SKELETAL SEMANTICS FOR MESSAGE-PASSING CONCURRENCY

MIGUEL BARTELSMAN MEJÍA

From Actors to Channels and Back

July 2022 – version 3

Miguel Bartelsman Mejía: *Skeletal Semantics for Message-Passing Concurrency,* From Actors to Channels and Back, © July 2022

ABSTRACT

This project aims to stand at the junction between two needs in computing science: the need for easier and more portable concurrency, and the need for more universal formal specifications of programming languages. Making use of skeletal semantics—a novel framework for describing programming language semantics—we present two concurrent programming languages, one for the actor model and one for the channel model of process communication, along with their interpreters and two utilities for translating programs between the two languages. The goal is to test the capabilities of skeletal semantics in the context of message-passing concurrent programming languages.

ACKNOWLEDGMENTS

This project would not have been possible without the support of my friends and colleagues.

In particular, I want to thank Chingiz Dadashov-Khandan for his support throughout the project and for his help in proofreading this document. I want to thank my supervisors, Dan Frumin and Jorge A. Pérez, for their patience and assistance during this process. And, last but not least, I want to thank my family for their constant support over the years; I would not have made it this far without them.

CONTENTS

1 INTRODUCTION 1
  1.1 Skeletal semantics 2
  1.2 Concurrent programming 5
  1.3 Goals 7

2 LANGUAGES 9
  2.1 Base language 9
  2.2 Actor-based language 11
  2.3 Channel-based language 11

3 DEVELOPMENT 13
  3.1 Semantics in Skel 13
  3.2 Describing the semantics 14
  3.3 OCaml workflow 17
  3.4 Common language details 18
  3.5 Actors implementation 20
  3.6 Channels implementation 21
  3.7 Translators 21

4 ARCHITECTURE AND RESULTS 23

5 CONCLUSION 25

BIBLIOGRAPHY 27

6 BIBLIOGRAPHY 27

LIST OF FIGURES

Figure 1 Example semantics for an if-expression of an unspecified language 4
Figure 2 Example program in the base language 9
Figure 3 Example program avoiding the use of `Let` 10
Figure 4 A simple actor-based program. This program spawns a new actor and sends it the integer value 7. 11
Figure 5 A simple channel-based program. This program creates a process and a channel to send it the integer value 7. 12
Figure 6 Types in Skel 13
Figure 7 Filters and skeletons in Skel 14
Figure 8 Heuristic for translating from operational to skeletal semantics 15
Figure 9 Heuristic for combining skeletons 16
Figure 10 Example of variable mapping 17
Figure 11 Diagram showcasing concurrent process scheduling and evaluation. 20
Figure 12 Public project structure. Modules in purple, constructors in orange, functions in blue. 23
Figure 13 Process diagram for the use of the four tools 24
Figure 14 Example of the process and results for one of the interpreters. Orange are expression nodes, red are value nodes, green are values. 24

LIST OF TABLES

Table 1 Base language terms. e stands for terms, v stands for values, n stands for names 10
Table 2 Actor-based language exclusive terms. 11
Table 3 Channel-based language exclusive terms. 12

1

INTRODUCTION

Never before has concurrent computation been more important for our daily lives: the internet of things promises networks of devices in constant communication; cellphones and other wireless devices tap into global communication networks; cloud computing makes simultaneous use of devices around the globe to provide advanced and powerful capabilities to individual terminals; modern personal computers are powered by CPUs hosting several parallel cores, aided by GPUs with hundreds more; and data-centres must operate hundreds to thousands of servers harmoniously and concurrently to provide satisfactory services to their clients. This trend towards a concurrent future shows no sign of stopping. Moore's law—the empirical observation stating that the density of transistors within an integrated circuit double every two years (Moore, 1998)—is becoming increasingly difficult to maintain due to the physical limitations of nanoscale technology. Instead, much effort is being invested into computers using transistors more efficiently. An effort that relies, in no small part, on the parallelization of computations and the speedups that this achieves.

Concurrency is not cheap, however. Programming, even without the use of concurrency, is already notorious for its difficulty and the impact that its errors can have. A small bug present in a widely used library can be the source for multi-million dollar outages or vulnerabilities, such was the case with the infamous Heartbleed OpenSSL vulnerability, which affected between 24 and 55% of all HTTPS sites (Durumeric et al., 2014). Concurrency compounds the complexity of producing correct software: it brings to the table a whole set of potential pitfalls to avoid—such as deadlocks, livelocks, data and condition races, among others—and it makes programs significantly more difficult to test, debug, and reason about. Concurrency is expensive and its price is paid by the developers who must ensure that their software is correct.

As such, there is a great need to research methods that ensure the safety of concurrent software. Many theoretical frameworks have been developed to this end: Process-calculi formalize concurrent computations under a strict mathematical framework; session types allow for the verification and manipulation of complex communication protocols; and new data structures, schemes, and algorithms are constantly being developed for the purpose of building safer and faster concurrent applications.

However, many of such efforts are only possible when the foundations upon which they are built can be reasoned about and trusted. To establish those solid foundations for research—ones that ensure that theory aligns with reality—researchers must understand how their tools work, and their tools must behave according to the expectations of their users. In the realm of programming languages, this means that a language's behaviour must be clear and well-specified, and the interpreters and compilers for such languages must correctly abide by such specification.

This presents a problem: writing software that has been verified to be correct, according to a rigorous specification, requires a great amount of effort and resources that might be better used to further other goals. *Skeletal semantics* (Bodin et al., 2019) alongside *Necro* (Courant, Crance, and Schmitt, 2020; *Necro Library* 2019) are two tools that aim to solve this problem. Skeletal semantics provides a more scalable and reusable means of describing the semantics of programming languages, and Necro offers a partly-automated means of implementing a correct language interpreter based on such specifications.

1.1 SKELETAL SEMANTICS

To understand how skeletal semantics work, we first introduce the concept of *semantics* within the context of Computing Science and programming languages. Semantics can refer to either the field of study that deals with the rigorous description of the behaviour and properties of programming languages or to the concrete language descriptions produced in this field. There are three major schools for describing the semantics of a language: operational semantics, denotational semantics, and axiomatic semantics.

