Thesis for The Degree of Doctor of Philosophy

Choreographies and Cost Semantics for Reliable Communicating Systems

Alejandro Gómez-Londoño

CHALMERS

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2023

Choreographies and Cost Semantics for Reliable Communicating Systems
Alejandro Gómez-Londoño
ISBN 978-91-7905-872-2

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5338
ISSN 0346-718X

Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed by Chalmers Reproservice,
Gothenburg, Sweden 2023.

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 results that aim to increase the reliability of communicating systems. First, we implement a choreography language that can, by construction, only describe deadlock-free systems. Second, we develop a cost semantics to prove programs free of out-of-memory errors. Lastly, we improve both results by using novel semantic approaches that strengthen key theorems and facilitate further proof development. All 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!*

List of publications

This thesis is based on the following publications:

Paper I. J. Å. Pohjola, **A. Gómez-Londoño**, J. Shaker, M. Norrish
"Kalas: A Verified, End-to-End Compiler for a Choreographic Language"
13th International Conference on Interactive Theorem Proving, ITP 2022.

Paper II. **A. Gómez-Londoño**, M. O. Myreen, J. Å. Pohjola
"Dancing in a forest of interactions: An interaction tree-based semantics for choreographies"
Draft paper.

Paper III. **A. Gómez-Londoño**, J. Å. Pohjola, H. T. Syeda, M. O. Myreen, Y. K. Tan
"Do You Have Space for Dessert? a verified space cost semantics for CakeML programs"
Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA 2020.

Paper IV. **A. Gómez-Londoño**, M. O. Myreen
"A flat reachability-based measure for CakeML's cost semantics"
33rd Symposium on Implementation and Application of Functional Languages, IFL 2021.

Contents

Introduction

Software-based communicating systems are everywhere in today's society. They are at the core of stock exchanges, air traffic controls, and power grids while also being prevalent in our daily lives through phones, vehicles, and even doorbells. Moreover, the scale and relevance of communicating systems are only increasing for critical and everyday tasks.

It is well known that software systems can go wrong, especially when communications are involved. The complexity that arises from coordinating the communication of multiple components can introduce unpredictable and hard-to-spot errors. Given the ubiquity of communicating systems, failures might lead to the loss of capital or, in extreme cases, lives. Therefore, mitigating errors in such systems is crucial for their continuing adoption and growth.

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 the receiver from making any progress.

(P2) A program suddenly runs out of memory and stops sending messages halting the rest of the system.

This thesis comprises four papers tackling problems **P1** and **P2**. Our works present formal approaches to safeguard programs against *deadlock*, with improved results for usability and scalability. **Paper I** presents work on a choreography language to define communicating systems that, by construction, can not deadlock due to communication mismatches and therefore addresses **P1**. **Paper II** proposes an alternative semantics for our choreographic language with a more robust notion of deadlock freedom and semantic correspondence. **Paper III** develops a cost semantics for CakeML [17] programs, enabling space-bound reasoning to prevent occurrences of **P2**. **Paper IV** improves our

previous cost semantics by simplifying how space is measured, reducing the complexity of proving the absence of **P2**.

The following sections give a more in-depth explanation of choreographies and cost semantics, as well as highlight the main contributions of our papers and how they tackle challenges P1 and P2. This introductory chapter concludes with an outline of future work, including how the methods presented can be combined to achieve even stronger guarantees.

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, 11, 12, 13, 14]. 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 criterion 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) pay-

ment 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.

Choreographies, as introduced by Carbone et al. [6, 7], are languages for defining communicating systems that prevent communication mismatches (like the one in Example 4). First, a choreography defines a global protocol for all the components of the system alongside their internal computations; this allows for entirely automated translations—through a process called *projection*—that avoids mismatches between a specification and its implementation. Second, inconsistent protocols are ruled out by the language semantics (which by construction can not get stuck) and an accompanying notion of "projectability" which guarantees that choreographies behave the same as their projections. Finally, these two mechanisms ensure that all projectable choreographies generate protocol-compliant code that does not get stuck and can always progress—a property 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.

Example 6.

BUYER1

```
send(SELLER,item)
price = receive(SELLER)
send(BUYER2,price)
decision = receive(BUYER2)
If decision ≡ "buy"
Then
  select(SELLER,T)
  payment= "<cc info>"
  send(SELLER,payment)
  receipt = receive(SELLER)
Else
  select(SELLER,F)
```

SELLER

```
item = receive(BUYER1)
price = lookupPrice(item)
send(BUYER1,price)
Choice BUYER1 of
 T =>
 payment = receive(BUYER1)
 receipt = "Purchase No: <XYZ>"
 send(BUYER1,receipt)
 F => // Do nothing
```

BUYER2

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

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 communication is consistent and thus free of deadlocks. However, communication guarantees are only a part of what makes a communicating system truly reliable.

