated by whitespace, e.g., `1 2 3 5 8 13`.

- An untyped structure containing comma-separated unlabelled and labelled attributes, written as `("John", age: 50, "favourite colour": black)`. A labelled attribute is a key–value pair where the key is a string value; the key and value are separated by a colon. An unlabelled attribute is simply a value and must appear in the expected position (e.g., first) in the attribute list.[1]
- A typed structure of some type `type` and containing comma-separated attributes, written as `type("John", age: 50, "favourite colour": black)`.

Encoding Swift values as Sisp values

The Glyco compiler ships with a Sisp encoder and decoder, which apply the following mapping rules between Swift and Sisp values:

- `String`s and integers as well as `String`- and `Int`-representable values are encoded as string resp. integer Sisp values. This also applies to `PartiallyIntCodable` and `PartiallyStringCodable` values whose integer resp. string value is non-`nil`.
- `Bool`s and `PartiallyBoolCodable` values with a non-`nil` Boolean value are encoded as the string value `false` or `true`.
- `Collection`s including arrays are encoded as lists of Sisp values.
- A value of an `enum` type with no associated values is encoded as a string value containing the name of the `case`.
- A value of an `enum` type with associated values is encoded as a structure typed with the name of the `case`. Each labelled resp. unlabelled associated value is encoded as a labelled resp. unlabelled attribute.
- A value of a `struct` (or `class`) type is encoded as an untyped structure. Each property is encoded as a labelled attribute.

The above mapping rules assume an encoding and decoding implementation synthesised by the Swift compiler. They can be adjusted by providing a custom implementation. In particular, coding keys are used for the labels of attributes; a value encoded using a coding key whose string value consists of an underscore and an integer is encoded as an unlabelled attribute, with the number indicating the zero-based position within the attribute list of the Sisp structure.

[1] An attribute's position in a structure's attribute list is determined by the number of (labelled and unlabelled) attributes preceding it.

Glossary

basic block a labelled sequence of effects that is executed from beginning to end, before jumping or branching to itself or to other blocks. No effect in the block but the first is labelled. No effect except the continuation is a jump, call, or branch. 26, 28, 65, 80

callee-saved register a register that a procedure can only use after saving its previous contents, to be restored when the procedure is done using the register; contrast with caller-saved register. 15, 16, 18, 19, 64, 80

caller-saved register a register that a procedure can use freely but whose contents it must save if it wishes to preserve them across a procedure call; contrast with caller-saved register. 15, 16, 80, 81

calling convention a set of rules governing the invocation of procedures on a specific platform and architecture dealing with matters such as argument & result passing, register availability, call frame structure, jumping to the callee, and returning to the caller. iv, v, 2, 3, 9, 14–16, 30–32, 38, 40–44, 63, 64, 80–82

frame capability a capability in the `cfp` register that points to the base of the current call frame. 15, 16, 34, 35, 37, 80

GCCC Glyco Conventional Calling Convention. 15, 30, 34, 40, 41, 63, 80, 81, *see* Glyco Conventional Calling Convention

GHSCC Glyco Heap-based Secure Calling Convention. iv, v, 30, 31, 33, 34, 36, 38, 40, 41, 51, 55, 63, 65, 80, 81, *see* Glyco Heap-based Secure Calling Convention

Glyco Conventional Calling Convention a calling convention implemented in Glyco that is similar to a RISC-V calling convention. 15, 80, 81

Glyco Heap-based Secure Calling Convention a calling convention implemented in Glyco that ensures local state encapsulation by securely allocating call frames on the heap, clearing non-argument and non-result registers before passing control, and sealing return & frame capabilities with a unique seal before invocation. 30, 31, 80, 81

heap capability a capability that grants access to the heap and points to the next free location. 32, 33, 38, 41, 63, 80

IL intermediate language. 11, 12, 17, 19, 27, 28, 30, 38, 40, 45, 47, 50, 55, 57–62, 64–66, 80, 82, *see* intermediate language

initialiser a pseudo-method invoked on any new object of the type for which the initialiser is defined, ensuring that the new object has an appropriate initial state. 50–52, 61, 80

intermediate language a language produced or parsed by a nanopass as part of the compiler's routine. 11, 80, 82

local state encapsulation a security property guaranteeing that a procedure's state cannot be accessed by adversarial code in the same address space. iv, v, 2, 31, 34, 37, 41, 42, 44, 55, 63, 80, 81

lower language the IL output by a single lowering of a program in the IL currently being discussed. 11, 16, 20, 40, 64, 80

lowering the act of transforming a higher-level to a lower-level intermediate language through one or more nanopasses. 9, 11, 13, 14, 16, 18, 19, 21–23, 26, 27, 36, 38, 40, 46, 52–55, 57, 60, 80, 82