This project focuses exclusively on operational semantics because, out of the three approaches, it is the most commonly used, it is the approach that skeletal semantics takes, and it is the only approach considered for this project. Operational semantics describes languages through sets of reduction or evaluation rules that are repeatedly applied to the constructs within a program. These rules can be applied iteratively, evaluating programs step by step until they have been fully reduced (or until reduction is no longer possible), or they can be applied recursively by evaluating programs as a series of nested computations of decreasing complexity. The former approach is called structural or small-step operational semantics, while the latter is known as natural or big-step semantics. There are other, more elaborate schemes for describing operational semantics, but small- and big-step styles are the dominant ones. (Nielson and Nielson, 2007; Pierce, 2002)

Skeletal semantics is a theoretical framework for describing the operational semantics of a programming language. It is capable of representing small-step and big-step semantics[1] but it is primarily designed to be recursive in nature and, thus, is better suited to big-step semantics style. The skeletal semantics of a language is composed of a series of rules known as skeletons in skeletal semantics jargon. Each skeleton is described by a series of bones—computations—which can be arranged either in sequence or in a branching structure.

Bones can be of two types: skeletons (where the recursive nature of the framework stems from) and filters. Filters are abstractions that serve a dual purpose: they can play the role of functions—mapping an input to an output—or that of predicates—capable of failing and halting the branch of execution they appear in. Filters are particularly useful for applied language development, as they can describe behaviours that cannot be effectively described by the semantics alone.[2] All of these can be represented by filters without the need for *ad-hoc* notation or novel mathematical definitions. Bones must be well-typed and can be used in combination with pattern-matching and name-binding. The types used for these skeletons can be either base-types—which, much like filters, are abstractions—or program-types—which are composite types that build upon other program-types and base-types (Bodin et al., 2019). Figure 1 shows a simple comparison between traditional operational semantics and skeletal semantics, highlighting the different components of the latter.

Skeletons, by themselves, are not sufficient for a complete semantics; an *interpretation* is also required. Interpretations are a set of definitions that are used to describe how skeletons in a language's semantics should evaluate a program. *Skeletal semantics and their interpretations*, by Bodin et al. (2019), introduces three example interpretations: a concrete interpretation, an abstract one, and a well-formedness one. The concrete interpretation evaluates a program by following a single branch of execution within the semantics, even if multiple branches would be successful. Abstract interpretations, instead, accept all successful branches, yielding the set of all possible results for each evaluated program. Lastly, the well-formedness interpretation evaluates whether a program is valid (well-formed) or not, according to the types described by the semantics.

One of the primary advantages of skeletal semantics over other frameworks is that it is defined such that the semantics can be mapped to a correct language interpreter implementation. Necro is a tool that automates that very task. By taking as input a textual encoding of the semantics in a language called Skel, Necro is capable of producing a partial implementation of a language interpreter that abides by the semantics. As of the writing of this document, Necro can target both OCaml and the Coq proof assistant, both of which require the user to

---

[1] More niche schemes, such as Pretty-Big-Step semantics are also supported, but they may require creative approaches to incorporate.

[2] Things that filters can interface with include: system calls, I/O, special and proprietary encodings and programs, foreign-library functions, among others.

$$\textsc{If-true}\ \frac{t_1 \Downarrow \mathtt{true} \qquad t_2 \Downarrow v_2}{\mathit{if}\ (t_1)\ \mathit{then}\ (t_2)\ \mathit{else}\ (t_3) \Downarrow v_2}$$

$$\textsc{If-false}\ \frac{t_1 \Downarrow \mathtt{false} \qquad t_3 \Downarrow v_3}{\mathit{if}\ (t_1)\ \mathit{then}\ (t_2)\ \mathit{else}\ (t_3) \Downarrow v_3}$$

(a) A rule in traditional operational semantics

$$\textsc{If}\ (\mathit{if}\ (t_1)\ \mathit{then}\ (t_2)\ \mathit{else}\ (t_3)) :=$$
$$\left[\ \mathtt{is\_bool}(t_1)\ ;\ \left(\begin{array}{c} H(x_\sigma, t_1, x_f)\ ?\triangleright\ \mathtt{true}\ ;\ H(x_f, t_2, x_o) \\ H(x_\sigma, t_1, x_f)\ ?\triangleright\ \mathtt{false}\ ;\ H(x_f, t_3, x_o) \end{array}\right)\ \right]$$

(b) A skeleton in skeletal semantics. Legend: filter, branches, pattern-matching, recursion. Semicolons declare sequencing.

```
val eval_if (input: expression): value =
    let If (t1, t2, t3) = input in
    is_bool t1 ;
    branch
        let True = eval t1 in
        eval t3
    or
        let False = eval t1 in
        eval t3
    end
```

(c) Skeletal semantics in Skel

Figure 1: Example semantics for an if-expression of an unspecified language

supply implementations for all abstractions defined in the semantics. Other targets are also potentially possible, but no others are currently available. (Courant, Crance, and Schmitt, 2020; *Necro Library* 2019)

The automation provided by Necro has the potential to greatly reduce the effort that language developers must invest into creating a correct interpreter for a language. With Necro, rather than focusing on the development of tooling, efforts can instead be redirected into researching the applications for, and behaviour of the relevant languages, atop solid foundations.

With regards to concurrency, skeletal semantics is still a recent development; many of its potential applications have yet to be discovered or explored in-depth. Concurrent programming, especially, is almost entirely neglected—barring the implementation of co-routines in Jskel, an ECMAScript implementation made using Necro (Khayam, 2020). This presents an opportunity for research projects to delve into possible concurrent applications for skeletal semantics.

1.2 CONCURRENT PROGRAMMING

There are two primary approaches to concurrent programming: shared-state concurrency and message-passing concurrency. Shared-state concurrency poses an approach where multiple concurrent processes can communicate via a shared state (as the name implies). This shared state is most often a shared memory heap, though it need not be. Shared-state concurrency is a common approach for software running on personal computers and similar systems as it mirrors the underlying architecture of these devices: multiple cores that use a common memory device for storing program state. This makes implementing concurrency at a low level straightforward. (*Shared State Concurrency* 2005)

However, shared-state concurrency has some drawbacks that make it difficult to reason about concurrency. The non-deterministic nature of concurrent execution, combined with a shared, mutable state, results in processes that compete for read and write access to data which, in turn, yields unpredictable results in the form of data races. These races are sometimes catastrophic in nature, and thus easy to detect. But, sometimes, visible misbehaviour is only rarely triggered, making bugs hard to identify, track down, and fix. To prevent this, many synchronization data structures, algorithms, and conventions have been devised over the years; their use is often complex and introduces additional pitfalls that programmers must be aware of during the development process.