Unfortunately, the implementation of a consistent protocol can still go wrong if computation errors are present [8]. Steps in-between communications can fail in ways that affect the system as a whole. For example, a function that fails to terminate might stop a component from sending or receiving further messages, possibly causing a deadlock state. Moreover, incorrect results from a function (e.g., `-1`, `0/0`, `NaN`) can lead to failures down the line causing the system to reach a failure state. It is a non-trivial task to prevent these types of failures, and it goes well beyond what informal protocols typically consider. Nonetheless, it is paramount for a reliable communicating system to mitigate such errors as much as possible.

To safeguard against errors in local computations, one must ensure the correctness of individual component implementations. Various methods and formal approaches exist to show the correctness of a source program. Furthermore, the source program's properties and behaviors must also translate to target representations to add weight to any correctness claims. However, source programs (that can run on a computer) are ultimately compiled into machine code, making correctness results for intermediate representations inherently incomplete. A property that is preserved from source to machine code is known as "end-to-end" and is amongst the strongest guarantees a program can have.

The CakeML verified stack [17] allows end-to-end proofs of correctness for SML-like source programs, making it the perfect target representation for a reliable communicating system.

Paper I presents the implementation of a choreography language and, to the best of our knowledge, the first mechanized end-to-end proof of the 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.

This work was presented at the 13th International Conference on Interactive Theorem Proving (ITP) in 2022 [15].

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.

Paper II enhances our choreographic language semantics by adding a more flexible version based on interaction trees [18]. This new approach enhances our previous small-step semantics by clearly distinguishing between components' local behavior and global interactions. As a result, it becomes easier to prove semantic correspondence in a verified choreographic compiler–like Kalas. The main contributions of this paper are:

- A definition of an interaction tree-based semantics for Kalas with a clear separation between local behaviors and global interactions.
- A proof of correctness for Kalas's projection function showing semantic equivalence between source and target programs.
- A general interaction tree-based definition of deadlock-freedom, along with proof that it holds by design for Kalas' programs. Moreover, this definition also holds for our target language, thanks to projection correctness.

This work is currently being prepared for submission.

Statement of Contribution. I am this paper's main author of semantics definitions, properties, and proofs. This work was in collaboration with Magnus Myreen and Johannes Åman Pohjola.

Cost Semantics

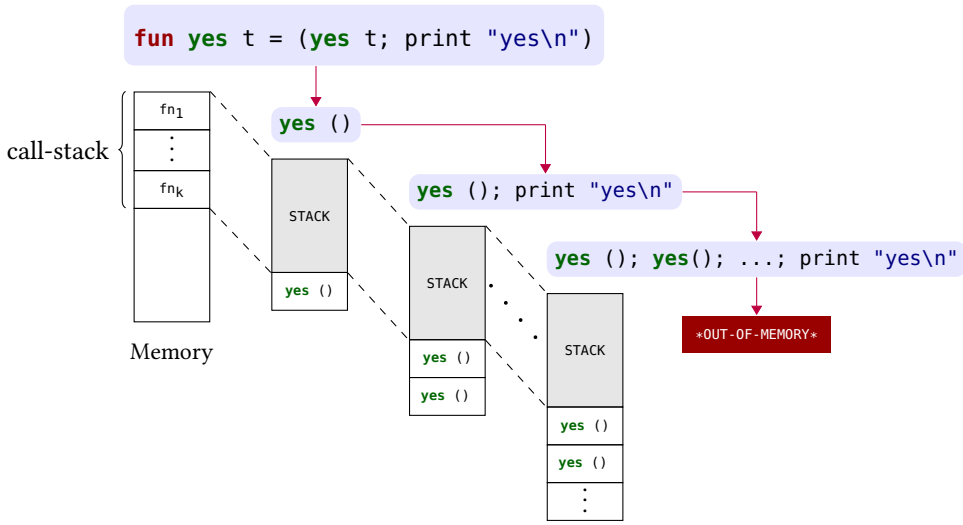
A program's space consumption is as relevant to its utility as the functional correctness of its implementation. Despite appearances to the contrary in high-level language semantics, programs only have a finite amount of memory available to them; the use of this resource has a direct impact on whether 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 7.

```
1 let
2   fun yes t = (print "yes\n"; yes t)
3 in yes ()
```

Figure 1



The code in Example 7 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 called function returns. Therefore, if a program recursively calls a function without giving a result, as seems to be the case in Example 7, 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 in the call-stack, as nothing is left of the current function. Hence, the original caller function can be directly resumed—a technique called tail-recursion and present in most compilers. 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 7) 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 of whether a given program might run 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 8.