method a function that can be invoked on an object and gets access to the object's state, if defined on the same type as the object. 50–52, 80, 82

method capability a capability to a method's code, sealed with a seal capability only used for objects and methods of the same object type. 52, 54, 80

nanopass a function that transforms a program in one IL to a program in another IL. iv, v, 2, 3, 9, 11–14, 16, 18, 19, 22, 23, 25–29, 36, 38, 40, 46, 49, 52, 55, 60–65, 80, 82

object capability a capability to a state record embodying a sealed object, sealed with a seal capability only used for objects and methods of the same object type. 52, 80

object type a numerical property of a capability indicating if it is unsealed (object type −1), sealed as a sentry capability (object type −2), or sealed using a seal capability (object type matching the seal capability's address). 6–8, 34, 35, 37, 80

return capability a capability containing a return address allowing a procedure to return to its caller. iv, 2, 8, 17–19, 31, 32, 34–37, 41–44, 63, 65, 80

runtime compiler-provided software that runs at program startup, initialises data structures that are needed as part of the operation of the user program, and provides routines with privileges normally not afforded to user programs. 17, 32–34, 36, 38, 51, 55, 80, 83

runtime routine a procedure provided by the runtime with a possibly different calling convention and more privileges afforded to it than a normal procedure. 30, 32, 34, 38, 40, 51, 55, 63, 80, 83

s-call a procedure call mediated by the s-call runtime routine which seals the frame and return capabilities using a unique seal capability, thereby ensuring the callee cannot dereference either capability and requiring that the callee use `CInvoke` to return to the caller. 34, 35, 37, 40, 51, 55, 80

seal capability a capability with the *Permit seal* permission which can be used with the `CSeal` instruction to seal an unsealed capability with the seal capability's address. 6, 7, 33–35, 37, 43, 44, 47, 51–53, 55, 57, 80, 82, 83

sealing the act of setting the object type of an unsealed capability to either the address of a seal capability or to a special value indicating that it is a sentry capability, so that it can no longer be modified or dereferenced until it is unsealed again. 5, 6, 34, 51, 52, 80

sentry capability a sealed entry capability, a sealed capability created with the `CSealEntry` instruction which can only be unsealed by jumping to its address using the `CJALR` instruction. 6–8, 32, 33, 38, 40, 42, 80, 82, 83

stack capability a capability in the `csp` register that points to the last pushed datum on the call stack, i.e., the top of the stack. 15, 41–44, 80

unseal capability a capability with the *Permit unseal* permission which can be used with the `CUnseal` instruction to unseal a capability sealed with the unseal capability's address. 7, 37, 80

unsealing the act of setting the object type of a sealed capability to -1 so that it is no longer sealed and thus can be used normally. 6, 80, 83

user program a program not provided by the compiler or operating system, normally contrasted with the runtime. 32, 33, 38, 80, 82

well-bracketed control flow a security property guaranteeing that a procedure can only invoke other procedures, return to its caller, or diverge. iv, v, 2, 31, 32, 37, 38, 41, 42, 44, 63, 65, 80

Bibliography

William J. Bowman, Steven Keuchel, and Dominique Devriese. Compilers. http://soft.vub.ac.be/compilers/book_top.html, 2022.

Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph colouring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, May 1994.

Nicolas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. *ACM*, 1994.

Itai Ferber, Michael LeHew, and Tony Parker. Swift archival & serialization. https://github.com/apple/swift-evolution/blob/main/proposals/0166-swift-archival-serialization.md, April 2017.

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cerise: Program verification on a capability machine in the presence of untrusted code. *ACM*, October 2021a.

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialised capabilities. *Proc. ACM Program. Lang.*, 5:30, January 2021b.

Tony Hoare. Null references: The billion dollar mistake. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/, August 2009.

Sander Huyghebaert. Secure calling convention with uninitialised capabilities. Master's thesis, Vrije Universiteit Brussel, 2020.

Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *ICFP '13: Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, 18, page 7. ACM, September 2013.

Henry M. Levy. *Capability-Based Computer Systems*. Bedford: Digital Press, 1984.

Jerome H. Saltzer. *Protection and the Control of Information Sharing in Multics*, volume 17. ACM, July 1974.

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass framework for compiler education. *J. Functional Programming*, page 15, 2004.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. *ACM Trans. Program. Lang. Syst.*, 42(1):53, December 2019a.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *ACM Programming Languages*, 3(19):28, January 2019b.

The Chromium Authors. Memory safety — the Chromium projects. https://www.chromium.org/Home/chromium-security/memory-safety, 2022.

Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, document version 20191213 edition, December 2019.

Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalisation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, May 2015.

Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An introduction to CHERI. Technical Report 941, University of Cambridge Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, September 2019.

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report 951, University of Cambridge Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, October 2020.