Furthermore, shared-state storage is difficult to scale to large physical distances. Even systems on the scale of a personal computer rely on non-shared caches to keep latency low during memory-related operations. These non-shared caches must store shared data, which requires complex synchronization procedures to prevent errors caused by parallel execution. Despite such schemes, *cache misses*—situations where local synchronization has failed to keep up with process execution—occur regularly and are a major concern for the performance of applications. With such issues arising at the centimetre scale, at the distances that distributed systems operate at, problems are magnified to the point of making the shared-state approach unscalable.

Message-passing concurrency sidesteps these two issues entirely by replacing shared-state in favour of discrete immutable messages for process communication. The channels by which these messages are transferred are abstract, which allows for them to be scaled up or down without affecting program design. In addition, the discrete and immutable nature of the messages means that data races are entirely avoided (*Message Passing Concurrency* 2010). This design also allows, with the help of type theory, for the design of complex but verifiable communication protocols to ensure the correct behaviour of concurrent applications.

Within the realm of message-passing, there are numerous theoretical models for reasoning about concurrency. Two of the better-known and most influential among these are the *actors* and the *channels* models. The actors model, first described in Agha's *Actors: A Model of Concurrent Computation in Distributed Systems*[3] (Agha, 1986) represents concurrent computations as performed by their namesakes, actors, which are processes capable of sending and receiving messages to and from one another. Each actor is associated with a mailbox where arriving messages are queued up, which can then be processed by the actor to execute conditional computation.

The channel model, founded by Milner with his *Calculus for Communicating Systems* (Milner, 1980), separates the concerns of computation and communication. In this model, rather than just actors, there are processes and channels. Channels are independent mailboxes that processes can interact with to communicate messages. Processes, unlike actors, have no associated mailboxes and cannot communicate directly with one another; message-passing is, instead, done via channels. Additionally, while the actor model is necessarily asynchronous, the channel model can be either synchronous—in which case synchronization is needed for communication—or asynchronous (Milner, Parrow, and Walker, 1992a,b).

Even though the actors and channels models pose two different ways to reason about concurrency, the two models are equivalent when dealing with asynchronous communication. This is made clear by Fowler, Lindley, and Wadler (2016), who have presented two translation schemes for the models, one going from the actors model to the channels model, and the other in the opposite direction. These translation schemes show that any program described with one model can be described in the other, thus proving their equivalence in computational capabilities. Since the synchronous channels model is not relevant within the scope of this discussion, going forward, all references to the channel model will refer to the asynchronous version.

---

[3] Its formal semantics were described a decade later by Agha et al. (1997)

1.3 GOALS

This project positions itself at the intersection point between skeletal semantics, message-passing concurrency, and language engineering. It stands there intending to explore the possibilities that skeletal semantics offers and the limitations that constrain it in the realm of language engineering; particularly, for languages that employ the actors and channels models of message-passing concurrency.

The hope is that the exploration that was undertaken during the development of this project, and the resulting products, will help in paving the way for future research into the development of concurrent programming languages, and tools to make this research more easily possible. That, by providing an account of a development process aided by skeletal semantics and Necro, future researchers and users will be able to reference these experiences and put them to use for their own purposes. And that, as a result of this, research in the fields of concurrency and programming languages can be further expanded.

To this end, four software tools have been developed: two interpreters for two programming languages, one making use of the actor model of concurrency and the other making use of the channel model; and two translator utilities, one translating programs from the actor model to the channel model, and the other translating in the opposite direction. All four tools have been developed by using skeletal semantics to specify their behaviour and by leveraging Necro to automatically generate a partial implementation in OCaml.

With the language interpreters, the goal is to explore the possibilities of skeletal semantics for implementing concurrent languages. The goal of the translation utilities is to evaluate the use of skeletal semantics not just for the implementation of languages, but for their manipulation, when guided by and compared against formal theoretical propositions.

These four tools, their development process, their use, and the conclusions drawn from them, will be described in the following sections.

2

LANGUAGES

Two languages were developed for this project, one for each of the two relevant models of concurrency: an actor-based language, and a channel-based language. The two languages were designed around a non-concurrent base language, allowing them to share a common core of semantics.

The language is written using S-expressions. These are parenthetical expressions that contain a token, describing an operation, followed by a list of arguments. S-expressions were chosen as the syntactic foundation for the language because they are consistent, simple to parse and serialize, and there is extensive tooling supporting their use in OCaml. This makes their implementation straightforward, allowing the project to focus on the semantics of the languages, rather than their grammar.

2.1 BASE LANGUAGE

There are two reasons for the use of a base language as a common core. First, the non-concurrent aspects of the languages are not the intended focus of the project, so minimizing efforts on this front is beneficial. Second, the references used as a guideline for the development of these languages (Fowler, Lindley, and Wadler, 2016) also describe the two languages as sharing the non-concurrent aspects of the semantics.

This base language is an extension of the typed λ-calculus, with the addition of 2-tuples product types, Left/Right sum types, recursive functions, simple arithmetic operations on integers, sequencing, and fine-grained eager evaluation.

Recursion, sum types and product types are necessary for the translation schemes; arithmetic was included to allow for easily identifiable computation results; finally, sequencing (`Seq`) and fine-grained eager evaluation (`Let`) were included to make programs easier to read and write, which is exemplified by comparing Figure 2 and Figure 3.

```
(Let (
    (x) (Ret (IntVal 32))
    (Add (
        (Var x)
        (Ret (IntVal 10)) )) ))
```

Figure 2: Example program in the base language

```
(Call (
    (Fun (
        (x)
        (Add (
            (Var x)
            (Ret (IntVal 10)) )) ))
    (Ret (IntVal 32)) ))
```

Figure 3: Example program avoiding the use of `Let`

Table 1 lists all the possible operations that the expression nodes can take in the base language. Each of the two concurrent languages extends this list with additional concurrent and communication operations.