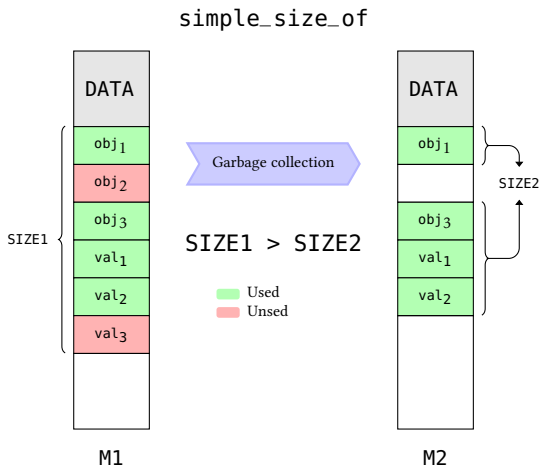
```
// Using a list
let
  fun bar1 0 = []
    | bar1 n = foo n :: bar1 (n - 1)
in foldl (op +) 0 (bar1 10000000)

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

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 at any point 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

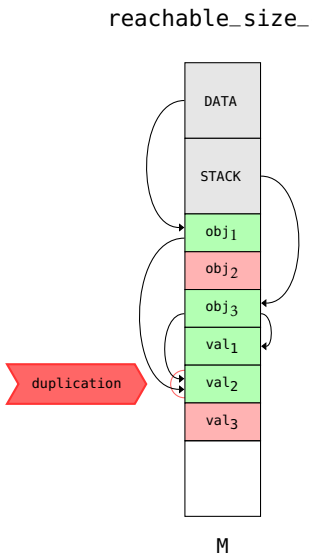


A cost function essentially measures the size of the data used by the program; thus, a simple implementation (`simple_size_of`) could just add the sizes of all objects currently in memory. However, in the presence of a garbage collector (GC), not all objects in memory are relevant for the measurement 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]. Furthermore, most results target approximated measurements, either as a function of the inputs (asymptotic bounds) or a conservative over-approximation (upper bounds).

Figure 3



Paper III presents a cost semantics that can be used to prove that a given CakeML program does not run out of memory for a specific amount of memory, which, to the best of our knowledge, is the first time this result has been obtained for a garbage-collected language. The approach presented addresses common pitfalls in the following ways:

- The cost semantics is defined in 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.

This work was presented at the conference in Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) in 2020 [10].

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.

Paper IV in this thesis presents the development of an alternative cost function for CakeML, which exhibits compositional properties that greatly simplify space reasoning. This new cost function is a drop-in replacement for our previous results; however, empirical results show a reduction of LOC in proof files of close to 30% when the new formulation is used. The main contributions of this paper are:

- A new reachability-base cost function for CakeML with rich compositional properties that aid reasoning.
- A proof of equivalence to existing cost functions in CakeMl, with a direct end-to-end soundness result.

This work was presented at the 33rd Symposium on Implementation and Application of Functional Languages (IFL) in 2021 [9].

Statement of Contribution. For this paper, I was involved in all aspects of the development. This involved contributing to designing and implementing the cost function, formalization of crucial properties, and the proof of equivalence. This paper was done in close collaboration with Magnus Myreen.

Future Work

One clear area for cost semantics in CakeML to develop is proof automation with a focus on scalability. While proofs of space safety tend to be quite involved and labor intensive, common patterns and proof techniques are often used with case-by-case modifications that could be captured in proof tactics or automation. Furthermore, subsets of CakeML, or even completely new languages built on the CakeML stack, could impose restrictions on programs to make space safety decidable or at least more tractable. Developing these (or alternative) methods could help verify the space safety of larger programs without significantly augmenting cost or complexity.

There are also a number of natural directions to develop Kalas further. For

instance, adding process creation to Kalas' syntax and semantics could allow the implementation of more expressive communicating systems. Similarly, proving live-lock freedom for Kalas programs would significantly improve the language's end-to-end guarantees. More generally, we believe Kalas is a sandbox on which more and more features (e.g., typed communication, general projectability, message optimizations) could be added without changing much of its core structure.

Finally, the works presented in this thesis aim to improve the level of reliability that can be achieved for communicating systems. As such, choreographies and cost semantics can be combined to achieve even stronger assurances for programs. Concretely, deadlock-freedom guarantees provided by choreographies assume that each component is present and functions correctly. Space-bound reasoning and other guarantees provided by CakeML ecosystem [3, 16]—Kalas' target language—could be connected to projected programs to improve their reliability. By doing this, projected programs can be shown, with proof, not to stop responding due to an out-of-memory error or correctness bugs, significantly reducing the number of assumptions needed to rule out *deadlock* completely.

Thesis Outline

The rest of this thesis consists of four chapters. Chapter 1 introduces Kalas and our choreography semantics—**Paper 1**. Chapter 2 expands the semantics of Kalas with an interaction tree-based version of the semantics—**Paper 2**. Chapter 3 presents a cost semantics for CakeML with tight bounds—**Paper 3**. Finally, Chapter 4 improves the previous cost semantics for CakeML to facilitate reasoning about tight space bounds—**Paper 4**.

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] 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`.

[7] 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.

[8] 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.

[9] A. Gómez-Londoño and M. O. Myreen. A flat reachability-based measure for cakeml's cost semantics. In *33rd Symposium on Implementation and Application of Functional Languages, IFL 2021, Nijmegen, The Netherlands, September 1-3, 2021*, pages 1–9. ACM, 2021. doi: 10.1145/3544885.3544887. URL https://doi.org/10.1145/3544885.3544887.

[10] A. Gómez-Londoño, J. Å. Pohjola, H. T. Syeda, M. O. Myreen, and Y. K. Tan. Do you have space for dessert? a verified space cost semantics for cakeml programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):204:1–204:29, 2020. doi: 10.1145/3428272. URL https://doi.org/10.1145/3428272.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] J. Å. