Thesis for The Degree of Licentiate of Engineering

Choreographies and Cost Semantics for Reliable Communicating Systems

Alejandro Gómez-Londoño

Division of Formal Methods
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2020

Choreographies and Cost Semantics for Reliable Communicating Systems
ALEJANDRO GÓMEZ-LONDOÑO

Department of Computer Science & Engineering
Division of Formal Methods
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.

Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

Abstract

Communicating systems have become ubiquitous in today's society. Unfortunately, the complexity of their interactions makes them particularly prone to failures such as deadlocked states caused by misbehaving components, or memory exhaustion due to a surge in message traffic (malicious or not). These vulnerabilities constitute a real risk to users, with consequences ranging from minor inconveniences to the possibility of loss of life and capital. This thesis presents two results that aim to increase the reliability of communicating systems. First, we implement a choreography language which can by construction only describe systems that are deadlock-free. Second, we develop a cost semantics to prove programs free of out-of-memory errors. Both of these results are formalized in the HOL4 theorem prover and integrated with the CakeML verified stack.

Acknowledgments

I want to thank the many people that helped make this thesis possible. First, my supervisor Magnus for allowing me to be a part of the CakeML project, an endeavor that while challenging has been remarkably enriching thanks to his guidance and support. To Johannes, for patiently helping me navigate the often treacherous craft of theorem proving and inspiring me to prove forward even if it is all trivial at the QED. To my friends and colleagues, for turning hard days into cheerful ones, and always letting me know when there is cake upstairs. To my family, for encouraging me to be better with their nurturing love. And finally, to my dear partner Elisabet, thank you for being the person I could always count on, with whom I can always laugh, cry, or just be myself. *¡Te amo!*

Contents

Introduction

Computers programs are commonly used to implement communicating systems. From air traffic controls, to messaging and video conferencing apps, examples of such systems are abundant in today's society, and our reliance on their functionality for both critical and everyday tasks is increasing.

It is well known that computer programs can go wrong, and this is especially true when interaction between programs is involved, as is the case with communicating systems. The complexity that arises from coordinating multiple components can introduce errors that are unpredictable and hard to spot. Furthermore, the utility of these systems is based upon the reliability of their communication and disruptions can lead to the loss of capital, or in extreme cases, lives.

One way a communicating system can go wrong is when it reaches a state where multiple components wait on each other indefinitely without making any progress; this is referred to as a *deadlocked* state. Some of the problems that might lead to such a state are:

(P1) A communication protocol instructs some components to receive a message, but does not require another component to send one. Hence, stopping that part of the system from making any progress.

(P2) A program suddenly runs out of memory and stops sending messages leaving the rest of the systems waiting.

This thesis consists of two papers that tackle problems **P1** and **P2**. The first paper presents work on a choreography language to define communicating systems that, by construction, can not deadlock due to communication mismatches and therefore addresses **P1**. The second paper develops a cost semantics for CakeML [15] programs, enabling space-bound reasoning that can prevent occurrences of **P2**.

The following subsections introduce both papers in more detail, providing an overview of their results and how they deal with **P1** and **P2**. This chapter ends with a description of plans for future work, including work that aims to bring the results of these two papers together.

Choreographies

We consider communicating systems where components interact with each other only through a well-defined interface of *communication primitives*. This approach allows for well-defined boundaries between components, heterogeneous system implementations (e.g., across different devices, using multiple programming languages, or based on various frameworks), and forms the basis of many fundamental models of concurrency [1, 10, 11, 12, 13]. As a specific example of such a system, consider the process for making a purchasing decision between two buyers and a seller modeled as follows:

Example 1.

```
1.    BUYER1 asks SELLER for the price of an item
2.    SELLER gives the prices back to BUYER1
3.    BUYER1 shares the price with BUYER2
4.    BUYER2 tells BUYER1 if they decide to buy
5.    IF they decided to buy
5.1       BUYER1 gives its payment details to SELLER
5.2       SELLER responds with the receipt to BUYER1
6.    OTHERWISE nothing happens
```

In Example 1, a single buyer interacts with the seller; however, both buyers are involved in the decision to buy or not. An informal system definition like the one presented in Example 1 can be more concretely described using the following pseudo-language, where `A.x -> B.y` means component `A` sends value `x` to `B`, which binds it to its variable `y`.

Example 2.

```
BUYER1.item     → SELLER.item
SELLER.price    → BUYER1.price
BUYER1.price    → BUYER2.price
BUYER2.decision → BUYER1.decision
If BUYER1.decision ≡ "buy"
Then BUYER1.payment → SELLER.payment
     SELLER.receipt → BUYER1.receipt
```

This high-level view of a system's interactions is referred to as a *protocol*. Protocols describe how components "talk" to one another, and are meant as high-level blueprints for concrete system implementations. However, the details of internal computations are left unspecified—e.g., how `SELLER` fetches prices or how `BUYER2` decides when to buy.

To illustrate how one can go from the idea of a protocol to an implementation, consider the following pseudo-code implementation of `BUYER1` and `BUYER2` from our previous example:

Example 3.

BUYER1

```
send(SELLER,item)
price = receive(SELLER)
send(BUYER2,price)
decision = receive(BUYER2)
If decision ≡ "buy"
Then send(SELLER,payment)
     confirmation = receive(SELLER)
```

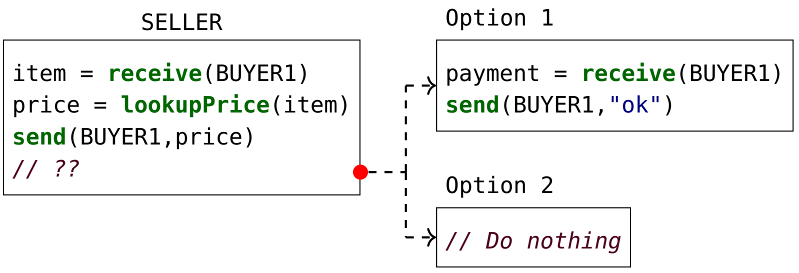
BUYER2

```
price = receive(BUYER1)
If price < 100
Then
  send(BUYER1,"buy")
Else
  send(BUYER1,"pass")
```

The structure of the code for each component follows from the protocol actions in which it is involved, either as a sender or as a receiver. This leaves only the internal computations unaccounted for. In Example 3, the expression `price < 100` was chosen as the criteria for whether to buy or not, but a different predicate could have been used while still adhering to the protocol.

It is however not a trivial task to correctly implement or even define a protocol; this is perhaps best illustrated by attempting to implement `SELLER` as follows:

Example 4.



SELLER

```
item = receive(BUYER1)
price = lookupPrice(item)
send(BUYER1,price)
// ??
```

Option 1

```
payment = receive(BUYER1)
send(BUYER1,"ok")
```

Option 2

```
// Do nothing
```

From the protocol defined in Example 1, the actions of `SELLER` are unambiguous up until step 4. However, once `BUYER1` and `BUYER2` make their decisions it is unclear what `SELLER` should do. In the first option (from Example 4) payment information is expected to arrive from `BUYER1`, but there is no guarantee of the purchasing decision being affirmative, in which case SELLER might never get a response and wait forever. Similarly, in the second option, if BUYER1 relays the payment information, it will never receive a confirmation from SELLER. This mismatch is due to an inconsistency in our original protocol definition, since SELLER does not have enough information at hand to follow the actions of the other components. These kinds of errors can be avoided by using a protocol language with well-defined semantics (see, e.g., Briais and Nestmann [6]) that only accepts "good" protocols, as opposed to the informal definitions used in Examples 1 and 2.

Unfortunately, the implementation of a consistent protocol can still go wrong if translation or computation errors are present [9]. Suppose, for example, the implementation of SELLER mistakenly performed all communications with BUYER2 instead of BUYER1 (a single character typo); in this scenario, all components will grind to a halt, and no further actions will take place in the system. Errors like these can be addressed by automating the majority of the protocol translation, leaving only the internal computations to be filled-in. Nevertheless, steps in-between communications can also fail in ways that affect the system as a whole. If a function (say `lookupPrice` in Example 4) does not terminate or returns an incorrect value (e.g., `-1`, `0`, `NaN`), this might lead to failures down the line which in turn leads to actions not being executed and the system reaching a failure state—possibly deadlock. It is a non-trivial task to prevent these types of failures, and it goes well beyond what informal protocols typically consider. Still, it is paramount for a reliable communicating system to mitigate such errors as much as possible.

Choreographies, as introduced by Carbone et al. [7, 8], are languages for defining communicating systems that address some of the issues mentioned before. First, a choreography defines a global protocol for all the components of the system alongside with their internal computations; this allows for entirely automated translations—by a process called *projection*—that by construction avoids mismatches between the specification and its implementation. Second, inconsistent protocols are ruled out by the semantics and a "projectability" criterion. These two mechanisms ensure all valid choreographies generate protocol-compliant code that does not get stuck and can always progress according to the protocol—this last property is known as deadlock-freedom.

An improved version of our running example can be defined using a choreography as follows:

Example 5.

```
Let item@BUYER1  = "Cake" in
BUYER1.item     → SELLER.item
Let price@SELLER = lookupPrice(item) in
SELLER.price    → BUYER1.price
BUYER1.price    → BUYER2.price
Let decision@BUYER2 =
   If price < 100
   Then "buy"
   Else "pass"
in
BUYER2.decision → BUYER1.decision
If BUYER1.decision ≡ "buy"
Then BUYER1 → SELLER[T]
     Let payment@BUYER1 = "<cc info>" in
     BUYER1.payment  → SELLER.payment
     Let receive@SELLER = "Purchase No: <XYZ>" in
     SELLER.receipt  → BUYER1.receipt
Else BUYER1 → SELLER[F]
```

Here `Let v@p = expr in` binds the result of the internal computation `expr` to variable `v`, in the context of component `p`. Moreover, the selection primitive `p →q[b]` communicates to component `q` that the branch `b` has been selected by `p`—a more formal definition can be found later in 1.2. By using choreographies to define our original example, we can provide a single global description that is concrete enough to generate implementations for all components directly. Furthermore, the ambiguity in the original version of the protocol (Example 2) is removed by communicating `BUYER1`'s branch choice to `SELLER` using a selection primitive, which ensures projectability.

The first paper in this thesis presents the implementation of a choreography language and, to the best of our knowledge, the first mechanised end-to-end proof of correctness of a projection function. The main contributions of this paper are:

- A formalization of a choreography language semantics with machine-checked proofs of confluence and deadlock-freedom.
- A projection function that leverages the characteristics of multiple intermediate languages to facilitate the machine-checked proof of semantics correspondence between choreographies and our target language, CakeML.

- A novel end-to-end result stating each component in the choreography will follow the global protocol as long as all other components are present and correctly perform their function. Additionally, this result extends to machine code thanks to the CakeML verified stack.

Statement of Contribution. For this paper, I contributed to the definition of the top-level choreography language and semantics, and was involved in the development of proofs for various properties (e.g., congruence correspondence, source level deadlock-freedom). Additionally, I was the main contributor to the implementation and verification of the projection function. This paper was in collaboration with Johannes Åman Pohjola, James Shaker, and Michael Norrish.

Cost Semantics

A program's space consumption is as relevant to its utility as the functional correctness of its implementation. Despite appearances to the contrary in high-level language semantics, programs only have a finite amount of memory available to them; the use of this resource has a direct impact on whether the programs will be able to execute their function correctly.

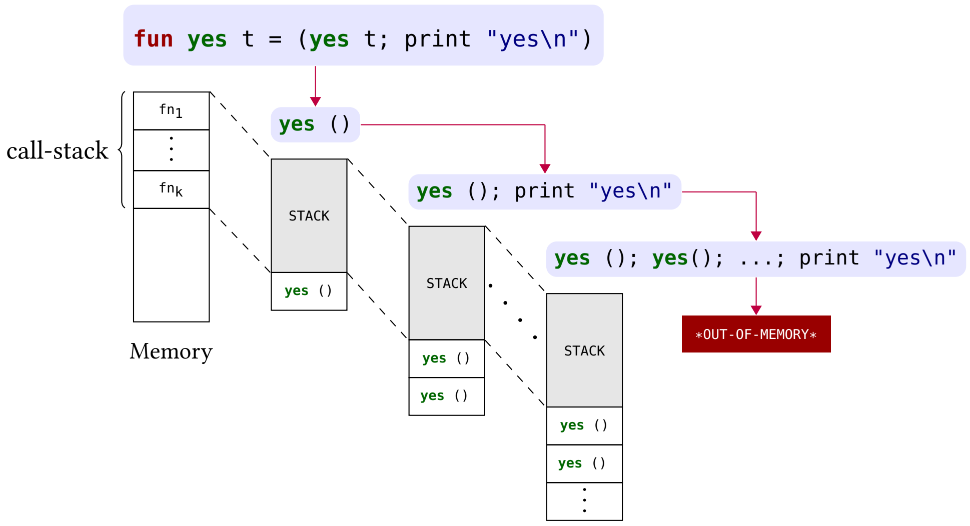
Consider the implementation of `yes`, a program whose expected behaviour is to prints the string "yes" forever. If, at any point during execution, memory is exhausted, then the program will exit and, as a result, will *not* print "yes" indefinitely, which is contrary to what the programmer intended. This is the reason why any correct implementation of `yes` must ensure that sufficient memory is available so it can indeed run indefinitely.

Example 6.

```
let
   fun yes t = (print "yes\n"; yes t)
in
   yes ()
end;
```

The code in Example 6 presents a valid implementation of `yes` in an SML-like language. However, a non-terminating recursive function like the one shown above could be a source of concern regarding its space use. Performing a function call often requires the program to store, to the call-stack, the current environment and return location in order to resume appropriately after the

Figure 1



called function returns. Therefore, if a program recursively calls a function without giving a result, as seems to be the case in Example 6, the call-stack's growth would eventually exhaust the memory. Thankfully, when a recursive call occurs at the end of a function, nothing needs to be stored to the call-stack, since there is nothing left to be done for the current function. Hence, the original caller function can be directly resumed—a technique called tail-recursion. The `yes` implementation shown above exhibits a tail-recursive structure; thus, memory is not exhausted despite the unbounded number of recursive calls. As a comparison, replacing line 2 (in Example 6) with `fun yes t = (yes t; print "yes\n")` fails to print "yes" and runs out of memory due to lack of tail-recursion (see Figure 1).

Answering the question whether a given program might runs out of memory during execution requires some compilation and runtime considerations. At first glance, the structure of a program provides good evidence of its memory consumption, but, other factors can sway the actual result. Consider the following two implementations of a program that computes the sum of function `foo` applied to numbers 0 to 10000000 (10^7) —where `foo` is any function from `int` to `int`.

Example 7.

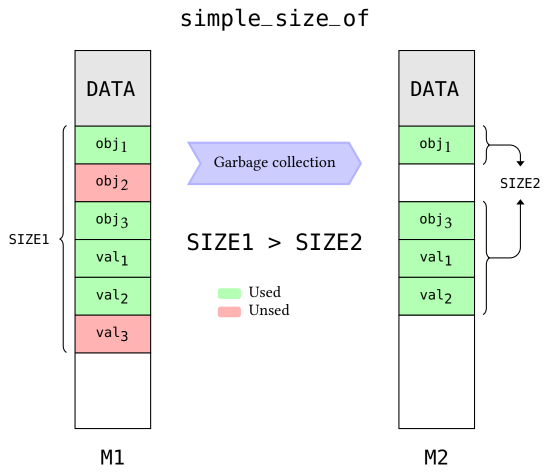
```
// Using a list
let
  fun bar1 0 = []
    | bar1 n = foo n :: bar1 (n - 1)
in foldl (op +) 0 (bar1 10000000)
end

// Using an accumulator
let
  fun bar2 0 x = x
    | bar2 n x = bar2 (n-1) (x + foo n)
in bar2 10000000 0
end
```

In the first implementation (`bar1`), the result is generated by traversing a list of 10000000 applications of `foo` and adding each element. In contrast, the second example (`bar2`) accumulates intermediate results on each `foo` application. Initially, it would appear that `bar1`'s use of a large list would result in a higher memory footprint than that of `bar2`, which only uses an accumulator argument; the intuition being that representing 10000000 elements ought to take more space than just one. However, looks can be deceiving, and while this observation holds for SML-like languages where arguments are fully evaluated—hence, represented in memory—before function calls, it does not hold for languages with on-demand argument consumption like Haskell. Furthermore, compiler optimizations could take `bar1`'s code and transform it into a structure similar to that of `bar2`, modifying its space consumption completely. Other aspects, like language design or underlying architecture, could further complicate reasoning about memory costs. Hence, intuition will only take us so far, and a formal approach might be more appropriate.

The *space cost* of a program is the highest memory consumption required during its execution. Therefore, if it exceeds the available space, the program will run out of memory. The formal measurement of a program's space consumption can be done through a cost function, which determines the amount of memory being used by the program at a given point. Hence, if one can show that this function never goes above some bound m, it follows that running the program with space greater or equal to m should not result in an out-of-memory error. A semantics with a concrete memory model can be used to perform such reasoning by implementing the corresponding cost function—in what is known as a *cost semantics* [2]—which in turn can be used to prove a concrete bound exists.

Figure 2

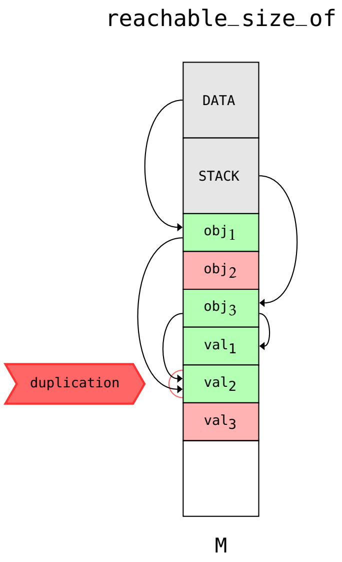


A cost function essentially measures the size of the data used by the program; thus, a simple implementation (`simple_size_of`) could just add the sizes of all objects currently in memory. However, in the presence of a garbage collector (GC), not all objects in memory are relevant for the measurement of space cost. Specifically, unused or unreachable objects can not exhaust a program's memory, as they are preemptively removed by a GC pass before an out-of-memory error can occur; thus, they are indistinguishable to free space from a space cost perspective. Therefore, when a GC is available, `simple_size_of`'s measurement is not well suited for space-bound reasoning, as it includes objects that could have been safely ignored.