`(Call` e_f e_v`)`	Calls function e_f with e_v
`(Func` n e`)`	Binds variable n in e
`(RecFunc` n_f n_v e`)`	Binds itself to n_f and n_v in e
`(Let` n e_v e_f`)`	Binds e_v to n in e_f
`(Seq` e_a e_b`)`	Evaluates e_a then e_b
`(Ret` v`)`	Returns value v
`(Var` n`)`	Returns the value of variable n
`(Neg` e_i`)`	Negates e_i
`(Add` e_i e_j`)`	Adds e_i and e_j
`(Sub` e_i e_j`)`	Subtracts e_j from e_i
`(Mul` e_i e_j`)`	Multiplies e_i and e_j
`(Div` e_i e_j`)`	Divides e_i by e_j
`(Pair` e_a e_b`)`	Makes a pair with e_a and e_b
`(Fst` e`)`	Takes first value of pair e
`(Snd` e`)`	Takes second value of pair e
`(Left` e`)`	Makes a left variant with e
`(Right` e`)`	Makes a right variant with e
`(Match` e_v e_l e_r`)`	Conditionally calls e_l or e_r with the contents of e_v depending on its variant: left or right

Table 1: Base language terms. e stands for terms, v stands for values, n stands for names

For the base language, all reducible expressions are reduced to values of one of the following types: `IntVal` for integers, `PairVal` for pairs, `EitherVal` for left-right variants, `FuncVal` for functions and `RecFuncVal` for recursive functions. The language expects the appropriate types for each expression but does not enforce this, as type-checking fall outside of the scope of the project.

2.2 ACTOR-BASED LANGUAGE

The actor-based language makes use of the actor model of concurrency. In this model, concurrent communication occurs directly between actors, which can identify one another via their process id (*PIDs*). To recreate this model, it is necessary to be able to create new actors, send and receive messages, and discover an actor's own PIDs. To this end, four syntactic constructs, detailed in Table 2, were added to the base language.

In addition to these syntactic constructs, the actor-based language also sees PIDs utilized and communicated within and across processes, so a new `PidVal` was added to the list of possible values.

A small example program making use of the features of the language can be seen in Figure 4.

(Self)	Returns the actor's own PIDs
(Send e_v e_a)	Sends the value of e_v to actor e_a
(Receive)	Dequeues a message from the actor's own mailbox
(Spawn e)	Spawns a new actor with program e and returns that actor's PIDs

Table 2: Actor-based language exclusive terms.

```
(Let (
  (act_1) (Spawn (Receive))
  (Send ( (Ret (IntVal 7)) (Var act_1) ))
))
```

Figure 4: A simple actor-based program. This program spawns a new actor and sends it the integer value 7.

2.3 CHANNEL-BASED LANGUAGE

In the channel model, communication occurs via globally accessible mailboxes called channels. The channel model requires a means for processes to interact with said channels and a way for creating new channels and processes.

There's also a need to be able to address channels, so the value type `ChanIdVal` has been added. The necessary additions to the language's syntax can be seen in Table 3.

A small example program making use of the features of the language can be seen in Figure 5.

`(NewCh)`	Creates new channel, returns its ID
`(Give` e_c e_v`)`	Enqueues e_v in channel e_c
`(Take` e`)`	Dequeues a value from channel e
`(Fork` e`)`	Creates a new process evaluating e

Table 3: Channel-based language exclusive terms.

```
(Let (
  (ch) (NewCh)
  (Seq (
    (Give ( (Var ch) (Ret (IntVal 7)) ))
    (Fork ( (Take (Var ch)) ))
  ))
))
```

Figure 5: A simple channel-based program. This program creates a process and a channel to send it the integer value 7.

3

DEVELOPMENT

3.1 SEMANTICS IN SKEL

For a language interpreter to be generated by Necro, its semantics must be described in Skel, an explicitly typed, declarative language designed for describing skeletal semantics. There are two main top-level statements in Skel: types and values.

Type statements can be declarations, definitions, or aliases. Type declarations start with `type` and are followed by a type name; they do not list an implementation for the type. These declarations are interpreted as base-types when compiled and will be left unimplemented in the generated OCaml module so that the user can provide a suitable OCaml implementation for them. Type definitions describe automatically implemented program-types; they use the same keyword but are followed by `=` after the name, along with one or more constructors naming the possible variants for the type. Finally, with a slightly different syntax that uses `:=` instead of `=`, type aliases can be declared. Alias syntax can also be appropriated for defining named product types. Examples for each of these can be seen in Figure 6.

```
(* Base type *)
type base

(* Sum type *)
type sum =
| VarA
| VarB (base)

(* Type alias *)
type alias := base

(* Named product type *)
type prod := (base, sum)
```

Figure 6: Types in Skel

Values encompass both filters and skeletons. Value declarations that list no implementation in Skel are compiled as filters, and their implementation is expected to be provided in OCaml for the interpreter to function correctly. Value definitions, those that directly provide an implementation, are treated as skeletons and their code is automatically generated during compilation. Skeletons are limited to pattern-matching declarations that make use of filters and other skeletons, sequencing, and branching. For both filters and skeletons, a complete type signature must be explicitly provided. See Figure 7 for examples.

```
(* Filters *)
val filter_value: A
val filter_function: A -> B

(* Skeleton *)
val skel (x: X): Y =

  (* pattern matching *)
  let Var (v) = func_a (x) in

  (* sequencing *)
  func_b (v) ; func_c (v) ;

  (* branching *)
  branch
    test_a (v)
  or
    test_b (v)
  end
```

Figure 7: Filters and skeletons in Skel

As previously explained, skeletal semantics also require an interpretation to be complete. In the case of Necro, with OCaml as a target, this is provided by implementing an *interpretation monad* in OCaml after the semantics have been compiled. The interpretation monad is an OCaml module fulfilling a specialized monadic interface that determines how the interpreter evaluates branching and sequenced semantics. The generated code is designed in such a way that a trivial identity monad (one that does not modify its contents nor the functions it binds), behaves the same way a concrete interpretation of the semantics would.

Detailed information on using Skel with Necro can be seen in the work of Noizet and Schmitt (2022).

3.2 DESCRIBING THE SEMANTICS

The first step in the development of the language interpreters is the translation of the language specification into skeletal semantics. In the case of this project, the specifications for the languages were partly described as operational semantics by Fowler, Lindley, and Wadler (2016). Much of the unspecified aspects of the language are aspects belonging to the well-understood λ-calculus, so their specifications were retrieved from other sources (Nielson and Nielson, 2007; Pierce, 2002).

For translating operational semantics into Skel, some heuristics were developed. The case where semantic rules follow a simple structure—one consisting of various premises followed by a simple evaluation relation as consequence—can be translated in the manner shown in Figure 8. Translating axioms—semantic rules that have no premises—is also done by following this same heuristic.

$$\frac{\text{Premise A} \qquad \text{Premise B} \qquad \dots}{\text{Input term} \Rightarrow \text{Output term}}\ \textsc{Rule name}$$

(a) Operational semantics

```
val <Rule name> (<Input>) =
    let <Input term> = <Input> in
    <Premise A>
    <Premise B>
    ...
    <Output term>
```

(b) Skeletal semantics

Figure 8: Heuristic for translating from operational to skeletal semantics

In this heuristic, the skeleton has an initial pattern-matching expression unifying its input with the shape of the input term in the operational semantics. This expression ensures that the skeleton will only evaluate the appropriate terms, failing execution on any others. After that, for each premise in the operational semantics rule, a corresponding expression will be added to the skeleton in sequence. The type of expression will depend entirely on the specifics of each premise but, generally speaking, premises that assert that a term evaluates to some form will translate to a pattern-matching expression. Finally, after all the premise declarations are written in the skeleton, the output state is described as a final expression, which will then be returned by the skeleton.

Skeletons that apply to equal initial terms can be combined with the help of branching, to make for more succinct semantics. However, this is not strictly necessary for the correct evaluation of the language. This is not unlike factoring out statements from conditionals in other programming languages. An example of this can be seen in Figure 9.

These heuristics work well for rules that describe simple reductions or rewrite operations. Many other operations, however, are significantly more complex or have nuances that cannot be translated in such a straightforward manner. For example, semantic rules that rely on some sort of mapping will require those maps to be declared, implemented, passed as arguments across the entire rule hierarchy, and queried whenever used. Figure 10 shows a possible translation

```
val <Rule A> (<Input>) =
    let <Input term> = <Input> in
    <Expression A>

val <Rule B> (<Input>) =
    let <Input term> = <Input> in
    <Expression B>
```

(a) Separate rules applying to a common input term

```
val <Rule> (<Input>) =
    let <Input term> = <Input> in
    branch
        <Expression A>
    or
        <Expression B>
    end
```

(b) Rules after having been combined

Figure 9: Heuristic for combining skeletons

for a basic variable substitution rule. As can be observed, despite most implementation details being omitted from the skeletal semantics, the implementation of a seemingly simple rule is fairly complex. This complexity compounds when translating semantics dealing with real-world concerns: should the mappings be local or global? How should mappings be synchronized or maintained across processes or reductions? How should global or deferred declarations be handled?

The issue with variable mapping can be said to be part of a larger problem affecting the translation phase, that of the handling of mathematical notation and its definitions. The VAR rule described in Figure 10 appears simple to those familiar with the notation used, but that notation hides a significant amount of complexity that ultimately surfaces when the need to translate it to a universe that does not provide such abstractions arises. This is the case with Skel and Necro.

This ties in with the task of choosing which aspects of the language should be abstracted away and which should not be. Filters are powerful in that they can abstract away enormous portions of a language's semantics if so desired. However, the price for these abstractions is that their implementation must either be trusted blindly or be verified independently; the former compromising the safety of the implementation, while the latter requires a large investment of time and effort.

Due to these drawbacks, it is worth carefully considering whether a particular feature is worth abstracting. However, it is important to bear in mind that even though Skel is powerful enough to represent

$$\frac{x \mapsto v \in \Gamma}{\Gamma \vdash x \Rightarrow v}\ \textsc{Var}$$

(a) Operational semantics

```
(* Mapping definitions *)
type var_map (* ... *)
val query_var_map (* ... *)

(* Environment definition *)
type env := (..., variable_map, ...)

(* Rule *)
val <Var> (<Input>, <Env>) =
    let Var x = <Input> in
    let v = query_var_map (x) in
    Val v
```

(b) Skeletal semantics

Figure 10: Example of variable mapping

any computation that can be represented in other programming languages[1], filters provide the only means to achieve certain things: interfacing with foreign resources (such as operating system calls), using high performance, close-to-the-metal implementations (such as binary integer representations and operations), and, more humbly, reducing the effort required to implement specific functionality by leveraging existing libraries for the target platform.

These choices in the development of the semantics play a crucial role in the overall process of building a language interpreter using Necro. Not only do they form the foundations for the rest of the process, but they are also expensive to modify later on in development: the semantics need to be recompiled after modification, and the conflicting code needs to be merged and updated—a process prone to errors.

3.3 OCAML WORKFLOW

Once the semantics are complete and the OCaml code has been generated with Necro, four steps remain before a correctly-working interpreter has been completed:

- Choice of the interpretation monad
- Filter and base-type implementations
- Interpreter interface implementation

---

[1] This is made possible due to the typed λ-calculus embedded within Skel

- Testing

As explained previously, the choice of the interpretation monad determines how the semantics are interpreted when evaluating a program. The monad determines—in addition to the standard monadic behaviour of binding functions and wrapping and extracting values—how filters and skeletons are applied, how branching is executed, and how branch failures are handled. The combination of these functionalities can shape the type of interpretation that takes place while evaluating a language.

For this project, the *identity monad* was used, which does not modify values or functions. Its behaviour replicates a concrete interpretation of a semantics, as described by Bodin et al. (2019).

The implementation of filters is a more complex matter. For this project, filters were only used to abstract away maps, queues, strings, and integers. All of these implementations both require significant effort to be fully represented in Skel and have existing implementations in the target platform that can be trusted to be correct. Additionally, their use in the semantics is either wrapped in a layer of native abstractions—to ensure safe execution and reusability—or made to have a minimal footprint—to reduce the adverse effects of potential bugs stemming from their implementation.

It is also necessary to define interfaces and utilities by which the interpreter can be utilized. These are not limited to just a public API, but also include wrappers and other functionalities that can enable complex behaviour during execution. For example, semantics written in a small-step style—which is the case for the language interpreters developed in this project—require an external execution loop for repeatedly applying reduction rules to the program until it is fully reduced, or until it can't be reduced any longer. Another example is a REPL[2] for the language, which would similarly require a wrapper around the interpreter to enable user interactivity.

Finally, testing is done to ensure that all aspects of the implementation behave as expected. An iterative process of bug-fixing followed by more testing is also performed until the product is working as intended.

3.4 COMMON LANGUAGE DETAILS

The semantics for the base language, as described in Fowler, Lindley, and Wadler (2016), are written in a small-step style. Small-step semantics describe evaluation as an iterative series of reductions on a program. The iterative nature of small-step semantics allows for program execution to be paused or halted at arbitrary points. A behaviour that is useful for modelling concurrency: parallel processes can be paused, scheduled, and resumed as necessary.

---

[2] Also known as an *interactive toplevel* or *language shell*

Despite the usefulness of small-step semantics in describing concurrency, their implementation with Necro requires some additional functionality before the iterative reductions can fully evaluate a program. Namely, a means of repeatedly evaluating an expression and a means of halting once all concurrent expressions have been fully reduced.

Repeated evaluation is not difficult to implement. It requires a recursive function that receives and evaluates the output of the previous reduction as input. Halting poses a bigger challenge, particularly in the face of programs with recursive functions or other means of boundless looping, as it overlaps with the halting problem. This problem, in its general form, states that writing a program that can determine if an arbitrary program terminates is an unsolvable problem (Sipser, 1996). However, we can put in place some heuristics to terminate execution in specific, but common cases.

The first heuristic terminates execution once all processes have been reduced to a value. In other words, once all problems have returned a final result to their computations. In practical terms, an expression is considered fully reduced when the root node of its syntax tree is of type `Ret`.

The second heuristic attempts to expand the halting conditions to include the case when processes are waiting for a message that will never arrive. This case arises when a process receives messages in a loop. In such situations, whenever a program is waiting for a message[3], the process in question will be marked as waiting until a message arrives. If all processes in an active program are either in waiting or have been fully reduced, it means that no messages will be sent in the future and, thus, the waiting state will never be cleared. At that point, the interpreter will judge the program to have entered a non-reducible state and terminate.

Some of the features of the languages, such as recursive functions, or the implementation of mailboxes, require a shared state that is accessible at arbitrary points during the evaluation of expressions. To this end, it is necessary for all skeletons in the semantics to pass an argument, the *execution environment*, that stores this information. Both the actor- and the channel-based languages store some data in common in this structure: a table mapping recursive function names to the expressions that make up their bodies and a queue of executing processes storing the expressions being evaluated by all active processes.

---

[3] That is, whenever `Take` (for channels) or `Receive` (for actors) fails to reduce due to an empty mailbox

Each of the two languages also stores some additional information in their execution environments, as a means to implement their respective models of concurrency.

The queue stored in the environment is the key to the concurrent execution of processes in both languages. This queue is used for scheduling the evaluation of each expression. After the expression of a process has been reduced once, the expression in question is rescheduled to the end of the queue and a new process is prepared for reduction for the next pass of the interpreter. A diagram of this process can be seen in Figure 11. Effectively, this results in all processes being executed concurrently. The choice of a single reduction or pass of the interpreter for each process is arbitrary; concurrency is accepted to be indeterminate in terms of execution times, so, while a computation may differ when changing the scheduling configuration, the results would be just as valid.

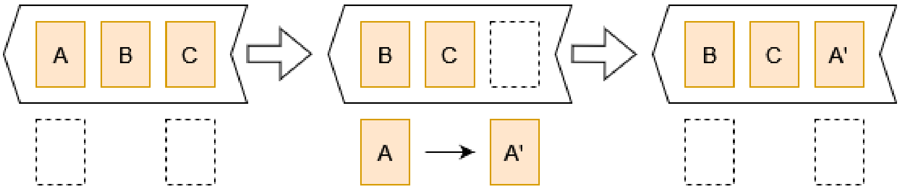


Figure 11: Diagram showcasing concurrent process scheduling and evaluation.

The execution environment can also be used for variable mapping. However, for these languages, direct substitution was chosen over variable mapping. This choice stays close to the description of the original semantics for the two languages and reduces the footprint that dictionaries, which are implemented via filters, have on the overall semantics.

As briefly mentioned before, recursive function names are an exception to this and do use an environment-stored dictionary instead of immediate substitution.

3.5 ACTORS IMPLEMENTATION

Given that actors communicate by means of their PIDs, the execution environment of the actor-based language has been extended to include process PIDs in the process scheduler, as well as a global PID-to-mailbox mapping that is accessed during communication operations. This also necessitates that some set-up procedures instantiate the initial actor, its PID, and its mailbox when the interpreter first opens a program.

Beyond that, and the unique semantics for the 4 additional expression constructors—seen in Table 2—the actor-based language requires no special considerations.

3.6 CHANNELS IMPLEMENTATION

As is the case for the actor-based language, the channel-based language modifies the evaluation environment to include information that is necessary for concurrent communication. In this case, a Channel ID-to-mailbox mapping is included. Unlike the actor-based language, no special initialization procedures are necessary, as the first channel needs to be explicitly created before being instantiated.

These mappings are interacted with by 3 of the 4 operations unique to the channel-based language, seen in Table 3; `Fork` requires no interaction with channels or mailboxes.

3.7 TRANSLATORS

The translators had been initially planned as stand-alone applications developed entirely in OCaml. During the development of the semantics for the two languages, it became apparent that skeletal semantics were flexible enough for the translation procedure to be almost fully described in Skel.

To achieve this, two distinct expression types were defined, one describing expressions in the actor-based language, and the other for expressions in the channel-based language. Skeletons then were typed to have one of the two expression types as input and the other as output, effectively taking expressions in language and producing expressions in the other.

As for value types across the two languages: because all values, barring strings and integers, are instantiated at run-time, both languages were defined within the scope of the translator semantics to use a common set of value types, which simplified the development process.

In terms of both architecture and semantics, the translators are simpler compared to the interpreters. The translation scheme, described by Fowler, Lindley, and Wadler (2016), uses big-step style semantics and, as such, does not require an external loop or special halting mechanisms for proper execution. Big-step rules are also less verbose to describe in skeletal semantics than their small-step counterparts, due to the recursive nature of skeletons.

The translator utilities each do have certain idiosyncrasies resulting from the nature of the translation schemes and the languages' semantics. In the case of actors-to-channels, an initial channel needs to be created at the root of the syntax tree. This channel serves to model the associated mailbox of the root actor, which is implicitly spawned on program execution and thus has no explicit syntax that can be translated.

In the case of channels-to-actors, actors are used to emulate the behaviour of channels. The channel-behaving actors establish a simple communications protocol that emulates the primitive communication operations available to actors: `Send` and `Receive`. The protocol allows an actor to receive and queue messages from other processes, and to relay these messages to other actors who request them.

To enable this functionality, it is necessary to establish a loop for processing requests. This loop is defined via a series of functions that are injected into the program via filters[4]. These functions make use of recursion, sum types, and product types, and are the reason why these constructs were included in the common core for the languages.

Manual testing was performed for both translators to review the correctness of the implementation of the translation schemes. Automated tests are a possibility for the future, to further increase the reliability of the tools, but their use was eschewed for this project due to time constraints. The non-deterministic nature of concurrent communications and the intrinsic differences in the management of values and program execution between the two language interpreters made it too expensive to set up appropriate testing facilities.

---

[4] Filters are the simplest way to inject them. These functions make use of string literals, which are not available at the Skel level; they are abstractions implemented in OCaml.

4

ARCHITECTURE AND RESULTS

The project consists of an OCaml library that provides four modules:

- `Actors`, for the actor-based language interpreter
- `Channels`, for the channel-based language interpreter
- `Act2Ch`, for the translator from the actor language to the channel language
- `Ch2Act`, for the translator in the opposite direction.

All four modules expose a similar API consisting of: type constructors for expressions and values, a function for parsing a textual representation of the language, a function for evaluating the syntax tree according to the function of the module, a function for serializing a syntax tree into a textual representation, and a function that chains the behaviour of these last three functions. This modular structure was chosen to allow for maximum flexibility for the user when making use of the library, while providing shortcuts for the most basic use-cases.

An architecture diagram of this can be seen in figure 12, and the typical execution process of the modules can be seen in figure 13, with a concrete example in figure 14.

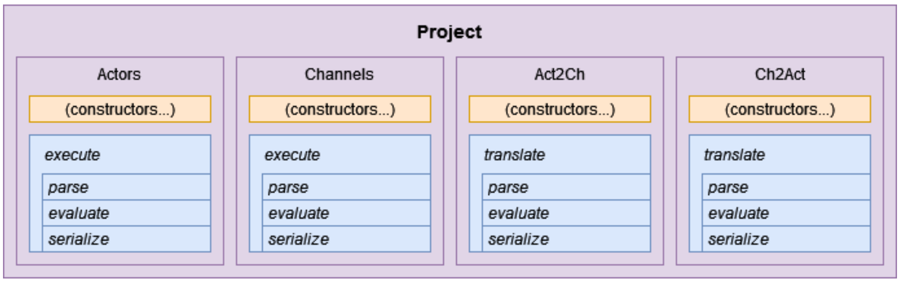

Project
Actors
(constructors...)
execute
parse
evaluate
serialize
Channels
(constructors...)
execute
parse
evaluate
serialize
Act2Ch
(constructors...)
translate
parse
evaluate
serialize
Ch2Act
(constructors...)
translate
parse
evaluate
serialize

Figure 12: Public project structure. Modules in purple, constructors in orange, functions in blue.

The constructors provided by each of the modules are not compatible with one another, as they are declared separately, so the parsing and serializing functions can be used along with a textual representation of the language as an exchange format across modules.

The project in question, along with more detailed documentation and usage instructions, is hosted in an online repository (Bartelsman, 2022).

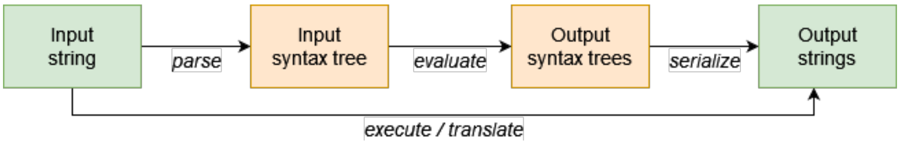
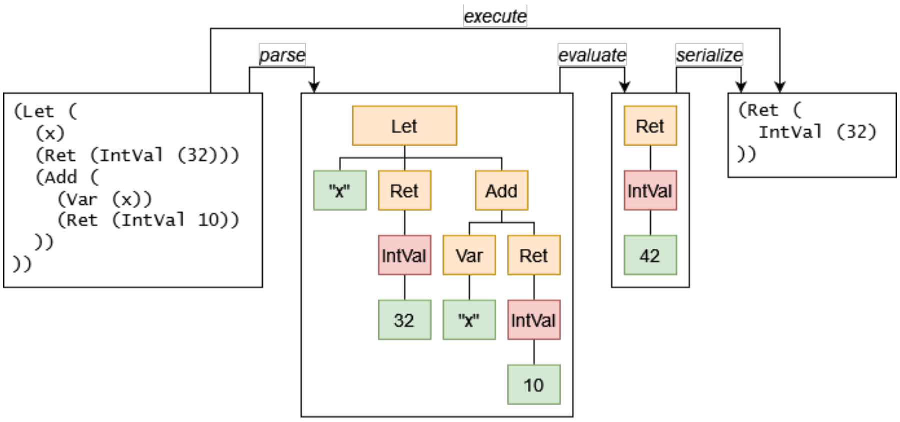

Input string
parse
Input syntax tree
evaluate
Output syntax trees
serialize
Output strings
execute / translate

Figure 13: Process diagram for the use of the four tools


execute
parse
evaluate
serialize
Let
"x"
Ret
Add
IntVal
Var
Ret
32
"x"
IntVal
10
Ret
IntVal
42

```
(Let (
  (x)
  (Ret (IntVal (32)))
  (Add (
    (Var (x))
    (Ret (IntVal 10))
  ))
))
```

```
(Ret (
  IntVal (32)
))
```

Figure 14: Example of the process and results for one of the interpreters. Orange are expression nodes, red are value nodes, green are values.

5

CONCLUSION

Skeletal semantics is a versatile and powerful tool. Combined with the automation that Necro provides, it makes it possible to significantly speed up the development of verified interpreters for programming languages. The abstraction mechanisms provided by filters and base types can be leveraged to speed up this process even more or to better allocate efforts into more significant aspects of programming language research. Filters permit powerful abstractions, and could even be used to nest entire interpreters within one another, easily extending the functionality of languages or allowing for more modular architectures in the development process. This would enable the rapid testing and analysis of languages with similar grammar and semantics.

In the realm of concurrency, skeletal semantics open the doors for rapid exploration of different models of message-passing concurrency, as evidenced by this very project. The ability to rigorously describe a particular concurrency model and then confidently generate a correct language interpreter provides ample opportunity for the practical analysis of concurrent software applications.

The flexibility of this semantics framework also allows for the abstraction of concurrent behaviour, permitting the use of different implementations, such as those offered by foreign platforms. This would allow for testing the behaviour of programs or algorithms, with certain properties guaranteed by the semantics, running on top of a variety of different real-life platforms and architectures.

However, using skeletal semantics is not without difficulty. Choosing what parts of a language should be abstracted can be a tricky subject without definite or correct answers. Especially so when a wrong assumption or semantic definition can result in having to discard and rewrite large portions of a code-base.

Mistakes in the skeletal semantics of a language pose a large enough roadblock to development that research into mitigating these risks could be of great benefit. Efforts to this end could focus on determining ways to structure the code bases for filters, base-types, and the interpreter itself so that changes to the generated code have as minimal an impact on development as possible.

Another promising path for research is investigating different ways to represent concurrent operations via skeletal semantics. This project utilizes a process queue that cycles processes after every reduction, but other, better or more flexible, approaches might also be viable.

Furthermore, skeletal semantics offer the possibility for diverse interpretations of a language via interpretation monads. Alternative interpretations could, then, become powerful tools for the analysis of languages implemented with the help of skeletal semantics. Verification for well-formed language structures is already possible thanks to a well-formedness interpretation. It might not be out of the realm of possibility that interpretations could be used to verify other aspects of a language, such as session types for concurrent communications.

6

BIBLIOGRAPHY

Agha, Gul (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press. ISBN: 978-0-262-01092-4.

Agha, Gul, I.A. Mason, S.F. Smith, and C.L. Talcott (1997). “A foundation for actor computation.” In: *Journal of Functional Programming* 7 (1), pp. 1–72. DOI: `10.1017/S095679689700261X`.

Bartelsman, Miguel (2022). *Actors and Channels*. `https://github.com/mbartelsman-rug/actors-and-channels`.

Bodin, Martin, Philippa Gardner, Thomas Jensen, and Alan Schmitt (2019). “Skeletal semantics and their interpretations.” In: *Proceedings of the ACM on Programming Languages* 3 (POPL). ISSN: 24751421. DOI: `10.1145/3290357`.

Courant, Nathanaël, Enzo Crance, and Alan Schmitt (June 2020). *Necro: Animating Skeletons*. Tech. rep. Inria Rennes - Bretagne Atlantique.

Durumeric, Zakir et al. (2014). “The Matter of Heartbleed.” In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC ’14. Vancouver, BC, Canada: Association for Computing Machinery, 475–488. ISBN: 9781450332132. DOI: `10.1145/2663716.2663755`. URL: `https://doi-org.proxy-ub.rug.nl/10.1145/2663716.2663755`.

Fowler, Simon, Sam Lindley, and Philip Wadler (2016). “Mixing Metaphors: Actors as Channels and Channels as Actors.” In: *CoRR* abs/1611.06276. arXiv: `1611.06276`. URL: `http://arxiv.org/abs/1611.06276`.

Khayam, Adam (2020). *JSkel : A JavaScript semantics in Skeletal*. URL: `https://gitlab.inria.fr/skeletons/jskel`.

Message Passing Concurrency (2010). URL: `https://wiki.c2.com/?MessagePassingConcurrency`.

Milner, Robin (1980). *A Calculus of Communicating Systems*. Ed. by Robin Milner. Vol. 92. Springer Berlin Heidelberg. ISBN: 978-3-540-10235-9. DOI: `10.1007/3-540-10235-3`.

Milner, Robin, Joachim Parrow, and David Walker (1992a). “A calculus of mobile processes, I.” In: *Information and Computation* 100 (1). ISSN: 10902651. DOI: `10.1016/0890-5401(92)90008-4`.

— (1992b). “A calculus of mobile processes, II.” In: *Information and Computation* 100 (1). ISSN: 10902651. DOI: `10.1016/0890-5401(92)90009-5`.

Moore, Gordon E. (1998). “Cramming more components onto integrated circuits.” In: *Proceedings of the IEEE* 86 (1). ISSN: 00189219. DOI: `10.1109/JPROC.1998.658762`.

Necro Library (2019). URL: `https://gitlab.inria.fr/skeletons/necro`.

Nielson, Hanne Riis and Flemming Nielson (2007). *Semantics with Applications: an Appetizer*.

Noizet, Louis and Alan Schmitt (2022). *Stating and Handling Semantics with Skel and Necro*. Tech. rep. Inria Rennes - Bretagne Atlantique. URL: `https://hal.inria.fr/hal-03543701v1`.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st. The MIT Press. ISBN: 0-262-16209-1.

Shared State Concurrency (2005). URL: `https://wiki.c2.com/?SharedStateConcurrency`.

Sipser, Michael (1996). "Introduction to the Theory of Computation." In: *ACM SIGACT News* 27 (1). ISSN: 0163-5700. DOI: `10.1145/230514.571645`.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede.

Final version as of July 22, 2022