`reachable_size_of` (Figure 3) improves on `simple_size_of` by only considering reachable objects; that is, objects that are being used by the program, and thus can be reached from global constants, local values of functions in the call-stack, or (recursively) other reachable objects. This approach often provides a better approximation than that of `simple_size_of`. Nonetheless, due to data aliasing—multiple pointers referring to the same data—objects stored in memory might be counted multiple times; thus, the traversal of reachable data needs to account for this to be effective.

Previous works on space cost semantics have targeted either languages without a GC [4], or only part of a larger compiler [5]. The second paper in this thesis presents a cost semantics that can be used to prove that a given CakeML program does not run out of memory, which, to the best of our knowledge, is the first time this result has been obtained for a garbage col-

Figure 3



lected language. The approach presented addresses common pitfalls in the following ways:

- The cost semantics is defined at an intermediate language of the CakeML compiler, which provides two main advantages. First, since most optimizations have already happened by that stage, the cost function does not need to account for optimizations. Second, the memory model is closer to the machine representation; thus allowing the cost semantics to be more concrete.

- The cost function provides a tight approximation of memory consumption by only considering reachable objects.

- Data aliasing is mitigated by marking every created value with a timestamp; this way, "seen" timestamps can be recorded as the reachable data is traversed and previously seen values can be ignored.

Statement of Contribution. For this paper, I worked on the addition of timestamps to the CakeML intermediate language DataLang, as well as, the definition and proof of soundness of a cost semantics for DataLang programs. Furthermore, I worked on the implementation and verification of two complete examples, the `yes` program, and a linear congruential generator. The other authors on this paper are Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus Myreen, and Yong Kiam Tan.

Future Work

The results of our first paper can be extended by increasing the expressiveness and usability of our choreography language by including convenience primitives and a more streamlined syntax. Likewise, the work described in our second paper can be extended by the development of proof automation to further facilitates space-bound reasoning for CakeML programs.

Finally, the two papers presented in this thesis aim to improve the level of reliability that can be achieved in a communicating system, and as such, despite their different approaches, they can be combined to provide even stronger assurances. Concretely, the deadlock-freedom guarantees provided by choreographies assume that each component is present and functions correctly. Space-bound reasoning can be used in conjunction with other features of the CakeML verified stack [3, 14]—the target language of our projection function—to guarantee that the projected components do not stop responding due to an out-of-memory error and thus can perform their task correctly.

Bibliography

[1] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990. ISBN 978-0-262-01092-4.

[2] R. M. Amadio, N. Ayache, F. Bobot, J. P. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. Sacerdoti Coen, I. Stark, and P. Tranquilli. Certified complexity (cerco). In U. Dal Lago and R. Peña, editors, *Foundational and Practical Aspects of Resource Analysis*, pages 1–18, Cham, 2014. Springer International Publishing.

[3] J. Åman Pohjola, H. Rostedt, and M. O. Myreen. Characteristic formulae for liveness properties of non-terminating cakeml programs. In *Interactive Theorem Proving (ITP)*. LIPICS, 2019.

[4] F. Besson, S. Blazy, and P. Wilke. A concrete memory model for compcert. In *Interactive Theorem Proving*, pages 67–83, Cham, 2015. Springer International Publishing.

[5] F. Besson, S. Blazy, and P. Wilke. Compcerts: A memory-aware verified c compiler using a pointer as integer semantics. *Journal of Automated Reasoning*, 63(2):369–392, Aug 2019.

[6] S. Briais and U. Nestmann. A formal semantics for protocol narrations. *Theor. Comput. Sci.*, 389(3):484–511, 2007. doi: 10.1016/j.tcs.2007.09.005. URL `https://doi.org/10.1016/j.tcs.2007.09.005`.

[7] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274, 2013. doi: 10.1145/2429069.2429101. URL `https://doi.org/10.1145/2429069.2429101`.

[8] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 2–17, 2007. doi: 10.1007/978-3-540-71316-6_2. URL `https://doi.org/10.1007/978-3-540-71316-6_2`.

[9] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014.

[10] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21 (8):666–677, 1978. doi: 10.1145/359576.359585. URL `https://doi.org/10.1145/359576.359585`.

[11] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. ISBN 3-540-10235-3. doi: 10.1007/3-540-10235-3. URL `https://doi.org/10.1007/3-540-10235-3`.

[12] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi: 10.1016/0890-5401(92)90008-4. URL `https://doi.org/10.1016/0890-5401(92)90008-4`.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992. doi: 10.1016/0890-5401(92)90009-5. URL `https://doi.org/10.1016/0890-5401(92)90009-5`.

[14] A. Sandberg Ericsson, M. O. Myreen, and J. Åman Pohjola. A verified generational garbage collector for CakeML. *J. Autom. Reasoning*, 63(2): 463–488, 2019. doi: 10.1007/s10817-018-9487-z.

[15] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019.

1

An end-to-end verified compiler for a choreography language

Johannes Åman Pohjola
Alejandro Gómez-Londoño
James Shaker
Michael Norrish

Abstract. Choreographies are a way of describing communicating systems as global programs, that offer several strong properties such as deadlock freedom, by construction. A choreographic program can be compiled into multiple endpoints, that when combined, exhibit the same behaviour as the original program. In this paper, we present a verified compiler for a choreography language. Its machine-checked, end-to-end proof of correctness ensures all generated endpoints adhere to the system description, preserving the top-level communication guarantees along the way. This work uses the verified CakeML compiler and HOL4 proof assistant, allowing for concrete executable implementations and statements of correctness at the machine code level for multiple architectures.

1.1 Introduction

A choreography is a global description of a communicating system, written in a style reminiscent of the `Alice` → `Bob` notation for protocol descriptions. Compared to the more traditional approach of writing separate programs for every `Alice` and `Bob` that participates, the choreographic approach has the advantage that it is impossible to write a program with a communication mismatch. In particular, deadlock freedom holds by construction. A procedure called *endpoint projection* compiles the choreography into separate programs for each endpoint, such that their parallel composition implements the global behaviour.

In this paper, we present a compiler for a choreography language with a machine-checked, end-to-end proof of correctness. That is, we create an environment based on the HOL4 interactive theorem prover [22] where programmers can write choreographies. The programmer can, at the proverbial push of a button, generate executable code for each endpoint, along with a proof that the endpoints are correctly compiled down to the machine-code level.

Pen-and-paper correctness proofs of endpoint projection in various settings abound [5, 8, 16, 21], but ours is, to the best of our knowledge, the first machine-checked proof. We believe ours is also the first where theory and practice coincides: the compiler we prove theorems about and the compiler that runs is the same object—there is no gap to mind.

Our compiler is structured into four phases. The first step is endpoint projection, where the global choreography is projected into a parallel composition of sequential programs implementing each endpoint, expressed in a process algebra we call ENDPOINT. Second, the ENDPOINT primitives for internal and external choice are encoded using send and receive. Third, ENDPOINT is compiled to a second process algebra, PAYLOAD. While messages in ENDPOINT can be arbitrarily large, messages in PAYLOAD have a fixed size. This step introduces a protocol that divides long messages into chunks, thus accounting for the fact that real communication protocols have bounds on message size, without burdening the application programmer with the details. The fourth and final compilation phase compiles Payload's endpoints to CakeML [14], a sequential, functional programming language with a verified compiler that guarantees semantics preservation down to the level of machine code.

Composing the compiler correctness results for each phase, we show that the deadlock freedom of the choreography language carries over to the compiler output: the generated CakeML code never diverges or aborts with a runtime error. By the CakeML compiler correctness theorem, neither does the machine code (unless it runs out of memory).

The CakeML code is parameterised on primitives for sending and receiving messages, and we assume that these actions have the same semantics as the corresponding Payload primitives. This means we are communication-backend agnostic: the same code is used irrespectively of whether the communication happens via (say) TCP/IP, MPI, or the operating system's API for inter-process communication (IPC). Like other choreography languages, our deadlock freedom guarantee depends on the rather strong assumptions implicit in the operational semantics: the backend stays live, and messages will never be lost in transit. In practice, our theorems are only as good as the backend's ability to abide by this.

Therefore, we implement a backend where communication is through IPC between user processes running on seL4 [13], a formally verified operating systems microkernel. Hence there is strong evidence, in the form of machine-checked proofs of functional correctness of the kernel [13] and the delivery guarantees of the component platform [7], that this backend is up to the task, even though we do not connect our proofs with theirs.

On a high level, our motivation for this work is to build a bridge between concurrency theory and systems verification. The former abounds with beautiful high-level specification formalisms and reasoning techniques for communicating systems; the latter with detailed proofs about low-level computing infrastructure such as compilers, language runtimes and operating systems. By joining these worlds, we can have high-level descriptions of communicating systems with guarantees that reach down to the low-level implementation.

All definitions and proofs in this paper are mechanised in HOL4 [22] and available online.[1]

[1] `https://github.com/CakeML/choreo/`

1.2 A choreography language

In this section we introduce a simple choreography language that can be used to concretely describe a communicating system in terms of the messages its nodes (known as endpoints) exchange. To create an intuition of how choreographies operate, we consider a common situation in component-based systems: a producer wishes to send a sequence $\widetilde{d}$ of messages to a consumer, but the consumer is only willing to receive messages of a certain form. Therefore, a filter that discards unwanted messages is inserted between them.

Example 8 (Message filter).

```
1.  for $d \in \widetilde{d}$ do
2.      let $v$@producer = $d$ in
3.      producer.$v$ -> filter.$temp$;
4.      let $test$@filter = test($temp$) in
5.      if $test$@filter then
6.          filter -> consumer[$\top$];
7.          filter.$temp$ -> consumer.$v$
8.      else filter -> consumer[$\bot$]
```

For each message $d \in \widetilde{d}$, the consumer binds d to its local variable v (line 2). The consumer then communicates the contents of v to the filter which stores it locally in *temp* (line 3). The filter computes test(*temp*) (line 4). If test(*temp*) is true (line 5), the filter informs the consumer that a message is coming (line 6), and finally forwards the contents of *temp* to the consumer (line 7). Otherwise, the filter informs the consumer that a message was dropped (line 8).

This example highlights two important features of choreographies. First, they capture both the concrete behaviour of its participants and a global view of the communication occurring between them. That is, interactions between endpoints are presented together with their internal computations—e.g., test. This is what allows individual endpoints to be translated into sequential programs. Second, communication mismatches are impossible by construction, since the interaction primitives captures both sending and receiving: if no message is forthcoming, the consumer will never be stuck waiting for one.

1.2.1 Syntax and semantics

Our language is very similar to Core Choreographies [5], but features arbitrary local computation and asynchronous communication. The main datatype under consideration is strings, or to be precise, finite sequences of bytes. Strings are used as endpoint names (p_i), variable names (v_i), and as the concrete data that gets bound to variables and transmitted between endpoints (d). The use of such a concrete representation keeps the language simple and allows for both low-level optimisations (see subsection 1.3.2) and the encoding of complex data through marshalling. Let ϵ denote the empty string, and + denote concatenation of strings. $d_{n,m}$ denotes the substring of d that is obtained by taking m of its characters, starting from the n:th character, and $d_{n\ldots}$ denotes the suffix of d obtained by dropping the first n characters. The booleans (ranged over by b) are written $\top$ and $\bot$. When we use booleans where strings are expected, we tacitly identify $\top$ with [0x01], and $\bot$ with [0x00]. We use a to range over the union of strings and booleans, and f to range over functions of type $\mathsf{string}^* \to \mathsf{string}$.

Definition 1 (Choreography syntax).

Choreographies, *ranged over by C, are inductively defined by the grammar:*

$$\begin{array}{lllllll} C & ::= & p_1.v_1 \mathrel{\text{->}} p_2.v_2; C & \textit{(com)} & p_1 \mathrel{\text{->}} p_2[b]; C & \textit{(sel)} \\ & & \textbf{if } v@p \textbf{ then } C_1 \textbf{ else } C_2 & \textit{(if)} & \textbf{let } v@p = f(\widetilde{v}) \textbf{ in } C & \textit{(let)} \\ & & \mathbf{0} & \textit{(nil)} & & \end{array}$$

The prefix (com) sends the data bound to variable v_1 at endpoint p_1 to endpoint p_2 which stores it in variable v_2, (sel) communicates the selection of a branch from p_1 to p_2, (if) branches over the value in variable v at process p, and (nil) denotes the empty choreography. Finally, (let) takes all values bound to the variables $\widetilde{v}$ at endpoint p and applies them as arguments to the function. The result is then stored in v. Note that we do not commit to any particular syntax for functions; rather, we take f to be a function in the meta-language in which the choreography language is defined. In our case, the meta-language is higher-order logic (HOL). Hence our syntax is only concerned with interaction and branching of endpoints, offloading computation to HOL. This flexibility is convenient for specifying open systems, or systems with legacy components: the internal behaviour of an endpoint that we have no control over can be modelled by functions that are non-computable, underspecified, or even completely uninterpreted, and the compiler can ignore such endpoints for code generation. For endpoints that we do intend to project, we require that the f's used in their let-bindings are "sufficiently code-like"—otherwise, code generation will fail. For example, functions with Hilbert choice, sets or quantifiers are not supported.

We do not (yet) have a looping construct—the **for** loop in Example 8 is syntactic sugar for the loop body iterated $|\widetilde{d}|$ times.

Unlike most presentations of choreography languages, we use a labelled structured operational semantics instead of a reduction semantics with auxiliary equivalence relations to reorder actions; this greatly simplifies mechanised inductive proofs.

Definition 2 (Labels).

Labels*, ranged over by* α, β*, are defined as follows:*

$$\begin{array}{llllll} \alpha, \beta & ::= & p_1.v_1 \mathbin{\texttt{->}} p_2.v_2 & \textit{(lcom)} & p_1 \mathbin{\texttt{->}} p_2[b] & \textit{(lsel)} \\ & & \textbf{let } v@p = \widetilde{v} & \textit{(llet)} & \tau_p & \textit{(ltau)} \end{array}$$

The operational semantics is inductively defined by the rules in Tables 1.1–1.3. Transitions are labelled—using Definition 2— to indicate both the action being performed (upper α), and the trace (lower l) of deferred asynchronous actions α needs to perform. We will explain the latter mechanism later in this section. We refer to both labels and prefixes as actions, since they directly correspond to all operations that can be performed in the language. A *store* (s) is a partial function $\mathsf{string} \times \mathsf{string} \hookrightarrow \mathsf{string}$ representing a global view of the endpoints' variable binding environment: $s(p, v)$, if defined, denotes the value bound to v in p's binding environment. We opted for an explicit binding environment instead of substitution-based semantics to facilitate the proof effort and remain close to the the target language (see section 1.5).

Definition 3 (Label equivalence). Label equivalence*, written $\simeq$, is the smallest equivalence on sequences of labels that is prefix-closed, suffix-closed, and satisfies the following rule (permitting exchange of labels mentioning disjoint processes):*

$$\frac{\mathrm{fp}(\alpha) \cap \mathrm{fp}(\beta) = \emptyset}{[\alpha, \beta] \simeq [\beta, \alpha]}$$

The first set of rules, given in Table 1.1, deal with the behaviour of the syntactic constructions introduced in Definition 1, where all rules follow the intuition previously presented.

Our language uses non-blocking, asynchronous communication. Hence, a sender process participating asynchronously in an interaction should be able

$$\textsc{Com}\ \frac{s(p_1, v_1) = d \qquad p_1 \neq p_2}{s \triangleright p_1.v_1 \texttt{->} p_2.v_2; C \xrightarrow[\epsilon]{p_1.v_1 \Rightarrow p_2.v_2} s[(p_2, v_2) := d] \triangleright C}$$

$$\textsc{Sel}\ \frac{p_1 \neq p_2}{s \triangleright p_1 \texttt{->} p_2[b]; C \xrightarrow[\epsilon]{p_1 \Rightarrow p_2[b]} s \triangleright C}$$

$$\textsc{Let}\ \frac{s(p, \widetilde{v}) = \widetilde{d}}{s \triangleright \textbf{let}\ v@p = \widetilde{v}\ \textbf{in}\ bp_2; C \xrightarrow[\epsilon]{\textbf{let}\ v@p=\widetilde{v}} s[\widetilde{(p, v)} := \widetilde{d}] \triangleright C}$$

$$\textsc{If}\ \frac{(s(p, v) = \top \land i = 1) \lor (s(p, v) \neq \top \land i = 2)}{s \triangleright \textbf{if}\ v@p\ \textbf{then}\ C_1\ \textbf{else}\ C_2 \xrightarrow[\epsilon]{\tau_p} s \triangleright C_i}$$

Table 1.1: Choreography semantics, I: behavioural rules

to perform further actions before the message has arrived at the receiver. Table 1.2 captures this behaviour by allowing any action α to occur before other interactions, provided only the sender process is present in α. A trace of every interaction being thus deferred is kept to ensure the consistency between asynchrony and concurrency rules, specifically (If-S). Here $\text{fp}(\alpha)$ denotes the endpoint names occurring in α, and $\text{wv}(\alpha)$ denotes the set of variables written to by α.

Finally, concurrency is handled by swapping rules (see Table 1.3) allowing the actions of disjoint sets of processes to be performed in any order regardless of their location in the choreography's structure. Rules (Com-S), (Sel-S), and (Let-S) apply the inner action α over their corresponding outer actions as long as the processes involved in each are disjoint. Moreover, (If-S) allows action α, which does not refer to process p, to occur on both branches of an **if** statement, with the added restriction that the traces l and l' of asynchronous deferrals be equivalent under Definition 3. This constraint guarantees that regardless of the choice of branch, the asynchronous actions that need to be deferred in order to perform α are the same for each of the processes involved, implying that α is independent of the branching in p.

We prove that the resulting semantics is locally confluent, and at least as expressive as a similar semantics with structural congruence instead of swap-

$$\textsc{Com-A}\ \frac{s \triangleright C \xrightarrow[l]{\alpha} s' \triangleright C' \qquad p_1 \in \mathrm{fp}(\alpha) \qquad p_2 \notin \mathrm{fp}(\alpha) \qquad \mathrm{wv}(\alpha) \neq (v_1, p_1)}{s \triangleright p_1.v_1 \text{ -> } p_2.v_2; C \xrightarrow[(p_1.v_1 \text{->} p_2.v_2)::l]{\alpha} s' \triangleright p_1.v_1 \text{ -> } p_2.v_2; C'}$$

$$\textsc{Sel-A}\ \frac{s \triangleright C \xrightarrow[l]{\alpha} s' \triangleright C' \qquad p_1 \in \mathrm{fp}(\alpha) \qquad p_2 \notin \mathrm{fp}(\alpha)}{s \triangleright p_1 \text{ -> } p_2[b]; C \xrightarrow[(p_1 \text{->} p_2[b])::l]{\alpha} s' \triangleright p_1 \text{ -> } p_2[b]; C'}$$

Table 1.2: Choreography semantics, II: asynchrony rules

ping rules. Additionally, to facilitate some results, a functional big-step version [19] of the semantics was developed and proven equivalent.

1.3 Intermediate languages

This section introduces the intermediate languages Endpoint and Payload used by our compiler. They are process algebras designed to make for convenient targets for endpoint projection, and also convenient source languages for CakeML code extraction. Therefore, they inherit many design decisions from our choreography language: we eschew named channels in favour of explicit point-to-point communication, and the only kind of data passed is strings. Because the intent is to generate verified code per-endpoint in a sequential language, we adopt a two-layer syntax: the endpoint layer is purely sequential and represents the code that gets executed in a single endpoint, and the *network* layer is a parallel composition of endpoints, each with its own name, queue and binding environment.

1.3.1 Endpoint: syntax and semantics

A *queue*, ranged over by q, is a function $\mathsf{string} \to \mathsf{string}^*$. The intuition is that $q(p)$ denotes the sequence of messages, from first to last, received from p but not yet processed. We let $q + (p, a)$ denote q with a appended to the end of $q(p)$, and $q - p$ denote q with the first element of $q(p)$ removed; if $q(p)$ is empty, $q - p$ is undefined.

$$\textsc{If-S}\ \frac{s \triangleright C_1 \xrightarrow[l]{\alpha} s' \triangleright C_1' \qquad s \triangleright C_2 \xrightarrow[l']{\alpha} s' \triangleright C_2' \qquad l \simeq l' \qquad p \notin \mathrm{fp}(\alpha)}{s \triangleright \textbf{if } v@p \textbf{ then } C_1 \textbf{ else } C_2 \xrightarrow[l]{\tau_p} s' \triangleright \textbf{if } v@p \textbf{ then } C_1' \textbf{ else } C_2'}$$

$$\textsc{Com-S}\ \frac{s \triangleright C \xrightarrow[l]{\alpha} s' \triangleright C' \qquad p_1, p_2 \notin \mathrm{fp}(\alpha)}{s \triangleright p_1.v_1 \to p_2.v_2; C \xrightarrow[l]{\alpha} s' \triangleright p_1.v_1 \to p_2.v_2; C'}$$

$$\textsc{Sel-S}\ \frac{s \triangleright C \xrightarrow[l]{\alpha} s' \triangleright C' \qquad p_1, p_2 \notin \mathrm{fp}(\alpha)}{s \triangleright p_1 \to p_2[b]; C \xrightarrow[l]{\alpha} s' \triangleright p_1 \to p_2[b]; C'}$$

$$\textsc{Let-S}\ \frac{s \triangleright C \xrightarrow[l]{\alpha} s' \triangleright C' \qquad p \notin \mathrm{fp}(\alpha)}{s \triangleright \textbf{let } v@p = f(\widetilde{v}) \textbf{ in } C \xrightarrow[l]{\alpha} s' \triangleright \textbf{let } v@p = f(\widetilde{v}) \textbf{ in } C'}$$

Table 1.3: Choreography semantics, III: swapping rules

An *environment e* is a partial function from variable names to values.

Definition 4 (Endpoint syntax).

$$\begin{array}{rcll}
P, Q & := & \overline{p}\, v.P & \textit{(output)} \\
 & & p(v).P & \textit{(input)} \\
 & & p \oplus b.P & \textit{(internal choice)} \\
 & & p\,\&\{\top : P, \bot : Q\} & \textit{(external choice)} \\
 & & \textbf{if } v \textbf{ then } P \textbf{ else } Q & \textit{(if)} \\
 & & \textbf{let } v = f(\widetilde{v}) \textbf{ in } P & \textit{(let)} \\
 & & \mathbf{0} & \textit{(nil)}
\end{array}$$

$$\begin{array}{rcll}
N & := & N_1 \mid N_2 & \textit{(parallel)} \\
 & & (p, e, q) \triangleright P & \textit{(endpoint)} \\
 & & \mathbf{0} & \textit{(nil)}
\end{array}$$

Definition 5 (Endpoint labels). *The labels of* Endpoint *are of kind*

$$p_1 \rightarrow p_2 : d \text{ (send from } p_1 \text{ to } p_2\text{)} \qquad p_2 \leftarrow p_1 : d \text{ (receive)} \qquad \tau \text{ (internal)}$$

Definition 6 (Endpoint semantics). *The semantics of* Endpoint *is inductively defined by the rules in Table 1.4.*

The If and Let rules are the obvious counterparts to the corresponding rules of the choreography language, and the Com and Par rules are standard. $\overline{p}\, v.P$ represents an endpoint ready to send the contents of variable v to p, using the Send rule; the Enqueue rule allows a message thus sent to arrive in p's queue. $p(v).P$ denotes a process ready to dequeue a message from its queue originating from p, and bind the contents of the message to the variable v (Dequeue); if there is no message from p, the endpoint waits until one arrives before acting. Similarly, $p \oplus b.P$ represents an endpoint ready to tell process p that it has made a choice, and chosen the b-branch. The corresponding IntChoice rule interacts with Enqueue to add the choice to b's message queue. $p \,\&\{\top : P, \bot : Q\}$ represents a process waiting for p to communicate its choice of branch. If it finds a $\top$ from p in the queue, it proceeds as P (ExtChoice-L); if it finds something else from p, it proceeds as Q.

1.3.2 Payload: syntax and semantics

Payload is parameterised by a *payload size* $\sigma > 0$. Unlike in previous languages where messages can have arbitrary size, here every message in transit must have size exactly $\sigma + 1$ bytes. Messages that are too long are divided into smaller chunks before being transmitted, and messages that are too short are padded; the extra byte in the message size is for encoding the bookkeeping necessary to realise this. In particular, we must keep track of whether a given chunk is the final part of a message, or whether it will be continued in future messages. This is handled by functions $\mathsf{pad}, \mathsf{unpad}, \mathsf{final}, \mathsf{intermediate}$ that satisfy the following laws:

$$|\mathsf{pad}(d)| = \sigma + 1 \qquad \mathsf{unpad}(\mathsf{pad}(d)) = d_{0,\sigma} \qquad |d| \le \sigma \Leftrightarrow \mathsf{final}(\mathsf{pad}(d))$$

$$|d| \ge \sigma \Leftrightarrow \mathsf{intermediate}(\mathsf{pad}(d))$$

Our compiler fixes particular implementations that satisfy the above laws; however, the details are unimportant, and we believe the compiler could be made parametric on these functions without too much difficulty, thus allowing different communication backends to use different message encoding schemes.

Definition 7 (Payload syntax)**.** *The syntax of* Payload *is obtained by removing internal and external choice from* Endpoint*, and replacing output and input with:*

$$\overline{p}\, v_n.P \quad \textit{(output)} \qquad\qquad p(v)\langle d\rangle.P \quad \textit{(input)}$$

The intent is that the prefixes record how far along in a transmission we are. Hence the prefix of $\overline{p}\, v_n.P$ will send the value of v to p, starting from the n-th byte, divided into as many chunks as necessary. Similarly $p(v)\langle d\rangle.P$ will receive chunks from p, recording every intermediate chunk in the temporary storage location d. When a final chunk arrives, the concatenation of all such chunks is bound to the variable d.

Definition 8 (Payload semantics)**.** *The semantics of* Payload *is obtained by replacing the* Send*,* Dequeue*,* IntChoice*, and* ExtChoice *rules of Table 1.4 with the rules in Table 1.5.*

1.4 Endpoint projection

In this section, we describe the structure of our compiler and its correctness proofs.

1.4.1 Phase I: endpoint projection

The main complication when defining endpoint projection is how to handle **if** statements, which are not always projectable. For an example, consider the choreography

$$\textbf{if } v\text{@Alice } \textbf{then } \text{Bob}.v \rightarrow \text{Alice}.v \textbf{ else } \text{Alice}.v \rightarrow \text{Bob}.v$$

where Alice makes an internal choice, and depending on the result, either Alice sends a message to Bob, or vice versa. How does Bob know whether to send or receive?

It is necessary to construct a projectability criterion that rules out such degenerate cases from consideration. Our criterion is, intuitively: whenever Alice chooses an **if** branch, every other endpoint whose projection depends on the choice must immediately be told which branch was chosen. Hence, the example above can be made projectable by adding selections as follows:

$$\begin{array}{l}\textbf{if } v@\text{Alice}\\ \textbf{then } \text{Alice} \texttt{->} \text{Bob}[\top];\\ \qquad \text{Bob}.v \texttt{->} \text{Alice}.v\\ \textbf{else } \text{Alice} \texttt{->} \text{Bob}[\bot];\\ \qquad \text{Alice}.v \texttt{->} \text{Bob}.v\end{array}$$

To formalise this criterion, we use the auxiliary function sp to split off initial selections pertaining to a pair of endpoints and check which branch was chosen.

Definition 9 (Split selections). *The partial function* sp *is inductively defined as follows (in all other cases,* sp *is undefined)*

$$\mathsf{sp}_{p_1,p_2}(p_3 \texttt{->} p_4[b];C) = \begin{cases}(b,C) & \textit{if } p_1 = p_3 \textit{ and } p_2 = p_4\\ \mathsf{sp}_{p_1,p_2}(C) & \textit{if } p_1 = p_3 \textit{ and } p_2 \neq p_4\\ \text{undefined} & \textit{otherwise}\end{cases}$$

Rather than define projection and projectability separately, we define a single function pr that given an endpoint name and a choreography returns both a boolean (its projectability), and an endpoint (its projection). We overload the Endpoint operators by letting

$$\overline{p}\,v.(b,C) = (b,\overline{p}\,v.C) \qquad p\,\&\{\top:(b_1,P),\bot:(b_2,Q)\} = (b_1 \wedge b_2, p\,\&\{\top:P,\bot:Q\})$$

and similarly for the other operators.

Definition 10 (Projection and projectability). *A choreography C is* projectable *if*

$$\forall p \in \mathsf{procs}(C).\ \ \pi_1(\mathsf{pr}_p(C)) = \top$$

The projection *of the endpoints $\widetilde{p}$ from a choreography C with state s is defined as*

$$[\![s \rhd C]\!]_{\mathsf{E}}^{\widetilde{p}} = \Pi_{p_i \in \widetilde{p}}.\ (p_i, s\downarrow_{p_i}, \epsilon) \rhd \pi_2(\mathsf{pr}_{p_i}(C))$$

where Π denotes iterated parallel composition, $s\downarrow_p$ denotes $\lambda v.s(p,v)$, and pr *is defined inductively by the equations in Table 1.6. We write $[\![s \rhd C]\!]_{\mathsf{E}}$ to abbreviate $[\![s \rhd C]\!]_{\mathsf{E}}^{\mathsf{procs}(C)}$.*

1.4.2 Phase II: remove choice

In this compilation phase, we implement Endpoint's internal choice primitives using send and receive actions. This serves two purposes: first, it simplifies the correctness proofs for later compilation phases. Second, it simplifies the implementation of a communication backend, which only needs to implement two primitives (send and receive) for message-passing.

Definition 11. *The compilation function $[\![\cdot]\!]_{\mathsf{C}}$ is homomorphic on all operators except internal and external choice, where it is defined as follows:*

$$[\![p \oplus b.P]\!]_{\mathsf{C}} = \mathbf{let}\ v = (\lambda x.b)\epsilon\ \mathbf{in}\ \overline{p}\, v.[\![P]\!]_{\mathsf{C}} \quad \textit{where } v \notin \mathrm{fv}(P) \qquad [\![p \,\&\{\top : P, \bot : Q\}]\!]_{\mathsf{C}} = p(v).(\mathbf{if}\ v\ \mathbf{then}\ [\![P]\!]_{\mathsf{C}}\ \mathbf{else}\ [\![Q]\!]_{\mathsf{C}}) \quad \textit{where } v \notin \mathrm{fv}(P, Q)$$

1.4.3 Phase III: Endpoint to Payload

Definition 12. *The compilation function $[\![\cdot]\!]_{\mathsf{P}}$ is homomorphic on all operators except: internal and external choice, where it is undefined; and input and output, where it is*

$$[\![p(v).P]\!]_{\mathsf{P}} = p(v_0)\langle\epsilon\rangle.[\![P]\!]_{\mathsf{P}} \qquad [\![\overline{p}\, v.P]\!]_{\mathsf{P}} = \overline{p}\, v_0.[\![P]\!]_{\mathsf{P}}$$

1.4.4 Compiler correctness

Let $[\![\cdot]\!]^{\widetilde{p}}$ denote the composition $[\![\cdot]\!]^{\widetilde{p}}_{\mathsf{E}} \circ [\![\cdot]\!]_{\mathsf{C}} \circ [\![\cdot]\!]_{\mathsf{P}}$ and let $[\![C]\!] = [\![C]\!]^{\mathsf{procs}(C)}$. We prove that this composite compiler satisfies weak operational correspondence up-to strong bisimilarity:

Theorem 1.
If c is a projectable choreography and $\mathrm{fv}(c) \subseteq \mathrm{dom}(s)$, then

1. *(Operational completeness) If $s \rhd C \implies s' \rhd C'$ then there exists s'', C'', N such that $s' \rhd C' \implies s'' \rhd C''$ and $[\![s \rhd C]\!] \implies N$ and $N \stackrel{\sim}{\cdot} [\![s'' \rhd C'']\!]^{\mathsf{procs}(C)}$*

2. *(Operational soundness) If $[\![s \rhd C]\!] \implies N$ then there exists s', C', N' such that $N \implies N'$ and $s \rhd C \implies s' \rhd C'$ and $N' \stackrel{\sim}{\cdot} [\![s' \rhd C']\!]^{\mathsf{procs}(C)}$*

Here $\Longrightarrow$ over networks denotes $\xrightarrow{\tau}^{\star}$, and $\Longrightarrow$ over choreographies denotes $(\bigcup_{a,l} \xrightarrow[l]{a})^{\star}$. The catch-up transition for operational completeness is because projectability is not, in general, preserved by reduction. However, for any non-projectable choreography C' reachable from a projectable choreography C, C' can always reduce to a projectable choreography. The only use of bisimilarity is to clean up the temporary variables introduced in Phase II.

The proof follows the structure of the compiler: we prove operational correspondence separately for each compilation phase. This leads to some duplication of effort, with routine proofs of similar lemmas duplicated for different intermediate languages and compiler phases. Nonetheless, it helps tremendously in making the compiler tractable for mechanised proof by allowing us to focus on one problem at a time.

The challenging direction is operational soundness: completeness is usually by induction on the length of the reduction sequence, but soundness generally requires an invariant to characterise the set of intermediate states the target language term might reach.

Our proof for Phase II uses an invariant, in the form of a binary relation $\mathcal{R}$ that characterises which source terms correspond to which target terms. It satisfies $N \mathrel{\mathcal{R}} [\![N]\!]_{\mathsf{C}}$, but also relates source terms to intermediate states in which some action has been partially evaluated. The key technical lemma states that if $N \mathrel{\mathcal{R}} N'$ and $N' \xrightarrow{\tau} N''$, then there exists N''' such that $N \Longrightarrow N'''$ and $N''' \mathrel{\mathcal{R}} N''$. Moreover, since this N'' may also be an intermediate state, we must prove that it can always reduce to a state that is within the range of $[\![\cdot]\!]_{\mathsf{C}}$, which is by induction on a metric over $\mathcal{R}$-related pairs.

This rather heavyweight approach is tractable for Phase II, requiring some 1500 lines of proof script. We tried the same approach for Phase III; this led to a world of pain, and we never finished the proof. Unlike Phase II, where each source-language transition results in just one intermediate state that can be described locally for a single endpoint, the invariant for Phase III is not as compositional: if I have intermediate messages from you in my queue, but no final message, you must be in the process of sending something whose beginning corresponds to what I have, and this partial transmission may be interleaved in arbitrary ways with other partial transmissions. This complicated relation resulted in a combinatorial explosion of the number of proof cases.

To make this proof tractable, we use a technique based on *inert reduction*, first conceived by van Glabbeek to study encodings from the synchronous to the asynchronous π-calculus [27]. Intuitively, an inert reduction is one that performs a bookkeeping step without committing to a branch. We say that $N \implies N'$ is *inert* if for every $N'' \neq N'$ such that $N \implies N''$, there is an N''' such that $N \implies N'''$. van Glabbeek's insight is that for translations that only need inert catch-up transitions, operational soundness can indeed be proven by induction on the length of the reduction sequence, and with no need to invent an invariant: since inertness is a form of confluence, it suffices to consider just one interleaving of the intermediate steps, namely the one that mimics one source-language step at a time in the most direct manner possible. This makes the proofs of Phases I and III tractable: since all our intermediate languages are confluent, it follows trivially that every catch-up transition is inert. The resulting proof is about as long as for Phase II, with roughly half the effort going into proving confluence.

Confluence also plays a major role in the operational completeness proofs for Phase I. The asynchrony and swapping rules of Table 1.2–1.3, which are otherwise a pain point, play no role in these proofs. This is justified because any reduction involving them has a common successor with a reduction that only uses the rules from Table 1.1. This results in a simpler proof than e.g. Montesi [16, Appendix C]; his choreography language is also confluent, yet his proof makes no use of this fact and includes cases for the swapping and asynchrony rules.

1.5 Compilation into CakeML

CakeML [14] is an impure, sequential, functional programming language similar to Standard ML. Its most notable feature is a compiler correctness proof in HOL4 that extends down to the machine code level for mainstream architectures such as x86-64 and ARM [23]. Interaction with the outside world is supported by a foreign function interface (FFI), which allow calls to arbitrary external code. We will assume two foreign functions, `send` and `receive`, that support communication with the other endpoints. Compilation to CakeML consists of two parts: the static part, which is verified once and for all, and the dynamic part, which is proof-producing and generates code for the functions used in source-language let-bindings.

1.5.1 Static compiler

The static compilation is performed by the function $[\![\cdot]\!]_{\mathsf{ML}}$ which maps Payload endpoints to CakeML expressions. Its full definition is too big to fit here, but to show the flavour, $[\![p(v)\langle d\rangle.P]\!]_{\mathsf{ML}}$ produces the code

```
let val v = let val buff = Word8Array.array (σ + 1) 0
                fun receiveloop () =
                   (#(receive) p buff;
                    let val m = unpad buff in
                      if final buff then [m] else m::receiveloop()
                    end)
            in
                List.concat (List.append d (receiveloop ()))
            end
in
  [[P]]ML
end
```

First, a buffer of size $\sigma + 1$ for receiving messages is allocated. Then, the function `receiveloop` repeatedly calls the foreign function `#(receive)`, until a final chunk from p is received. All intermediate chunks of the message are unpadded, and finally concatenated and bound to the variable v before proceeding.

CakeML uses functional big-step semantics [19]; that is, its evaluation semantics is a function eval which maps $(\mathrm{state}, \mathrm{binding\,environment}, \mathrm{expression})$ triples to $(\mathrm{state}, \mathrm{result})$ pairs. Possible results include successful termination returning a value, raising an exception, aborting, or timing out (which encodes divergence). The semantics is parametric on the behaviour of foreign functions: states include a freely chosen model of the outside world, and a freely chosen *oracle function* that describes how this model reacts to FFI calls. We are interested in how our generated CakeML code interacts with the other endpoints in the choreography, so we model the outside world as a triple (p, q, N), where p is the name of the endpoint under consideration, q is its queue, and N is a Payload network that p can interact with, and we use the operational semantics of Definition 8 to define our oracle function. There is an unfortunate mismatch here: the FFI model is expected to be a function, but the semantics of Definition 8 is a one-to-many relation: when we receive a message from N, there is not in general a unique N' that the network will reach after sending, since actions internal to N may or may not fire before N sends the message. However, note that if all endpoints in N have unique names, Payload reductions and send actions are locally confluent. So whether such internal actions fired or not, the resulting states are observationally equivalent from the point of view of p. This justifies using Hilbert choice to obtain such a state.

Let $N \xrightarrow{p\to\widetilde{p}:\widetilde{d}} N'$ denote $N \xrightarrow{p\to p_0:d_0} \dots \xrightarrow{p\to p_n:d_n} N'$. We define the oracle function so that when `#(send) p d` executes in a state (p_1, q, N), then if there is no endpoint named p in N, we abort with a run-time error; otherwise we succeed, producing the new state

$$\varepsilon(p_1, q + \widetilde{(p,d)} + \widetilde{(p',d')}, N').N \xrightarrow{\widetilde{p}\to p_1:\widetilde{d}} \xrightarrow{p\leftarrow p_1:d} \xrightarrow{\widetilde{p'}\to p_1:\widetilde{d'}} N'$$

The semantics of `#(receive)` is similar, with the addition that the FFI call diverges if there is no reduction sequence that will ever cause a message to be enqueued. A key sanity check and technical lemma to show that this Hilbert choice is innocuous is the following:

Lemma 1 (FFI irrelevance).
Two CakeML evaluations starting from equal environments, equal expressions and bisimilar initial states yield equal results and bisimilar final states.

Let N_p denote the unique endpoint named p in N, if one exists, and let $N - p$ denote the network N with all endpoints named p removed. Operational completeness can be formulated as follows.

Theorem 2 (Operational completeness).
If all endpoints in N have unique names, and $N \Longrightarrow N'$, and $N_p = (e, q, P)$, and env is a good environment for N_p, then there exists a good environment env' for $N'_p = (e', q', P')$ such that $\mathsf{eval}((p, q, N - P), env, [\![P]\!]_{\mathsf{ML}})$ and $\mathsf{eval}((p, q', N' - P'), env', [\![P']\!]_{\mathsf{ML}})$ yield equal results and bisimilar states.

Here, a good initial environment for (e, P) is one in which: all bindings of e are present; a few generic library functions such as `List.drop` are defined and have the expected behaviour; and for every function f used in a let expression in P, a CakeML function that is a totally correct implementation of f is present.

By combining Theorem 2 with Theorem 1, we can carry the signature deadlock freedom property of choreographies down to the level of CakeML code.

Corollary 1 (Deadlock freedom). *CakeML evaluation of an endpoint projected from a projectable choreography, when executed in a good environment, will always eventually terminate successfully.*

1.5.2 Dynamic compiler by example

The dynamic compiler creates the initial environment assumed in Theorem 2, and proves that it is good. The environment is built on top of the CakeML basis library by invoking CakeML's proof-producing code synthesis tool [17] on each function used in the endpoints' let expressions.

The choreography language and the compiler are all deeply embedded in HOL4. Hence, users program choreographies by writing instances of the HOL4 datatype that encodes the choreography syntax. We have defined the system from Example 8 as a function `filter` which is parameterised on a test function and a list of messages. In our case, the `test` is a simple function which checks if the message starts with `"a"` or not. To run the compiler, the invocation is

`project_to_camkes` *builddir filename* `“filter test [Kmsg1;Kmsg2;Kmsg3]”;`

By this invocation, we automatically perform the following tasks. We prove that the current environment is good. We evaluate the compiler in the logic to produce CakeML code for each of the three endpoints. We compose instances of Theorems 1 and 2 specialised to each endpoint with all assumptions discharged. Finally, we generate all the glue code and build instructions necessary to create a complete system image that runs our choreography on top of the verified microkernel seL4 [13]. The system consists of three components in parallel, each running our generated CakeML code. The CakeML code is linked with a thin layer of C glue code that implements `send` and `receive` using the dataport and IPC mechanisms of the CAmkES [15] component platform with a message queue implementation on top.

In summary, the user writes a choreography, calls `project_to_camkes`, and obtains a correctly compiled choreography running on a verified component platform on a verified microkernel.

1.6 Related work

Session types [10] have been extensively used to structure communication in concurrent languages, most notably, the π-calculus [11] along with many others [6, 12, 18, 28]. In recent years, an increasing number of results in the topic have included mechanised proofs, perhaps a preventive measure given past occurrences of mistakes in pen-and-paper proofs [20, 29]. In Castro et al. [4] a revised version of the session-typed π-calculus [29] is formalised using the Coq proof assistant [25]. Furthermore, Thiemann [26] proves type soundness and session fidelity for an asynchronous functional session type language based on Guy et al. [8], with a machine-checked proof in Agda [1]. Tassarotti et al. [24] develop a higher-order concurrent logic, and present, as an example, the verification of a refinement procedure for a concurrent session-typed language.

Choreographic programming languages are closely related in scope and intent to session types, but differ in that instead of enforcing compliance with an abstract view of the system interaction, the program itself is a concrete global representation.

The work of Hallal et al. [9] synthesises each component of a communicating system from a global choreography into a distributed component based framework. This result differs from ours in that it aims to only capture the communicating logic of a system in both source and target languages, while our approach considers both communication and computation.

Carbone and Montesi [7, 16] presents a choreography language with multiparty asynchronous session types (demonstrating the combination of the two approaches to great effect) along with a projection function into a variant of the calculus for multi-party sessions [12] with a proof—albeit pen-and-paper—of projection correctness. We draw many parallels to this work, as our language is a simplified version of theirs and can potentially be extended to accommodate many of its features with the added benefit of a machine-checked end-to-end proof of correctness.

1.7 Conclusion

We have presented, to the best of our knowledge, the first end-to-end verified compiler for a choreography language supported by mechanised proofs. This suggests a number of interesting directions for future work. The choreography language could be made more full-fledged; in particular, a recursion construct is necessary to make the language practically useful. Programming could be made more convenient by supporting datatypes other than strings, by adding a framework for verified marshalling and de-marshalling on top. Our model of the communication backend assumes unboundedly long message queues, which is arguably unrealistic: it would be interesting to investigate if deadlock freedom carries over to a model where queue space is bounded. The CakeML compiler correctness theorem has an “unless the compiler output runs out of memory” side-condition, so liveness properties such as deadlock freedom carry over to the machine code only with this side condition. We are currently developing a cost semantics for CakeML that would let us discharge it.

$$\textsc{Send}\ \frac{e(v) = d \qquad p_1 \neq p_2}{(p_1, e, q) \rhd \overline{p_2}\, v.P \xrightarrow{p_1 \to p_2 : d} (p_1, e, q) \rhd P}$$

$$\textsc{Enqueue}\ \frac{p_1 \neq p_2}{(p_1, e, q) \rhd P \xrightarrow{p_2 \leftarrow p_1 : a} (p_1, e, q + (p_2, a)) \rhd P}$$

$$\textsc{Par-L}\ \frac{N_1 \xrightarrow{\alpha} N_1'}{N_1 \mid N_2 \xrightarrow{\alpha} N_1' \mid N_2}$$

$$\textsc{Com-L}\ \frac{p_1 \neq p_2 \qquad N_1 \xrightarrow{p_1 \to p_2 : a} N_1' \qquad N_2 \xrightarrow{p_2 \leftarrow p_1 : a} N_2'}{N_1 \mid N_2 \xrightarrow{\tau} N_1' \mid N_2'}$$

$$\textsc{IntChoice}\ \frac{p_1 \neq p_2}{(p_1, e, q) \rhd p_2 \oplus b.P \xrightarrow{p_1 \to p_2 : b} (p_1, e, q) \rhd P}$$

$$\textsc{Dequeue}\ \frac{q(p_2) = d :: \widetilde{a} \qquad p_1 \neq p_2}{(p_1, e, q) \rhd p_2(v).P \xrightarrow{\tau} (p_1, e[v := d], q - p_2) \rhd P}$$

$$\textsc{ExtChoice-L}\ \frac{q(p_2) = \top :: \widetilde{a} \qquad p_1 \neq p_2}{(p_1, e, q) \rhd p_2 \& \{\top : P, \bot : Q\} \xrightarrow{\tau} (p_1, e, q - p_2) \rhd P}$$

$$\textsc{If-L}\ \frac{e(v) = \top}{(p_1, e, q) \rhd \textbf{if } v \textbf{ then } P \textbf{ else } Q \xrightarrow{\tau} (p_1, e, q) \rhd P}$$

$$\textsc{Let}\ \frac{e(\widetilde{v}) = \widetilde{d}}{(p_1, e, q) \rhd \textbf{let } v = f(\widetilde{v}) \textbf{ in } P \xrightarrow{\tau} (p_1, e[v := f(\widetilde{d})], q) \rhd P}$$

Table 1.4: Endpoint semantics. The obvious symmetric versions of Par-L, Com-L, If-L and ExtChoice-L are elided.

$$\textsc{Send-F}\ \frac{e(v)=d \qquad |d|-n\leq\sigma \qquad p_1\neq p_2}{(p_1,e,q)\triangleright\overline{p_2}\,v_n.P \xrightarrow{p_1\to p_2:\mathsf{pad}(d_{n\ldots})} (p_1,e,q)\triangleright P}$$

$$\textsc{Send-D}\ \frac{e(v)=d \qquad |d|-n\geq\sigma \qquad p_1\neq p_2}{(p_1,e,q)\triangleright\overline{p_2}\,v_n.P \xrightarrow{p_1\to p_2:\mathsf{pad}(d_{n\ldots})} (p_1,e,q)\triangleright\overline{p_2}\,v_{n+\sigma}.P}$$

$$\textsc{Dequeue-F}\ \frac{q(p_2)=d_2::\widetilde{a} \qquad \mathsf{final}(d_2) \qquad p_1\neq p_2}{(p_1,e,q)\triangleright p_2(v)\langle d_1\rangle.P \xrightarrow{\tau} (p_1,e[v:=d_1+\mathsf{unpad}(d_2)],q-p_2)\triangleright P}$$

$$\textsc{Dequeue-I}\ \frac{q(p_2)=d_2::\widetilde{a} \qquad \mathsf{intermediate}(d_2) \qquad p_1\neq p_2}{(p_1,e,q)\triangleright p_2(v)\langle d_1\rangle.P \xrightarrow{\tau} (p_1,e,q-p_2)\triangleright p_2(v)\langle d_1+\mathsf{unpad}(d_2)\rangle.P}$$

Table 1.5: Payload semantics.

$$\begin{aligned}
\mathsf{pr}_p(\mathbf{0}) &= (\top,\mathbf{0})\\
\mathsf{pr}_p(p_1.v_1 \text{->} p_2.v_2;C) &= \begin{cases}(\bot,\mathbf{0}) & \text{if } p_1=p_2=p\\ \overline{p_1}\,v_1.\mathsf{pr}_p(C) & \text{if } p=p_1 \text{ and } p\neq p_2\\ p_2(v_2).\mathsf{pr}_p(C) & \text{if } p\neq p_1 \text{ and } p=p_2\\ \mathsf{pr}_p(C) & \text{otherwise}\end{cases}\\
\mathsf{pr}_p(\mathbf{let}\ v@p_1=f(\widetilde{v})\ \mathbf{in}\ C) &= \begin{cases}\mathbf{let}\ v=f(\widetilde{v})\ \mathbf{in}\ .\mathsf{pr}_p(C) & \text{if } p=p_1\\ \mathsf{pr}_p(C) & \text{otherwise}\end{cases}\\
\mathsf{pr}_p(\mathbf{if}\ v@p_1\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) &=
\end{aligned}$$

$$= \begin{cases}\mathbf{if}\ v\ \mathbf{then}\ \mathsf{pr}_p(C_1)\ \mathbf{else}\ \mathsf{pr}_p(C_2) & \text{if } p=p_1\\ p\,\&\{\top:\mathsf{pr}_p(C_1'),\bot:\mathsf{pr}_p(C_2')\} & \text{if } p\neq p_1 \text{ and } \mathsf{sp}_{p_1,p_2}(C_1)=(\top,C_1')\\ & \text{and } \mathsf{sp}_{p_1,p_2}(C_2)=(\bot,C_2')\\ \mathsf{pr}_p(C_1) & \text{if } p\neq p_1 \text{ and } \mathsf{pr}_p(C_1)=\mathsf{pr}_p(C_2)\\ (\bot,\mathbf{0}) & \text{otherwise}\end{cases}$$

Table 1.6: Projection and projectability, with the selection case (which is treated similarly to communication) elided.

Bibliography

[1] A. Abel, S. Adelsberger, and A. Setzer. Interactive programming in Agda—objects and graphical user interfaces. *J. Funct. Program.*, 27:e8, 2017. doi: 10.1017/S0956796816000319. URL `https://doi.org/10.1017/S0956796816000319`.

[7] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429101. URL `https://doi.org/10.1145/2429069.2429101`.

[8] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In R. De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007. ISBN 978-3-540-71314-2. doi: 10.1007/978-3-540-71316-6_2. URL `https://doi.org/10.1007/978-3-540-71316-6_2`.

[4] D. Castro, F. Ferreira, and N. Yoshida. EMTST: engineering the meta-theory of session types. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 278–285. Springer, 2020. ISBN 978-3-030-45236-0. doi: 10.1007/978-3-030-45237-7_17. URL `https://doi.org/10.1007/978-3-030-45237-7_17`.

[5] L. Cruz-Filipe and F. Montesi. A core model for choreographic program-

ming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi: 10.1016/j.tcs.2019.07.005. URL `https://doi.org/10.1016/j.tcs.2019.07.005`.

[6] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In D. Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006. doi: 10.1007/11785477_20. URL `https://doi.org/10.1007/11785477_20`.

[7] M. Fernandez, J. Andronick, G. Klein, and I. Kuz. Automated verification of RPC stub code. In N. Bjørner and F. S. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 273–290. Springer, 2015. ISBN 978-3-319-19248-2. doi: 10.1007/978-3-319-19249-9_18. URL `https://doi.org/10.1007/978-3-319-19249-9_18`.

[8] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi: 10.1017/S0956796809990268. URL `https://doi.org/10.1017/S0956796809990268`.

[9] R. Hallal, M. Jaber, and R. Abdallah. From global choreography to efficient distributed implementation. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*, pages 756–763. IEEE, 2018. ISBN 978-1-5386-7878-7. doi: 10.1109/HPCS.2018.00122. URL `https://doi.org/10.1109/HPCS.2018.00122`.

[10] K. Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi: 10.1007/3-540-57208-2_35. URL `https://doi.org/10.1007/3-540-57208-2_35`.

[11] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi: 10.1007/BFb0053567. URL `https://doi.org/10.1007/BFb0053567`.

[12] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous ses-

sion types. *J. ACM*, 63(1):9:1–9:67, 2016. doi: 10.1145/2827695. URL https://doi.org/10.1145/2827695.

[13] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. doi: 10.1145/1743546.1743574. URL https://doi.org/10.1145/1743546.1743574.

[14] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535841. URL https://doi.org/10.1145/2535838.2535841.

[15] I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES: a component model for secure microkernel-based embedded systems. *J. Syst. Softw.*, 80(5): 687–699, 2007. doi: 10.1016/j.jss.2006.08.039. URL https://doi.org/10.1016/j.jss.2006.08.039.

[16] F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

[17] M. O. Myreen and S. Owens. Proof-producing synthesis of ML from higher-order logic. In P. Thiemann and R. B. Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 115–126. ACM, 2012. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364545. URL http://doi.acm.org/10.1145/2364527.2364545.

[18] M. Neubauer and P. Thiemann. An implementation of session types. In B. Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004. doi: 10.1007/978-3-540-24836-1_5. URL https://doi.org/10.1007/978-3-540-24836-1_5.

[19] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In P. Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, vol-

ume 9632 of *Lecture Notes in Computer Science*, pages 589–615. Springer, 2016. ISBN 978-3-662-49497-4. doi: 10.1007/978-3-662-49498-1_23. URL `https://doi.org/10.1007/978-3-662-49498-1_23`.

[20] R. Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer, 1993. doi: 10.1007/3-540-58085-9_82. URL `https://doi.org/10.1007/3-540-58085-9_82`.

[21] M. D. Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017. doi: 10.23638/LMCS-13(2:1)2017. URL `https://doi.org/10.23638/LMCS-13(2:1)2017`.

[22] K. Slind and M. Norrish. A brief overview of HOL4. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. ISBN 978-3-540-71065-3. doi: 10.1007/978-3-540-71067-7_6. URL `https://doi.org/10.1007/978-3-540-71067-7_6`.

[23] Y. K. Tan, M. O. Myreen, R. Kumar, A. C. J. Fox, S. Owens, and M. Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019. doi: 10.1017/S0956796818000229. URL `https://doi.org/10.1017/S0956796818000229`.

[24] J. Tassarotti, R. Jung, and R. Harper. A higher-orders logic for concurrent termination-preserving refinement. In H. Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 909–936. Springer, 2017. ISBN 978-3-662-54433-4. doi: 10.1007/978-3-662-54434-1_34. URL `https://doi.org/10.1007/978-3-662-54434-1_34`.

[25] T. C. D. Team. The Coq proof assistant, version 8.11.0, Jan. 2020. URL `https://doi.org/10.5281/zenodo.3744225`.

[26] P. Thiemann. Intrinsically-typed mechanized semantics for session types. In E. Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 19:1–19:15. ACM, 2019. ISBN 978-

1-4503-7249-7. doi: 10.1145/3354166.3354184. URL https://doi.org/10.1145/3354166.3354184.

[27] R. J. van Glabbeek. On the validity of encodings of the synchronous in the asynchronous π-calculus. *Inf. Process. Lett.*, 137:17–25, 2018. doi: 10.1016/j.ipl.2018.04.015. URL https://doi.org/10.1016/j.ipl.2018.04.015.

[28] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368 (1-2):64–87, 2006. doi: 10.1016/j.tcs.2006.06.028. URL https://doi.org/10.1016/j.tcs.2006.06.028.

[29] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007. doi: 10.1016/j.entcs.2007.02.056. URL https://doi.org/10.1016/j.entcs.2007.02.056.

2

Do You Have Space for Dessert?

Alejandro Gómez-Londoño
Johannes Åman Pohjola
Hira Taqdees Syeda
Magnus Myreen
Yong Kiam Tan

Conditionally accepted to OOPSLA 2020

Abstract. Garbage collectors relieve the programmer from manual memory management, but lead to compiler-generated machine code that can behave differently (e.g. out-of-memory errors) from the source code. To ensure that the generated code behaves exactly like the source code, programmers need a way to answer questions of the form: what is a sufficient amount of memory for my program to never reach an out-of-memory error?

This paper develops a cost semantics that can answer such questions for CakeML programs. The work described in this paper is the first to be able to answer such questions with proofs in the context of a language that depends on garbage collection. We demonstrate that positive answers can be used to transfer liveness results proved for the source code to liveness guarantees about the generated machine code. Without guarantees about space usage, only safety results can be transferred from source to machine code.

Our cost semantics is phrased in terms of an abstract intermediate language of the CakeML compiler, but results proved at that level map directly to the space cost of the compiler-generated machine code. All of the work described in this paper has been developed in the HOL4 theorem prover.

2.1 Introduction

High-level programming languages with runtimes that include a garbage collector (GC) provide a layer of abstraction that makes memory seem unbounded. While this liberates the programmer from tedious and error-prone manual memory management, it leads to compiler-generated machine code that exhibits a form of *partiality*: the machine code will behave as the source semantics dictates, unless or until memory is exhausted.

Well written source-level programs stay clear of this partiality by making sure that the live data used by the program stays within some reasonable bound. For such programs, the GC can always reclaim enough memory to provide space for new allocations, even if there are an unbounded number of allocations during the program run.

For certain applications, programmers are keen to make sure that they stay clear of the partiality. In such circumstances, one has to find a way to answer the question: what is a sufficient amount of memory for my program to never reach an out-of-memory error? The answer clearly depends on the exact compilation strategy. In this paper, we provide *a proof-based approach* for answering such questions in the context of the CakeML compiler.

The CakeML compiler [15] is a formally verified compiler for a high-level source language that has no bounds on memory and no bounds on integers. However, the CakeML compiler targets real machine languages (x86-64, ARMv8, RISC-V, etc) where memory and integers have hard bounds. The CakeML compiler inserts a verified GC and bignum library into the code that it produces in order to make it seem as if memory and integers are unbounded. But the GC and bignum library can not always stop the machine code from hitting a hard resource bound, and the machine code might, as a result, have to resort to an out-of-memory error.

The partiality mentioned above is clearly visible in the top-level compiler correctness theorem for the CakeML compiler. This correctness theorem relates the set of behaviours allowed by the source semantics source_sem and the machine semantics machine_sem along the following lines:

$$\begin{array}{l}\mathsf{machine\_sem}\ \mathit{ffi}\ (\mathsf{compile}\ c\ \mathit{prog})\ \subseteq \\ \mathsf{extend\_with\_resource\_limit}\ (\mathsf{source\_sem}\ \mathit{ffi}\ \mathit{prog})\end{array}$$

Here extend_with_resource_limit is a function that augments a set of behaviours with the option to exit early with an out-of-memory error.

The partiality that is expressed using extend_with_resource_limit means that liveness properties proved at the source level do not transfer to liveness properties at the machine code level. For example, suppose one proves a liveness property that a source program will forever print "y" using a program logic [3]. It does *not* follow from the compiler correctness theorem that the generated machine code will forever do the same: the partiality means that only safety properties carry over. The safety property in our example is that, if the machine code produces output, then the output consists of only “y”s.

In this paper, we define a predicate is_safe_for_space that is sufficient to rule out this partiality and extend the CakeML compiler proofs to give stronger guarantees for when is_safe_for_space holds. The is_safe_for_space predicate defines a space cost semantics for CakeML programs, and the new compiler correctness theorem states that the cost semantics rules out all potential for early termination. The new top-level theorem has the following shape.

$$\begin{array}{l}\mathsf{is\_safe\_for\_space}\ \mathit{ffi}\ c\ \mathit{prog}\ \ldots\ \Rightarrow \\ \mathsf{machine\_sem}\ \mathit{ffi}\ (\mathsf{compile}\ c\ \mathit{prog}) = \mathsf{source\_sem}\ \mathit{ffi}\ \mathit{prog}\end{array}$$

Note that the new relationship between source and target semantics here is equality, not refinement: the (deterministic) source semantics defines exactly one permitted behaviour, and the machine semantics implements precisely that behaviour. This equality means that liveness properties proved for the source level carry over directly to liveness properties of the machine code.

Contributions. This paper’s contributions are:

- We define a formal space cost semantics for the CakeML programming language. The definition is stated in terms of one of the intermediate languages used by the CakeML compiler. This intermediate language is at a high enough level to avoid reasoning about data pointers and heap objects, and yet at a low enough level to allow precise reasoning about heap and stack space usage. In addition, the semantics is designed to handle the most common forms of pointer aliasing found in functional programming languages. The cost semantics only considers the live part of the state and, as a result, can be used to derive space bounds for programs that call memory allocation an unbounded number of times.

- We prove that the cost semantics is sound for an end-to-end verified compiler that relies on (verified) garbage collection for correct operation. This is the first such result. The proof covers not only the compiled program, but also the implementation of the GC and the bignum library. When the cost semantics is used to rule out early exits, we get a strong compiler correctness theorem in terms of equality of observable behaviour, since all out-of-memory errors and other resource bound errors are avoided.

- We show that the cost semantics is concrete enough to prove specific space bounds for a few sample programs and, once bounds have been proved, liveness properties proved at the level of source code transfer directly to liveness properties about the compiler-generated machine code. This paper is the first to demonstrate that this is possible in the context of a verified compiler for a language whose compilation relies on automatic memory management. We consider both finite and infinite time liveness properties.

All of the work presented in this paper has been developed in the HOL4 theorem prover [25] and is available as supplementary material with this paper submission.

Limitations. We delimit the scope of our investigation as follows. Our primary goal in this paper is to make space cost reasoning *possible*; making it *convenient* for CakeML users is future work. We do not consider (external) dynamic allocation: the CakeML binary asks the OS for all the memory it will ever need up-front, and manages its own stack and heap within this statically allocated region. Hence our memory model does not need to consider questions like "will the OS give us enough space when we call `malloc`?". For CakeML programs that use the foreign function interface (FFI), we do not model the space cost of code outside the FFI boundary; this does not impact soundness, because the foreign function cannot give memory it allocates back to CakeML.

2.2 Overview

This section describes our overall design and explains how the problem is divided up into separate parts. Subsequent sections describe the separate parts in more detail.

2.2.1 Why can generated code exit early?

The cost semantics needs to predict when early exits might happen, so let us start by looking at the circumstances under which the code emitted by the CakeML compiler resorts to an early exit. The circumstances are:

(H) the creation of a new heap element (e.g. a datatype constructor, array, or bignum integer) does not fit into the heap, even after a full GC run;

(S) a function or primitive operation attempts to allocate stack past the end of the memory region reserved for representing the stack;

(L) the program tries to create an object whose length exceeds what can be represented in the bits reserved for the length field in heap objects;

(F) the program tries to run an incompatible primitive, e.g. a floating-point instruction on a target architecture that does not support it.

Case H is an out-of-heap error. Case S is an out-of-stack error. Cases L and F are possibly more exotic. One could argue that the compiler should catch many instances of case L and F at compile time. However, for case L, this is not always possible because the length of new arrays and vectors can be computed dynamically. Regarding case F, we want to be able to compile a standard library (which includes floating-point primitives) to all targets; thus the compiler will generate some code for all primitives.

2.2.2 Where are the early exits generated?

The CakeML compiler uses 9 intermediate languages and makes in total more than 40 compilation passes over its input, but only two compilation passes insert code that can cause early exits. The relevant intermediate languages are the following:

- DataLang is an imperative language where values are abstract and integers arbitrarily large; there is no notion of garbage collector in this language (see Section 2.3).

- WordLang has a similar structure to DataLang but values are machine words and memory is an array of machine words; the garbage collector is an opaque primitive.
- StackLang has a concrete stack: the stack is a fixed-size array of machine words. The GC stops being a primitive; the compiler inserts code implementing it.

Early exits for cases H, L and F are inserted by the compilation pass that converts DataLang into WordLang, and early exits for case S are inserted by the WordLang to StackLang pass.

2.2.3 At what level of abstraction should the cost semantics be expressed?

At first glance, it seems most natural to express the cost semantics at the level of the source semantics. However, since we are interested in sound, concrete and tight bounds rather than asymptotic bounds, a source-level based approach would have several drawbacks.

The CakeML compiler makes many function-call related optimisations [22] that significantly improve speed and space usage. Because these optimisations mostly happen before the compiler phases that can introduce early exits, a source-level cost semantics must either (1) use very loose approximations of space usage, or (2) specify exactly which optimisations will be applicable on the given program, essentially re-implementing the compiler inside the cost semantics.

We consider both alternatives unacceptable. Our approach is based on the insight that instead of re-implementing the compiler inside the cost semantics, we can obtain the same precision by folding the compiler optimisations into the program under consideration before space cost analysis.

Hence our cost semantics is expressed at the level of the DataLang intermediate language. This allows us to be very precise with respect to resource usage without encumbering the cost semantics with compiler implementation details.

2.2.4 Definition of is_safe_for_space

As motivated above, we define our cost semantics based on the DataLang level of abstraction. The following is our definition of is_safe_for_space, which is our criterion for determining whether a *source-level program* is safe for space.

```
is_safe_for_space ffi c prog stack_heap_limit ≝
 let data_prog = fst (to_data c prog) ; word_prog = to_word c prog in
   c.data_conf.gc_kind ≠ None∧
   data_lang_safe_for_space ffi data_prog
     (compute_limits c c.data_conf.has_fp_ops
                       c.data_conf.has_fp_tern
                       stack_heap_limit)
     (compute_stack_frame_sizes c word_prog)
     Start_location
```

In this definition, to_data compiles the source program *prog* to DataLang; then data_lang_safe_for_space is used to decide whether the resulting DataLang program is safe for space (see Section 2.3).

The data_lang_safe_for_space predicate takes several arguments. It takes the initial state of the foreign function interface *ffi*, the DataLang program *data_prog*, the configuration of limits, a mapping describing how large each stack frame is, and finally the start location in the program.

The definition above mentions to_word which compiles the source program *prog* to WordLang. The input source program is compiled to WordLang in order to compute the size of stack frames for each function that appears in the DataLang program. The cost semantics for DataLang tracks stack usage based on the provided stack frame size mapping (see Section 2.5).

The last conjunct of the definition requires the compiler configuration *c* to have gc_kind not equal to None (i.e. some garbage collector needs to be used). The other alternatives are Simple for a non-generational copying GC [19], and Generational for a generational collector [14]. Our cost semantics requires a GC to be installed, therefore None is a disallowed configuration. We have proved our cost semantics sound w.r.t. the implementation of both the Simple and the Generational GC.

2.2.5 A note on semantics

The semantics of CakeML, and all of its intermediate languages, is defined in the functional big-step semantics style [21]. The core of such a semantics is a clocked big-step evaluation function evaluate which maps (state, program) pairs to (state, result) pairs. The state includes a clock which decrements at every instruction that might potentially induce divergence (such as function calls); if the clock runs out, evaluate aborts with a special timeout result. The state also includes a trace of all I/O events that have happened so far.

The top-level observable semantics function (called semantics) is defined based on the evaluate function described above. The semantics function returns a set of *behaviours*. A behaviour is one of the following:

Terminate *reason events* — indicates that, for some clock value, evaluate terminates in a well-defined way (for a specific *reason*) after producing the I/O events *events*.

Diverge *events* — indicates that, for every clock value, evaluate times out, and *events* is the supremum of the I/O traces produces by evaluate for different initial clock values.

Fail — indicates that the semantics can get stuck.

The function extend_with_resource_limit extends a set of behaviours to allow early termination with an out-of-memory error, i.e. Terminate where the reason is Resource_limit_hit. Here $\preccurlyeq$ checks whether the first list is a prefix of the second, l is a finite list of characters, and ll is a finite or infinite list of characters.

$$\begin{array}{l}
\mathsf{extend\_with\_resource\_limit}\ \mathit{behaviours} \overset{\text{def}}{=} \\
\ \mathit{behaviours} \cup \\
\ \{\ \mathsf{Terminate\ Resource\_limit\_hit}\ \mathit{io\_list} \\
\ \ \ \mid \exists\, t\ l.\ \mathsf{Terminate}\ t\ l \in \mathit{behaviours} \wedge \mathit{io\_list} \preccurlyeq l\ \} \cup \\
\ \{\ \mathsf{Terminate\ Resource\_limit\_hit}\ \mathit{io\_list} \\
\ \ \ \mid \exists\, ll.\ \mathsf{Diverge}\ ll \in \mathit{behaviours} \wedge \mathit{io\_list} \preccurlyeq ll\ \}
\end{array}$$

2.2.6 Structure of the proofs

The aim of our proofs is to show that the observational semantics is preserved completely, i.e. the semantics functions are related with equality = rather than … ⊆ extend_with_resource_limit … as described in the introduction.

Nearly all compiler phases preserve observational semantics with equality, so no changes are required to those. Recall from Section 2.2.1 that the two phases that use the weaker relationship are: the DataLang-to-WordLang phase, which quits on out-of-heap errors and cases L and F from Section 2.2.1; and the WordLang-to-StackLang phase, which quits on out-of-stack errors.

For both of these phases, we define a predicate that implies that the observational semantics is related by = directly. For the DataLang-to-WordLang phase, this is data_lang_safe_for_space. For the WordLang-to-StackLang phase, we define a similar predicate, called word_lang_safe_for_space.

In order to avoid burdening the user with proofs in two cost semantics, we instrument DataLang with enough stack size tracking to prove that data_lang_safe_for_space implies word_lang_safe_for_space. As a result, users only need to prove data_lang_safe_for_space.

2.3 DataLang and its semantics

As of this paper, DataLang has two roles: (1) it acts as an intermediate language of the CakeML compiler, and (2) it defines the heap and stack cost semantics for the compiler.

2.3.1 DataLang as an intermediate language

DataLang is an imperative language with abstract values, stateful storage of local variables, and a call stack. The semantics of DataLang models primitive values with the following datatype:

```
v = Number int
  | Word64 word64
  | CodePtr num
  | RefPtr num
  | Block timestamp tag (v list)
```

Here Number represents an arbitrarily large integer. Word64 is a 64-bit machine word. CodePtr is a code pointer, and RefPtr is a pointer to mutable state (such as ML arrays).

Block is more interesting: it is used to encode datatype constructors, tuples and vectors. For instance, the CakeML list [1,2] can be represented using DataLang Blocks as:

```
Block 8 cons_tag [Number 1;
                  Block 7 cons_tag [Number 2;
                                    Block 0 nil_tag []]]
```

Here the tag values, cons_tag and nil_tag, indicate which source-level constructor each Block represents. The tag information is for pattern matching.

```
α state = ⦇
  locals : v num_map;
  refs : ref num_map;
  stack : stack list;
  handler : num;
  global : num option;
  space : num;
  code : (num × prog) num_map;
  ffi : α ffi_state;
  clock : num;
  ...
⦈

ref = ValueArray (v list) | Bytes bool (word8 list)
```

Figure 2.1: The definition of the DataLang state.

The timestamps of the blocks are 8, 7 and 0, respectively; we will explain the purpose of timestamps in Section 2.3.2.

The runtime state of DataLang's semantics is represented by a record type state shown in Figure 2.1. The fields locals and refs represent the finite maps of local variables (v num_map) and references (v ref num_map) respectively. The stack is a list of frames, each frame containing only the relevant variables that should be restored after a call is completed. On exception, the length of the stack is set to be equal to handler, dropping the most recent frames and setting the value of handler according to the new current frame. The global field contains an optional reference to an array of global variables. The space field is a guaranteed amount of space available in the heap, and can be increased by doing allocation. This is for bookkeeping only; the DataLang semantics maintains the fiction that more space can always be allocated. Finally, the remaining fields (some of them elided in Figure 2.1) pertain to the code store, the state of the foreign function interface, and the semantic clock, respectively.

DataLang's abstract syntax (see Figure 2.2) provides most of the expected features for an imperative language. A notable omission is looping constructs. These are omitted because functional programs use (tail) recursion, which is available as part of Call.

In the abstract syntax presented in the Figure 2.2, var (a type alias for the

```
prog = Skip
     | Seq prog prog
     | If var prog prog
     | Move var var
     | Assign var op (var list) (var_set option)
     | MakeSpace var var_set
     | Raise var
     | Return var
     | Tick
     | Call ((var × var_set) option) call_dest (var list)
          ((var × prog) option)
```

Figure 2.2: DataLang's abstract syntax.

type of natural numbers) represents variable names; var_set is a set of local variables that are to be included in the stack frame when performing a Call, and should be considered live by the garbage collector when allocating (MakeSpace). The evaluation of a DataLang program returns an optional result along with a new state. We give a few samples of DataLang's evaluation semantics below.

The simplest program is Skip. It does nothing. The result is None because there was no return or exception raised.

$$\text{evaluate}\ (\text{Skip},s) \overset{\text{def}}{=} (\text{None},s)$$

Sequencing (Seq) continues execution as long as no return or exception is raised:

$$\begin{array}{l}\text{evaluate}\ (\text{Seq}\ c_1\ c_2,s) \overset{\text{def}}{=} \\ \ \text{let}\ (res,s_1) = \text{evaluate}\ (c_1,s)\ \text{in} \\ \ \ \text{if}\ res = \text{None then evaluate}\ (c_2,s_1)\ \text{else}\ (res,s_1)\end{array}$$

All of the primitive operations are performed by Assign, which deletes unused variable bindings, then reads the values of its arguments, and finally performs the primitive operations using the helper function do_app.

```
evaluate (Assign $dest\ op\ args\ names\_opt$,$s$) $\overset{\text{def}}{=}$
  case cut_state_opt $names\_opt\ s$ of
    Some $s$ $\Rightarrow$
    case get_vars $args\ s$.locals of
      Some $xs$ $\Rightarrow$
      case do_app $op\ xs\ s$ of
        Rval ($v$,$s$) $\Rightarrow$ (None,set_var $dest\ v\ s$)
      | Rerr $e$ $\Rightarrow$ (Some (Rerr $e$),$s$)
  | … $\Rightarrow$ (Some (Rerr (Rabort Rtype_error)),$s$)
```

For a more detailed description of the DataLang semantics, including MakeSpace and Call, we refer to Tan et al. [15].

2.3.2 DataLang as a cost semantics

DataLang provides a convenient level of abstraction for reasoning about space consumption since functions are first-order and data has predictable size. However, DataLang's semantics has no notion of the heap, does not specify which data elements are heap allocated, and does not represent stack frames in a way that makes their size clear. Therefore, we need to add some mechanisms to make DataLang suitable for accurate space measurements. We add elements to the semantics state of DataLang to model the following:

1. A measurement of heap cost: the total space consumed by all values that would be heap-allocated by the implementation. This measure should only count live data, and so needs to be unchanged by garbage collection.

2. A measurement for stack frame sizes, and subsequently the call stack, that is consistent with their eventual implementation in StackLang. We defer further explanation of stack costs to Section 2.5.

3. A signalling mechanism to track if at any point during execution either the stack or the heap surpassed some given limits. The signal is implemented as a new field called safe_for_space in the state record of DataLang.

These elements are represented as follows:

```
α state = ⦇                               limits = ⦇
  ...                                       heap_limit : num;
  safe_for_space : bool                     length_limit : num;
  stack_frame_sizes : num num_map;          stack_limit : num;
  limits : limits;                          arch_64_bit : bool
⦈                                         ⦈
```

The safe_for_space field is true as long as program evaluation stays within the limits. We say that a program *prog* is safe for space with respect to some *limits*, if every execution, regardless of the value of the initial clock *ck*, manages to keep safe_for_space set to true:

$$\begin{array}{l} \text{data\_lang\_safe\_for\_space}\ \mathit{ffi}\ \mathit{prog}\ \mathit{limits}\ \mathit{ss}\ \mathit{main} \overset{\text{def}}{=} \\ \quad \forall\, \mathit{ck}\ \mathit{res}\ s. \\ \qquad \text{evaluate (Call None (Some}\ \mathit{main}\text{) [] None,initial\_state}\ \mathit{ffi}\ \mathit{prog}\ \mathit{limits}\ \mathit{ss}\ \mathit{ck}) = (\mathit{res},s) \\ \qquad\quad \Rightarrow s.\text{safe\_for\_space} \end{array}$$

At every memory allocation, the semantics computes the size of the live data in the heap, adds this number to the requested space *k*, and checks whether we might be exceeding the heap limit:

$$\text{size\_of\_heap}\ s + k \leq s.\text{limits.heap\_limit}$$

The semantics also checks that the stack size is below the stack limit at every function call. If either of these tests fail at any point, safe_for_space is set to false. Further down, after discussing aliasing, we will show the definition of size_of_heap.

Aliasing information. Before presenting our strategy for computing the live heap data, we explain how the semantics maintains aliasing information. Functional programs give rise to a lot of pointer aliasing. Consider, for example, the following snippet of ML code:

```
let val a = [1,2] in (a,0::a) end
```

This code evaluates to a tuple of two lists of integers, `[1,2]` and `[0,1,2]`. To accurately compute the size of this tuple value, the semantics needs to carry information from which we can infer that the memory representation of the two lists share a tail.

We add timestamps to the Block values of the DataLang semantics that let us detect when Block values are pointer-equal. Each new heap element gets a unique timestamp for all of its Blocks. Hence, by keeping timestamps invariant through a Block's lifetime, we can infer that any two Blocks that share a timestamp must refer to the same location on the heap.

The example of a tuple holding two integer lists above can be represented by the following value in DataLang by our semantics.

$$
\begin{array}{l}
\mathsf{block\_example} \overset{\mathrm{def}}{=} \\
\quad \mathsf{let}\ a\ = \\
\qquad \mathsf{Block}\ 8\ \mathsf{cons\_tag} \\
\qquad\quad [\mathsf{Number}\ 1;\ \mathsf{Block}\ 7\ \mathsf{cons\_tag}\ [\mathsf{Number}\ 2;\ \mathsf{Block\_nil}]]\ \mathsf{in} \\
\quad\ \ \mathsf{Block}\ 10\ \mathsf{tuple\_tag}\ [a;\ \mathsf{Block}\ 9\ \mathsf{cons\_tag}\ [\mathsf{Number}\ 0;\ a]]
\end{array}
$$

If we expand the above let-expression, it is clear that Blocks with timestamps 8 and 7 repeat.

We compute the size of all live data on the heap using a function called size_of that is aware of the meaning of timestamps. Before we define it, let us consider its application to the above example. We have that size_of returns 12 when applied to block_example. The size_of function counts each two-element Block as size 3 and each zero-element Block as size 0. The example above has 4 unique two-element Blocks, thus $4 \times 3 = 12$. The unit is machine words of heap space.

$$\vdash \mathsf{fst}\ (\mathsf{size\_of}\ [\mathsf{block\_example}]\ \mathsf{empty}\ \mathsf{empty}) = 12$$

It is worth mentioning that a naive size measure that ignores aliasing information would have produced an over-approximation of $6 \times 3 = 18$, because there are 6 non-empty Blocks in the block_example.

Computing the size of the heap. The following are some of the equations of the definition of size_of. Other equations are provided further down.

```
size_of [] $refs$ $seen$ $\overset{\text{def}}{=}$ $(0,refs,seen)$
size_of [Number $i$] $refs$ $seen$ $\overset{\text{def}}{=}$
  (if is_smallnum $i$ then 0 else bignum_size $i$,$refs$,$seen$)
size_of [Block $ts$ $tag$ $vs$] $refs$ $seen$ $\overset{\text{def}}{=}$
  if $vs = [] \lor ts \in seen$ then $(0,refs,seen)$
  else
    let $(n,refs',seen')$ = size_of $vs$ $refs$ $(\{ts\} \cup seen)$ in
      $(n + |vs| + 1,refs',seen')$
size_of $(x::xs)$ $refs$ $seen$ $\overset{\text{def}}{=}$
  let $(n_1,refs_1,seen_1)$ = size_of $xs$ $refs$ $seen$ ;
      $(n_2,refs_2,seen_2)$ = size_of $[x]$ $refs_1$ $seen_1$ in
    $(n_1 + n_2,refs_2,seen_2)$
```

Small numbers are stored within their containing block or stack frame, and hence they have heap size 0; bignums use heap space proportional to the number of digits in their binary representation.

Empty Blocks are stack-allocated and have heap size 0. The size_of function ignores Blocks with timestamps that are present in *seen*. In all other cases, Blocks add the length of their payload plus one to the first return value of size_of. The size_of function uses *seen* to avoid counting the same Block twice.

The size_of function avoids counting references twice by deleting them from the reference store that it carries in the *refs* variable:

```
size_of [RefPtr $r$] $refs$ $seen$ $\overset{\text{def}}{=}$
  case lookup $r$ $refs$ of
    None $\Rightarrow$ $(0,refs,seen)$
  | Some (ValueArray $vs$) $\Rightarrow$
      (let $(n,refs',seen')$ = size_of $vs$ (delete $r$ $refs$) $seen$
       in $(n + |vs| + 1,refs',seen')$)
  | Some (ByteArray $v_2$ $bs$) $\Rightarrow$
      ($|bs|$ div $4 + 2$,delete $r$ $refs$,$seen$)
```

We define size_of_heap as size_of applied to all of the values stored in the DataLang state's stack and global variables.

```
size_of_heap $s$ $\overset{\text{def}}{=}$
  let $(n,\_,\_)$ = size_of (stack_to_vs $s$) $s$.refs empty in $n$
```

The size_of_heap function is used in the semantics whenever an operation that would allocate heap space is executed: if ever size_of_heap plus the amount of heap space requested exceeds the limits, we set is_safe_for_space to false.

2.4 Proving soundness of heap cost

Before this work, the DataLang-to-WordLang phase of the compiler had a correctness theorem phrased in terms of ⊆ and extend_with_resource_limit in order to allow early exits:

$$\begin{array}{l} \ldots \;\Rightarrow \\ \mathsf{semantics}_{\mathsf{word}}\, \mathit{ffi}\ (\mathsf{compile}\ c\ \mathit{prog}) \;\subseteq\; \mathsf{extend\_with\_resource\_limit}\ (\mathsf{semantics}_{\mathsf{data}}\, \mathit{ffi}\ \mathit{prog}) \end{array}$$

As part of this work, we have proved a new alternative correctness theorem which states that, if data_lang_safe_for_space is true, then all behaviours are preserved by equality =.

$$\begin{array}{l} \ldots \wedge \mathsf{data\_lang\_safe\_for\_space}\, \mathit{ffi}\ \mathit{prog}\ \ldots \;\Rightarrow \\ \mathsf{semantics}_{\mathsf{word}}\, \mathit{ffi}\ (\mathsf{compile}\ c\ \mathit{prog}) \;=\; \mathsf{semantics}_{\mathsf{data}}\, \mathit{ffi}\ \mathit{prog} \end{array}$$

One can read this as saying that cost semantics for DataLang is sound. The following subsections discuss our proof of this soundness result.

2.4.1 Proving evaluate-level simulation

Each proof about the relationship between observational semantics (i.e. semantics) is based on a theorem relating the evaluate functions of the languages involved. In order to prove the new semantics theorem that was sketched above, we need to update the main evaluate simulation theorem to state that DataLang's evaluate correctly predicts any early exits that the generated WordLang program might have resorted to.

The theorem describing the evaluate simulation has the following shape, which is similar to most CakeML compiler phases [15]. One can informally read it as follows: if the input program *prog* evaluates to some result (res, s_1) without hitting a dynamic type error (Rabort Rtype_error), then the compiled program, comp *c prog*, will evaluate to a final state that is similar enough according to a state relation state_rel. Here variable c is a compiler configuration.

$$
\begin{array}{l}
\vdash \mathsf{evaluate}_{\mathsf{data}}\ (\mathit{prog},s) = (\mathit{res},s_1)\ \wedge \\
\quad \mathsf{state\_rel}\ c\ s\ t\ \wedge \\
\quad \mathit{res} \neq \mathsf{Some}\ (\mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error})) \Rightarrow \\
\quad\ \exists\, t_1\ \mathit{res}_1. \\
\qquad \mathsf{evaluate}_{\mathsf{word}}\ (\mathsf{comp}\ c\ \mathit{prog},t) = (\mathit{res}_1,t_1)\ \wedge \\
\qquad (\mathit{res}_1 = \mathsf{Some\ NotEnoughSpace} \Rightarrow \\
\qquad\quad t_1.\mathsf{ffi.io\_events} \preccurlyeq s_1.\mathsf{ffi.io\_events}\ \wedge \\
\qquad\quad \boxed{(c.\mathsf{gc\_kind} \neq \mathsf{None} \Rightarrow \neg s_1.\mathsf{safe\_for\_space})}\ )\ \wedge \\
\qquad (\mathit{res}_1 \neq \mathsf{Some\ NotEnoughSpace} \Rightarrow \\
\qquad\quad \mathsf{state\_rel}\ c\ s_1\ t_1\ \wedge \ldots)
\end{array}
$$

Compared with other CakeML compiler phases, the unusual part here is the special case for the NotEnoughSpace result. For this result, the original theorem only concluded that the WordLang state's I/O events are a prefix ($\preccurlyeq$) of the I/O events produced by the DataLang program *prog*.

For the cost semantics proofs, we added the part in a $\boxed{\text{box}}$. This box adds that, whenever WordLang resorts to a NotEnoughSpace error result, the DataLang evaluation predicts that this might happen, if a supported GC configuration is used. Thus for the user to prove that the WordLang program never exits early, it suffices to prove that DataLang says it won't happen.

The proof of the evaluate simulation theorem sketched above is complicated and long. The original proof builds on some 35,000 lines of invariant definitions and proofs. All of these were updated to cope with the change highlighted above. Handling early exits due to reasons L and F (from Section 2.2.1) was straightforward. The cases that arise when heap space runs out (case H) are much more interesting and will be discussed in the following subsections.

2.4.2 Notation and invariants

In the next subsection, we describe how we have proved that DataLang's size_of_heap function predicts all heap allocation failures that can happen in the WordLang program. In this subsection, we explain the relevant heap abstractions we use to prove our heap cost analysis sound in Section 2.4.3.

The state relation used in the DataLang-to-WordLang proofs is defined in terms of several layers of abstraction. Fortunately for this work, the most abstract intermediate layer is sufficient for our proofs. In that layer, the heap is modelled as a list of heap_elements.

```
(α, β) heap_element =
   Unused num
  | ForwardPointer num α num
  | DataElement (α heap_address list) num β

α heap_address = Pointer num α | Data α
```

During normal program execution, the heap consists of only DataElements and Unused. ForwardPointers only exist while the GC runs. The natural number (type num in HOL) in Pointer values is the address. We dereference pointers using heap_lookup based on a natural number address a:

```
heap_lookup a [] ≝ None
heap_lookup a (x::xs) ≝
  if a = 0 then Some x
  else if a < el_length x then None
  else heap_lookup (a − el_length x) xs

el_length (Unused l) ≝ l + 1
el_length (ForwardPointer n d l) ≝ l + 1
el_length (DataElement xs l data) ≝ l + 1
```

In the DataLang-to-WordLang proofs, the relationship between DataLang's values and their abstract heap representation is specified by the predicate v_inv. We show the Number and Block cases of v_inv below. A number is represented as a value that will fit in a register if the number is small enough, and otherwise by a pointer to a heap element containing the large number. We omit the definition of Bignum, which is a form of DataElement.

```
v_inv c (Number i) (x,f,t,heap) ≝
 if is_smallint i then x = Data (Word (Smallnum i))
 else
  ∃ptr.
    x = Pointer ptr (Word 0w) ∧
    heap_lookup ptr heap = Some (Bignum i)
```

In our work, we changed the definition of v_inv for the Block case: we added a parameter t which dictates how timestamps stored in Blocks map to addresses in the heap. The fact that the timestamp dictates the representation address means that Blocks are pointer equal if their timestamp coincide. The new part is highlighted with a $\boxed{\text{box}}$.

$$
\begin{array}{l}
\text{v\_inv}\ c\ (\text{Block}\ ts\ n\ vs)\ (x,f,t,heap) \overset{\text{def}}{=} \\
\quad \text{if}\ vs = []\ \text{then}\ x = \text{Data}\ (\text{Word}\ (\text{BlockNil}\ n)) \wedge \ldots \\
\quad \text{else} \\
\quad\quad \exists\, ptr\ xs. \\
\quad\quad\quad \boxed{\text{lookup}\ t\ ts = \text{Some}\ ptr\ \wedge} \\
\quad\quad\quad \text{list\_rel}\ (\lambda\, v\ x.\ \text{v\_inv}\ c\ v\ (x,f,t,heap))\ vs\ xs\ \wedge \\
\quad\quad\quad x = \text{Pointer}\ ptr\ (\text{Word}\ (\text{ptr\_bits}\ c\ n\ |xs|))\ \wedge \\
\quad\quad\quad \text{heap\_lookup}\ ptr\ heap = \\
\quad\quad\quad\quad \text{Some}\ (\text{DataElement}\ xs\ |xs|\ (\text{BlockTag}\ n,[]))
\end{array}
$$

Finally, the next subsection uses the following combination of heap_lookup and el_length.

$$\text{get\_len}\ heap\ p \overset{\text{def}}{=} \text{case heap\_lookup}\ p\ heap\ \text{of None} \Rightarrow 0 \mid \text{Some}\ x \Rightarrow \text{el\_length}\ x$$

2.4.3 Correctness of heap allocation and size_of

The DataLang semantics decides that a heap allocation is *not safe for space* if the following test returns *true*. Here k is the number of words of space that have been requested.

$$s.\text{limits.heap\_limit} < \text{size\_of\_heap}\ s + k$$

This section describes our soundness proof for this test, i.e. why this test at the DataLang level must return true whenever an allocation failure might happen at the WordLang level.

At the WordLang level, a heap allocation failure happens only when not enough space is available after a full (compacting) GC run. Since the GC has run, we can assume that all of the DataElements in the heap are reachable from the root variables. And since the WordLang space test has failed, we can assume that the total amount of Unused space in the heap—call it *sp*—is not sufficient to satisfy the allocation request, i.e. $sp < k$. Thus it suffices to show:

$$s.\text{limits.heap\_limit} \le \text{size\_of\_heap}\ s + sp$$

which is equivalent to:

$$s.\text{limits.heap\_limit} - sp \le \text{size\_of\_heap}\ s$$

$$\frac{}{\textsf{traverse}\ \mathit{heap}\ p_1\ []\ p_1}$$

$$\frac{\textsf{traverse}\ \mathit{heap}\ p_1\ \mathit{vs}_1\ p_2 \qquad \textsf{traverse}\ \mathit{heap}\ \ p_2\ \mathit{vs}_2\ p_3 \qquad \textsf{set}\ \mathit{vars} = \textsf{set}\ (\mathit{vs}_1 \mathbin{++} \mathit{vs}_2)}{\textsf{traverse}\ \mathit{heap}\ p_1\ \mathit{vars}\ p_3}$$

$$\frac{}{\textsf{traverse}\ \mathit{heap}\ p_1\ [\textsf{Data}\ d]\ p_1}$$

$$\frac{\textsf{mem}\ n\ p_1}{\textsf{traverse}\ \mathit{heap}\ p_1\ [\textsf{Pointer}\ n\ t]\ p_1}$$

$$\frac{\textsf{heap\_lookup}\ n\ \mathit{heap} = \ \textsf{Some}\ (\textsf{DataElement}\ \mathit{xs}\ l\ d) \qquad \textsf{traverse}\ \mathit{heap}\ (n{::}p_1)\ \mathit{xs}\ p_2}{\textsf{traverse}\ \mathit{heap}\ p_1\ [\textsf{Pointer}\ n\ t]\ p_2}$$

Figure 2.3: Definition of traverse.

The left-hand side above is the same as the sum of the lengths of all heap elements in the DataElement-filled part of the heap. We will call this part of the heap: *heap*. Thus it suffices to prove:

$$\textsf{sum}\ (\textsf{map el\_length}\ \mathit{heap}) \leq \textsf{size\_of\_heap}\ s$$

We have now arrived at the tricky part of this proof: the statement above requires us to prove that every data element in *heap* must be counted (at least once) by size_of_heap, which is defined in terms of the size_of function. This is tricky because the size_of function has a slight disconnect from semantic state: it skips blocks with timestamps that it has accumulated in its *seen* argument and deletes reference values from its *refs* argument during recursion, which means that it cannot evaluate all reference pointers that it encounters even when they exist in the actual heap.

In order to make this proof manageable, we introduce a new inductively defined relation, called traverse, which captures abstractly the traversal patterns that size_of implements using its arguments *seen* and *refs*. The definition of traverse is shown in Figure 2.3. The traverse relation takes four arguments: *heap*, p_1, *vars*, p_2. Here *heap* is the heap being traversed; p_1 and p_2 are lists of addresses which can be viewed as states: p_1 is the input state, and p_2 is the output state; finally *vars* is a working list of heap addresses under consideration. The first rule states that the output state must be equal to the input state if *vars* is empty. The second rule shows how the working list can be split and the state threaded through. The third rule states that traverse can skip data elements on the working list. The fourth rule is more interesting: it states that traverse can skip a pointer if that pointer is already in the input state. The last rule allows traverse to lookup a heap element and place its payload on the working list. For the the last rule, it is worth noting that the traversal of the payload happens from state $n{::}p_1$, i.e. a state where the currently visited address n has already been added to state p_1; this allows traverse to break cycles in the graph of pointers in the heap.

With this definition of traverse, we can prove the following lemma that puts a lower bound on size_of. The following lemma assumes that we have DATALANG *values* that are v_inv-related to some *roots* and *heap*, and that *refs* are in a similar manner (ref_inv) represented in *heap*. If those assumptions hold, then traverse *heap* [] *roots* p_2 is true for some final state p_2. Furthermore, the sum of get_len applied to all addresses in p_2 is ≤ the first component of the result of size_of.

$$\begin{array}{l}
\vdash \mathsf{size\_of}\ \mathit{values}\ \mathit{refs}\ \mathsf{empty} = (n,r,s)\ \wedge \\
\quad \mathsf{v\_inv\_list}\ c\ \mathit{roots}\ (\mathit{values},f,t,\mathit{heap})\ \wedge \\
\quad (\forall n.\ n \in \mathsf{reachable\_refs}\ \mathit{values}\ \mathit{refs} \Rightarrow \mathsf{ref\_inv}\ c\ n\ \mathit{refs}\ (f,t,\mathit{heap},\mathit{be})) \Rightarrow \\
\qquad \exists p_2.\ \mathsf{traverse}\ \mathit{heap}\ []\ \mathit{roots}\ p_2 \wedge \mathsf{sum}\ (\mathsf{map}\ (\mathsf{get\_len}\ \mathit{heap})\ p_2) \le n
\end{array}$$

The proof of this lemma requires stating fiddly assumptions about the accumulated arguments of size_of, but is otherwise a reasonably straightforward proof by induction over the recursive structure of the size_of function.

Let us continue the soundness proof for the check of running out of space. In that context, we use the lower bound lemma from above to establish that there exists a p_2 such that:

$$\begin{array}{l}
\mathsf{sum}\ (\mathsf{map}\ (\mathsf{get\_len}\ \mathit{heap})\ p_2) \le \mathsf{size\_of\_heap}\ s\ \wedge \\
\mathsf{traverse}\ \mathit{heap}\ []\ \mathit{roots}\ p_2
\end{array}$$

With this knowledge, it suffices to prove:

$$\mathsf{sum}\ (\mathsf{map}\ \mathsf{el\_length}\ \mathit{heap}) \le \mathsf{sum}\ (\mathsf{map}\ (\mathsf{get\_len}\ \mathit{heap})\ p_2)$$

The rest of the proof establishes that every element of *heap* has its address included in p_2 and is thus counted (at least once) in sum (map (get_len *heap*) p_2). The fact that every heap address is included in p_2 follows from the fact that a full GC has been run immediately prior to this, and from the following lemma which states that traverse finds all reachable addresses:

$$\vdash \mathsf{traverse}\ \mathit{heap}\ []\ \mathit{roots}\ p_2 \Rightarrow \mathsf{reachable\_addresses}\ \mathit{roots}\ \mathit{heap} \subseteq \mathsf{set}\ p_2$$

This concludes our sketch of the proof that the DataLang check for heap exhaustion is sound with respect to WordLang's check. The target language, WordLang, operates over a lower level of abstraction, but fortunately all of the tricky proofs were confined to the algorithm-level described above rather than lower layers of data refinements that between DataLang and WordLang.

2.4.4 Lessons learned

Doing heap cost analysis at a level of abstraction where there is no heap has the advantage that reasoning can be carried out at a level closer to the source program. But when defining the size_of function, we were faced with an interesting trade-off between accuracy and ease of reasoning. Our implementation exploits timestamps to avoid counting the same block twice in the presence of aliasing. This significantly improves the tightness of our bounds, at the cost of encumbering the definition with accumulator arguments to keep track of which tags have been seen. This leads to a definition that fails to satisfy some natural algebraic laws; for example, size_of does not in general distribute over list append. It does for heaps that are well-formed in the sense that distinct data elements have distinct tags, but carrying around such well-formedness properties through proofs is cumbersome.

In situations where space is plentiful, precision might be less important than the question of whether there is a bound at all. There it might be more useful to have an imprecise size function that's tailored for ease of reasoning. To this end we have defined approx_of, an alternative to size_of that doesn't track timestamps and hence has nicer algebraic properties. We prove that approx_of is a sound over-approximation of size_of.

Another option is to make size_of even tighter by adding timestamps to data elements other than blocks. For example, our version will count pointer-equal bignums twice if they are aliased.

2.5 Proving soundness of stack cost

The DataLang and WordLang intermediate languages do not commit to a concrete implementation of the stack, and do not allow the programmer to manipulate the stack directly. The semantics of both languages model the stack as a list of *stack frames*, which consist of binding environments for local variables plus optional exception handlers. This list is allowed to grow unboundedly large; hence the semantics of both languages act as if stack space is unbounded.

In this section, we show how to make the DataLang and WordLang semantics stack space aware. As in Section 2.3, we add fields to their state records that track stack usage. These fields are a form of ghost state: they have no effect on the program's semantics beyond the fields themselves. But they are sound predictions of the program's maximum stack usage, and the compiler correctness theorem for the WordLang to StackLang phase—where the stack is implemented in a bounded memory region—shows that early exits due to out-of-stack errors never happen unless thus predicted.

As a first step, we annotate WordLang stack frames with an optional size (num option), measured in machine words:

```
stack_frame =
  StackFrame (num option) local_env (handler option)
```

The intuition is that None here denotes positive infinity, or in other words, a stack frame whose size we have no upper bound for. Its inclusion allows us to preserve soundness in the presence of language features that are not safe for space.[1]

Note that we cannot simply compute a bound for the stack frame from the size of the local environment. This is because the environment does not necessarily contain *all* stack-allocated variables, only those that are treated as roots by the GC; moreover, this is before register allocation, so we do not yet know which local variables will be stack-allocated and which will be stored in registers. Moreover, a stack frame is allocated at the beginning of a function, but during the execution of the function there can be unused areas of the stack frame that are not apparent from inspecting the abstract representation of the local environment.

The size of the entire stack can then be computed as follows:

$$\begin{array}{l}
\text{stack\_size (StackFrame } n\ l\ \text{None::}\mathit{stack}) \overset{\text{def}}{=} \\
\quad \text{option\_binop } (+)\ n\ (\text{stack\_size } \mathit{stack}) \\
\text{stack\_size (StackFrame } n\ l\ (\text{Some } \mathit{handler})\text{::}\mathit{stack}) \overset{\text{def}}{=} \\
\quad \text{option\_binop } (+)\ (\text{option\_map } ((+)\ 3)\ n)\ (\text{stack\_size } \mathit{stack}) \\
\text{stack\_size } [\,] \overset{\text{def}}{=} \text{Some } 1
\end{array}$$

[1]The only language feature of WordLang whose stack usage we don't provide bounds for is the Install instruction for dynamic code evaluation. At present, this instruction is not targeted by the CakeML compiler.

The fact that this is not just a straightforward list sum exposes two compiler-specific implementation details that we include for the sake of more precise bounds: the empty stack is one word long, and installing an exception handler requires three words of stack space.

We annotate the WordLang state with an extra field stack_max, which records the largest stack_size seen so far during the WordLang execution. This field is updated to the maximum of the old value and the current stack_size whenever a WordLang instruction that potentially allocates stack is executed; the relevant instructions for our purposes is function calls and semantic primitives that have an implementation (further down the compilation chain) that internally allocates stack as part of the implementation of the primitive in question.

To populate the stack frames with sizes, we assume that the state also contains a mapping, called stack_frame_sizes, which maps function names to stack frame sizes. It is possible to do symbolic computations about stack usage without committing to any particular mapping. To obtain sound and concrete bounds, the tooling we use in Section 2.7 obtains the actual stack frame sizes by evaluating the compiler in logic down to StackLang. This avoids cluttering the cost semantics with details of how lower parts of the compiler are implemented, in this case, specifically: register allocation which determines the size of stack frames.

These annotations allow us to soundly predict out-of-stack errors, as shown by the compiler correctness theorem for the WordLang-to-StackLang phase:

$$
\begin{array}{l}
\vdash \text{evaluate}\ (prog,s) = (res,s_1) \land res \neq \text{Some Error} \land \text{state\_rel}\ k\ f\ f'\ s\ t\ lens \land \ldots \\
\quad \Rightarrow \\
\quad \exists\, ck\ t_1\ res_1. \\
\qquad \text{evaluate}\ (\text{fst}\ (\text{comp}\ prog\ bs\ (k,f,f')),t\ \text{with clock}\ :=\ t.\text{clock} + ck) = (res_1,t_1) \land \\
\qquad \text{if option\_map compile\_result}\ res \neq res_1\ \text{then} \\
\qquad\quad res_1 = \text{Some (Halt (Word}\ 2w)) \land \\
\qquad\quad t_1.\text{ffi.io\_events} \preccurlyeq s_1.\text{ffi.io\_events} \land \\
\qquad\quad \boxed{s_1.\text{stack\_max} > s_1.\text{stack\_limit}} \\
\qquad \text{else} \\
\qquad\quad \ldots
\end{array}
$$

The $\boxed{\text{boxed}}$ conjunct is the novelty and the key: it states that if StackLang evaluation yields an unexpected result (i.e. *res* and *res′* disagree), then this must have been due to an early exit that was predicted by WordLang evaluation exceeding the stack budget at some point.

In order to allow reasoning about costs in just the one semantics, we lift this stack cost semantics from WordLang to DataLang. The treatment of function calls does not change significantly between the two languages, so that aspect of the semantics is mostly the same. The main difference is that many native operators of DataLang, such as equality and bignum arithmetic, are implemented by canned code in WordLang. When this code features calls to subroutines, the DataLang semantics must make sure to update stack_-max accordingly. Most of these subroutines are either tail-recursive or not recursive, in which case the stack consumption can be characterised as the largest of the involved WordLang stubs' stack frames.

The operator with the most interesting stack usage is probably the equality operator, which can compare arbitrarily nested trees of Blocks; its WordLang implementation must recursively step through these pointer structures and compare the payloads for equality. We prove that its stack usage is bounded from above by a metric on the constructor depth of the DataLang values that the pointer structures refine.

The DataLang-to-WordLang compiler also pastes in canned code that implements the bignum library and this code required some special attention regarding stack usage. The bignum library is reachable from any DataLang integer arithmetic operation that fails to fit within small enough numbers. The WordLang code implementing the bignum library is automatically generated from a higher-level specification [20] and consists of several nested WordLang functions. To ease the effort, we developed a little verified tool that can automatically infer maximum stack depths of WordLang functions where all cycles in the call graph consist of tail-calls. The bignum library fits within this subset of WordLang.

2.5.1 Lessons learned

Proving soundness of the stack cost semantics involved a tedious and cumbersome invariant preservation proof, but the effort invested helped us gain insight. Even though the stack cost semantics is relatively straightforward compared to heap cost, doing a formal soundness proof was invaluable for getting the cost semantics right down to every detail. There were a number of more or less subtle mistakes we made in early drafts of the semantics, that would have been difficult to catch and diagnose without formal proof:

- The WordLang semantics does not explicitly distinguish between whether the current local variables have already been pushed to the stack or not; this requires some care to avoid counting the current stack frame twice in the tally.

- In the StackLang implementation of function calls, stack allocation is done in two increments: enough space for the function arguments is allocated by the caller, then the callee allocates space for the remaining local variables. Our cost semantics abstracts away from this timing detail, which makes it important that we update stack_max before rather than after function calls; otherwise, our bounds will be unsound in case the Call instruction aborts.
- We initially modelled tail calls as not changing stack size, but this is unsound if the tail call is to another function with a larger stack size.
- Exception handler allocation needs to be counted separately from the rest of the stack frame size, as shown above, because the same function may be called both with and without exception handlers.

2.6 Top-level compiler theorem with cost

We have proved a new end-to-end correctness statement for the entire CakeML compiler. In the theorem below, compile performs the entire compilation chain from concrete syntax down to machine code. The new theorem leverages is_safe_for_space to show that, for any successful compilation, execution from any machine state *ms* that has the compiler-generated *code* and *data* installed will produce exactly the same *behaviours* as the source semantics.

$$
\begin{array}{l}
\vdash \mathsf{compile}\ \mathit{cc}\ \mathit{prelude}\ \mathit{input} = (\mathsf{Success}\ (\mathit{code},\mathit{data},\mathit{c}),\mathit{c}') \Rightarrow \\
\quad \exists\, \mathit{behaviours}\ \mathit{source\_decs}. \\
\qquad \mathsf{semantics\_init}\ \mathit{ffi}\ \mathit{prelude}\ \mathit{input} = \mathsf{Execute}\ \mathit{behaviours}\ \land \\
\qquad \mathsf{parse}\ (\mathsf{lexer\_fun}\ \mathit{input}) = \mathsf{Some}\ \mathit{source\_decs}\ \land \\
\qquad \forall\, \mathit{ms}. \\
\qquad\quad \mathsf{is\_safe\_for\_space}\ \mathit{ffi}\ \mathit{cc}\ (\mathit{prelude} +\!\!+\ \mathit{source\_decs})\ (\mathsf{read\_limits}\ \mathit{cc}\ \mathit{ms})\ \land \\
\qquad\quad \mathsf{installed}\ \mathit{code}\ \mathit{data}\ \ldots\ \mathit{mc}\ \mathit{ms} \Rightarrow \\
\qquad\quad \mathsf{machine\_sem}\ \mathit{ffi}\ \mathit{ms} = \mathit{behaviours}
\end{array}
$$

Here we assume is_safe_for_space (i.e. require the user to prove it), but we conclude an equality machine_sem *ffi ms* = *behaviours* instead of the weaker previous formulation that used ⊆ and extend_with_resource_limit as explained in the introduction.

Here read_limits is a function that computes the relevant limits for the cost semantics based on information from the compiler configuration *cc* and the initial machine state *ms*.

2.7 Proving that programs are safe for space

The aim of this paper is to provide a cost semantics that can be used to carry liveness properties proved at the source level down to the machine code level. In this section, we demonstrate that we can do exactly that with our new cost semantics.

2.7.1 Is yes safe for space?

As a first example, we use a CakeML implementation of the yes command, shown in Figure 2.4. This program prints its argument to `stdout` indefinitely.

```
fun put_line l = let
  val s = l ^ "\n"
  val a = Word8Array.array 0 (Word8.fromInt 0)
  val _ = #(put_char) s a (* ffi call *)
in () end;

fun printLoop l = (put_line l; printLoop l)

val _ = printLoop "y"
```

Figure 2.4: Implementation of `yes`.

Before we delve into a formal proof, let's convince ourselves that yes is indeed safe for space.

At first glance, we see a number of expressions within put_line that cause memory allocation. For example, string concatenation requires allocating space for the resulting string. Thus any call to printLoop, which recursively calls put_line indefinitely, will perform an unbounded number of allocations. This is fine since none of the variables in the body of put_line remain in scope, and hence will eventually be garbage collected. This in turn means that the heap footprint of printLoop, as measured by size_of_heap, does not increase between loop iterations.

As for the stack, it is enough to notice that (1) put_line is a non-recursive terminating function that consumes a bounded amount of stack space, and (2) printLoop is tail-recursive, and thus its recursive calls to itself do not grow the stack.

Informally, we conclude that yes must be safe for space, even though it's not clear yet with respect to what heap and stack bounds.

2.7.2 Is yes safe for space, formally?

We will now formalise our intuition from the previous section by showing that evaluation of the yes program satisfies is_safe_for_space as defined in Section 2.2.4. In other words, we show that during evaluation of its DataLang intermediate representation, heap and stack usage never goes above a provided limit. In order to avoid encumbering the proofs with a deeply embedded semantics, we have developed a sound and complete shallowly embedded representation of DataLang programs as a state monad for doing space cost reasoning.

Most of the initial DataLang code generated by the compiler can easily be evaluated in-logic from the concrete initial state; it is only when we reach the body of printLoop that things get interesting. The body of the printLoop looks as follows in the proof.

```
Seq (Call_put_line (Some (1,{0})) [0] None)
    (Call_printLoop None [0] None)
```

This corresponds very closely to the source program. The local variable 0 stores the value of "y". Abbreviations to make function calls readable are automatically installed; for example, Call_printLoop abbreviates $\lambda\, ret.$ Call *ret* (Some 285), where 285 is the code location where the DataLang code generated from printLoop happens to be installed.

From this point onwards the execution will repeat itself indefinitely, and thus data_is_safe_for_space can be proven by complete induction over the semantic clock, and provide us with the following bounds:

$$\begin{array}{l} \vdash \text{the (size\_of\_stack s.stack)} + 17 \leq \text{s.limits.stack\_limit} \land \\ \quad \text{size\_of\_heap s} + 11 \leq \text{s.limits.heap\_limit} \land \ldots \Rightarrow \\ \quad\; \text{(snd (evaluate(Seq (Call} \ldots\text{) (Call} \ldots\text{)))).safe\_for\_space} \end{array}$$

This shows that as long as there are 11 words (88 bytes) of heap and 17 words (136 bytes) of stack left when calling printLoop, we will not run out of memory. (We are compiling to a 64-bit architecture, thus machine words are 8 bytes long.)

The formal proof closely resembles our earlier informal argument, but the details of the formal proof are omitted here. The formal proofs is included as part of the supplementary material.

The resulting is_safe_for_space theorem for the entire yes program is:

$$\vdash \text{is\_safe\_for\_space}\ \mathit{ffi}\ \text{yes\_x64\_conf yes\_prog}\ (56{,}89)$$

Here, the 56 and 89 are the concrete stack and heap bounds measured in machine words. These bounds are obtained during the course of the proof. They are larger than the bounds for the call to printLoop because the surrounding program (e.g. standard library) allocates on the execution up to the point of the call to printLoop.

Having established that our program satisfies is_safe_for_space, a similar top-level correctness theorem, to the one shown in Section 2.6, can be instantiated to read:

$$\begin{array}{l}\vdash 56 \leq \mathit{stack\_limit} \wedge 89 \leq \mathit{heap\_limit} \wedge \\ \quad \text{read\_limits yes\_x64\_conf}\ \mathit{ms} = (\mathit{stack\_limit},\mathit{heap\_limit}) \wedge \\ \quad \text{installed yes\_code} \ldots\ \mathit{ms} \wedge \ldots \Rightarrow \\ \quad\ \ \text{machine\_sem} \ldots\ \mathit{ms} = \text{semantics\_prog} \ldots\ \text{yes\_prog}\end{array}$$

The equality in the theorem above allows us to carry over any liveness property from the source semantics into the machine code semantics.

For our example, we can prove that the yes source-level program will produce an infinite stream of "y" characters on `stdout`.

$$\begin{array}{l}\text{semantics\_prog} \ldots\ \text{yes\_prog} = \\ \quad \{\text{Diverge (lrepeat [put\_str\_event "y"])}\}\end{array}$$

Such a theorem is easy to establish thanks to a program logic for non-terminating CakeML programs [3], where proving this liveness property for the main loop is a 15-line proof. Unfolding the abstractions of the program logic to obtain a corresponding theorem about the CakeML semantics requires some additional boilerplate.

Finally, we combine these two theorems from above to obtain the same liveness property at the level of the compiler-generated machine code:

$$\begin{array}{l} \vdash 56 \le \mathit{stack\_limit} \land 89 \le \mathit{heap\_limit} \land \\ \quad \mathsf{read\_limits\ yes\_x64\_conf}\ \mathit{ms} = (\mathit{stack\_limit},\mathit{heap\_limit}) \land \\ \quad \mathsf{installed\ yes\_code} \ldots\ \mathit{ms} \land \ldots \Rightarrow \\ \quad\ \ \mathsf{machine\_sem} \ldots\ \mathit{ms} = \\ \quad\quad \{\mathsf{Diverge\ (lrepeat\ [put\_str\_event\ "y"])}\} \end{array}$$

One can read this as saying: in a machine state *ms* where there are 56 words of stack and 89 words of heap available, and where the compiler output yes_-code is installed and ready to run, execution from *ms* can exhibit one and only one behaviour: it will produce an infinite stream of "y" on `stdout`. In this case, the theorem is about x86-64 machine code. Since our cost semantics is not tied to a particular architecture, the same result could be reproduced for e.g. ARMv8 or RISC-V with no change to the space cost reasoning.

2.7.3 A linear congruential generator

A *linear congruential generator* (LCG) is a kind of pseudorandom number generator. The basic idea is that if x_i is the current element of the pseudorandom number sequence, the next element is generated by the following equation, for fixed values of a, c, m:

$$x_{i+1} = (ax_i + c) \bmod m$$

For this example, we implement a program that produces an infinite stream of LCG-generated numbers on `stdout`. The source code is shown in Figure 2.5.

This example shares some structural similarities with `yes`, but differs in several ways that have bearing on space-cost reasoning. First, it exercises more language features and reasoning techniques, including truely nested recursive function calls. In particular, `n2l_acc` tail-recursively constructs a list in accumulator passing style. Recall from Section 2.3 that lists are represented by DataLang's `Blocks`. Moreover, the length of the resulting list will depend on the size of the input, so its cost must be expressed as a function of its input. Finally, `put_chars` also tail-recursively deconstructs the same list. This exercises the way size_of infers aliasing information from timestamps: `put_char` requires constant space, but if our analysis failed to account for the structure sharing between the lists `cs` and `xs` and didn't distinguish live memory from garbage, we would be forced to conclude that `put_char` uses $O(|\mathsf{cs}|^2)$ heap space

```
fun n2l_acc n acc =
  if n < 10 then hex n :: acc
  else n2l_acc (n div 10) (hex (n mod 10) :: acc)

fun num_to_string n = n2l_acc n [#"\n"]

fun put_chars cs =
  case cs of [] => ()
  | x::xs => (put_char x ; put_chars xs)

fun print_num n = put_chars (num_to_string n)

fun lcg a c m x = (a * x + c) mod m

fun lcgLoop a c m x =
let
  val x1 = lcg a c m x
  val u = print_num x1
in
  lcgLoop a c m x1
end

val _ = lcgLoop 8121 28411 134456 42
```

Figure 2.5: Implementation of `lcg`. The definition of `put_char` is elided.

Another difference between this example and the previous yes example is that this example uses arithmetic. Arithmetic over small numbers has no stack or heap cost. However, once the numbers are large enough, arithmetic starts to incur the stack and heap costs of invoking the bignum library. The stack cost for bignum operations is not dependent on the size of the given integers, but the heap cost is of course dependent on how large the numbers are. Note that, for programs that only use small numbers, one has to prove that the numbers stay small enough to avoid the cost of bignum operations.

We have proved the code shown in Figure 2.5 to be safe for space (with stack bound 182 and heap bound 199). We proved this by showing that the code stays within the range of small enough integers to avoid triggering CakeML's bignum library. Our proof is largely agnostic to the precise values of the parameters so, in fact, `lcgLoop` can be called with different values of `a`, `c`, `m`, `x` with almost no change to the proofs (as long as the bounds described above are met).

2.7.4 List reverse

In this example, we illustrate the precision advantages we gain by expressing the cost semantics in an intermediate language. Consider the following naive implementation of list reverse, which uses list append (written here in SML syntax: `@`).

```
fun reverse [] = []
  | reverse(f::l) = reverse l @ [f]
```

Figure 2.6: Naive implementation of `reverse`.

An informal source-level cost analysis would force us to conclude that since this function is not tail-recursive, it requires $O(n)$ stack space, where n is the length of the input list, to accommodate the stack frames of the n recursive calls `reverse` makes.

However, the CakeML compiler performs tail-call introduction before it reaches DataLang [1], and this optimisation triggers on the body of `reverse`. In other words, the compiler produces essentially the same code for `reverse` as it does for `reverse'` below:

```
fun reverse'_aux [] acc = acc
  | reverse'_aux (f::r) acc = reverse'_aux r (f::acc)

fun reverse' l = reverse'_aux l []
```

Figure 2.7: Tail-recursive implementation of `reverse`.

Therefore, we can use our cost semantics to prove that our initial naive version of `reverse` uses only a constant amount of stack space:

$$\begin{array}{l}\vdash \text{evaluate}\ (s, \text{reverse\_body}) = (res, s') \wedge \ldots \Rightarrow \\ \quad \exists\, k.\ s'.\text{stack\_max} = \text{option\_map}\ (+\ k)\ s.\text{stack\_max}\end{array}$$

We note that a source-level cost semantics would have to know exactly when the tail-call introduction optimisation kicks in to be able to prove such a property for `reverse`.

The stack costs are concrete enough that we could prove a theorem similar to the one above with a precise numeric value in place of k, and we could additionally consider heap cost to prove that `reverse` is safe for space. However, that is not the point here: our cost semantics is modular enough that when we are only interested in stack usage, we can reason about it separately by considering only stack_max and ignoring safe_for_space. This results in a simpler proof than the previous examples because we do not need to reason about heap usage at all.

2.8 Related work

There has been much interest in defining cost semantics for both imperative and functional programming languages to reason about the resource usage of programs. The main types of resources are *execution time* and *memory space* (heap and stack), and the cost semantics aim to estimate worst-case bounds for these resources either at the source level or during transformation phases through compilers.

Source-level Cost Analysis. Source-level techniques enable static cost analysis. For instance, Hofmann and Jost [16] provide static prediction of heap space usage for functional programs, and Jost et al. [17] develop a type system with heap annotations for determining the execution costs of lazily evaluated functional languages. RelCost [10], CostIt [11], and RaML [15] are resource-aware type systems for source-level programs based on refinement types, and Guéneau et al. [13] provide worst-case asymptotic time complexity of higher-order imperative programs. Wang et al. [28] present an ML-like functional language with time-complexity annotations in indexed types. Handley et al. [14] implement a system based on refinement types to enable reasoning about resource usage of pure Haskell programs in Liquid Haskell. Aspinall et al. [4] develop a program logic for proving statements about resource consumption for the Java Virtual Machine Language (JVML), Atkey [5] formalises a separation logic for heap-resource analysis within the Coq proof assistant, and Vasconcelos [27] uses sized types to obtain upper bounds on dynamic space usage of functional programs. While these source-level analysis techniques provide formal estimates of cost analysis, they ignore the effect of compilation and program transformation on resource consumption, leaving an inherent trust gap between the analysis and the actual machine code that runs.

Preservation of Resource Bounds through Compilation. Resource bounds estimated at the source level can be made accurate and certified by proving their preservation throughout the compilation chain. Crary and

Weirich [12] estimate upper bounds on resources through a decidable type system and a bounds-certifying compiler from the impure functional language PopCron to typed assembly. Resources are modelled as semantic clocks, and a resource-safe program is one for which the clock never expires. While this approach is best suited to modelling time (where resource usage is monotonic), it does in principle generalise to stack and heap usage because there is a mechanism to recover spent resources, provided allocation and deallocation is explicit in the program text. Since this assumption fails to hold in the presence of garbage collection, their approach is not well suited to languages with automatic memory management.

Paraskevopoulou and Appel [23] develop a cost model for the CPS lambda-calculus, in which they derive time and space bounds for a closure conversion compilation phase in the Coq proof assistant. In our work we do not need to explicitly model the space cost of closure conversion; instead, we derive space bounds on code that has already been closure-converted. Their work is also notable for taking garbage collection into account: their measure of space usage assumes that an ideal, complete garbage collector is invoked often enough so that actual heap usage can only exceed the size of the reachable heap by a bounded amount. They can also give bounds for diverging programs. The heap is explicitly present in the memory models of their source and target languages. In contrast, we are able to lift our cost analysis to a level of abstraction where there is no notion of heap, by annotating values in the variable store with timestamps. Unlike Paraskevopoulou and Appel, we cash out our cost model using the completeness proofs for the real garbage collector implementation. Their runtime is stack-less, which allows them to sidestep the problem of finding roots in the stack. CakeML maintains its own stack, and so implements and verifies such root-finding. Finally, our work is fully integrated into an end-to-end verified compiler, allowing space bounds to be leveraged to transfer liveness properties all the way from source to machine code; theirs is not (yet).

Our technique for estimating stack space consumption through an end-to-end compiler closely relates to that of CerCo project [2]. The CerCo project has built a verified C compiler producing object binaries for the 8051 microcontroller in the Matita theorem prover. The compiler precisely estimates the non-asymptotic computational cost involving execution time and stack space usage of input programs at the source level. It also generates source-level annotations that correctly model low level costs. These invariants are then certified through automated theorem provers. Case studies include certifying the exact reaction time of Lustre dataflow programs compiled to C.

While the CerCo project inspires our work for estimating stack space, it does not consider heap usage, let alone garbage collection. Their compiler correctness proof only considers preservation of cost bounds and not functional correctness, whereas the CakeML compiler with our extensions considers both.

The CompCert compiler [18] has also been employed to formally estimate resource bounds for imperative C programs. Carbonneaux et al. [9] develop a logic for reasoning at the source level about stack space consumption of the corresponding CompCert compiler output. They introduce resource consumption events to CompCert that are preserved by compilation and use the compiler itself to determine the actual size of stack frames. Besson *et. al* introduce finite memory and integer pointers to the memory model of CompCert, extend CompCert's front-end for this concrete memory model, and continue to verify its back-end layers to develop CompCertS in Coq [4, 5, 6]. CompCertS estimates the memory usage of individual functions directly at the C level, proves that compiled programs use no more memory than source programs, and ensures that the absence of memory overflow is preserved by compilation. It also provides stronger guarantees about arbitrary pointer arithmetic and avoids the miscompilation of programs performing bit-level pointer manipulation. Wang et al. [29] enrich the memory model of CompCert with an abstract and bounded stack to develop Stack-Aware CompCertX: a complete extension of CompCert with compositional compilation. The main distinction between our work and these is the level of abstraction at which the cost semantics is expressed. In this respect, C is very similar to our WordLang: both languages give the programmer an explicit view of the heap and responsibility for managing heap memory, while abstracting the stack. We express our cost semantics in a language that abstracts away from the heap and features no explicit memory management.

2.9 Conclusion

We have presented a space cost semantics for CakeML programs that makes it possible to prove the absence of out-of-memory errors in the generated machine code. The semantics does so by estimating the resource usage of programs an intermediate representation that avoids reasoning about pointers and heap objects, yet takes aliasing of data elements into account for an accurate estimate. The cost analysis is proven sound down to the machine code, and we have demonstrated that it can be used to carry source-level liveness properties down to machine code: the space analysis rules out all partiality induced by potential out-of-memory errors, and can be applied even to programs that make unboundedly many heap allocations.

In this paper, our primary goal was to make sound space cost reasoning about CakeML programs possible. What remains to show is how such reasoning can be made scalable; while our examples do exhibit interesting and relevant features like non-termination and unbounded allocation, they are admittedly small. There are several interesting ideas to explore in this direction. One is to use coarser overapproximations of the heap size metric to make analysis more compositional. Another is to develop a framework of sound abstractions of the monadic DataLang semantics.

Bibliography

[1] O. Abrahamsson and M. O. Myreen. Automatically introducing tail recursion in cakeml. In M. Wang and S. Owens, editors, *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*, volume 10788 of *Lecture Notes in Computer Science*, pages 118–134. Springer, 2017. ISBN 978-3-319-89718-9. doi: 10.1007/978-3-319-89719-6_7. URL `https://doi.org/10.1007/978-3-319-89719-6_7`.

[2] R. M. Amadio, N. Ayache, F. Bobot, J. P. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. Sacerdoti Coen, I. Stark, and P. Tranquilli. Certified complexity (cerco). In U. Dal Lago and R. Peña, editors, *Foundational and Practical Aspects of Resource Analysis*, pages 1–18, Cham, 2014. Springer International Publishing.

[3] J. Åman Pohjola, H. Rostedt, and M. O. Myreen. Characteristic formulae for liveness properties of non-terminating cakeml programs. In *Interactive Theorem Proving (ITP)*. LIPICS, 2019.

[4] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theoretical Computer Science*, 389(3):411 – 445, 2007.

[5] R. Atkey. Amortised resource analysis with separation logic. In A. D. Gordon, editor, *Programming Languages and Systems*, pages 85–103, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] F. Besson, S. Blazy, and P. Wilke. A precise and abstract memory model for c using symbolic values. In J. Garrigue, editor, *Programming Languages and Systems*, pages 449–468, Cham, 2014. Springer International Publishing.

[4] F. Besson, S. Blazy, and P. Wilke. A concrete memory model for compcert. In *Interactive Theorem Proving*, pages 67–83, Cham, 2015. Springer International Publishing.

[5] F. Besson, S. Blazy, and P. Wilke. Compcerts: A memory-aware verified c compiler using a pointer as integer semantics. *Journal of Automated Reasoning*, 63(2):369–392, Aug 2019.

[9] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for c programs. *SIGPLAN Not.*, 49 (6):270–281, June 2014.

[10] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational cost analysis. *SIGPLAN Not.*, 52(1):316–329, Jan. 2017.

[11] E. Çiçek, D. Garg, and U. Acar. Refinement types for incremental computational complexity. In J. Vitek, editor, *Programming Languages and Systems*, pages 406–431, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[12] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 184–198. ACM, 2000.

[13] A. Guéneau, A. Charguéraud, and F. Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *LNCS - Lecture Notes in Computer Science*. Springer, Apr. 2018.

[14] M. A. T. Handley, N. Vazou, and G. Hutton. Liquidate your assets: Reasoning about resource usage in liquid haskell. In *Principles of Programming Languages (POPL)*, 2020. to appear.

[15] J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ml. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 781–786, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 185–197, New York, NY, USA, 2003. ACM.

[17] S. Jost, P. Vasconcelos, M. Florido, and K. Hammond. Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning*, 59 (1):87–120, Jun 2017.

[18] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009. doi: 10.1145/1538788.1538814.

[19] M. O. Myreen. Reusable verification of a copying collector. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 6217 of *Lecture Notes in Computer Science*. Springer, 2010. ISBN 978-3-642-15056-2. doi: 10.1007/978-3-642-15057-9.

[20] M. O. Myreen and G. Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs (CPP)*, pages 66–81. Springer, 2013.

[21] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In P. Thiemann, editor, *European Symposium on Programming (ESOP)*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615. Springer, Apr. 2016.

[22] S. Owens, M. Norrish, R. Kumar, M. O. Myreen, and Y. K. Tan. Verifying efficient function calls in CakeML. *Proc. ACM Program. Lang.*, 1(ICFP), Sept. 2017.

[23] Z. Paraskevopoulou and A. W. Appel. Closure conversion is safe for space. *Proc. ACM Program. Lang.*, 3(ICFP):83:1–83:29, July 2019. ISSN 2475-1421.

[14] A. Sandberg Ericsson, M. O. Myreen, and J. Åman Pohjola. A verified generational garbage collector for cakeml. *J. Autom. Reasoning*, 63(2): 463–488, 2019. doi: 10.1007/s10817-018-9487-z.

[25] K. Slind and M. Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[15] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. The verified cakeml compiler backend. *Journal of Functional Programming*, 29, 2019.

[27] P. B. Vasconcelos. *Space Cost Analysis Using Sized Types.* PhD thesis, Ph.D. Dissertation. University of St. Andrews, 2008.

[28] P. Wang, D. Wang, and A. Chlipala. Timl: A functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA):79:1–79:26, Oct. 2017.

[29] Y. Wang, P. Wilke, and Z. Shao. An abstract stack based approach to

verified compositional compilation to machine code. *Proc. ACM Program. Lang.*, 3(POPL):62:1–62:30, Jan. 2019.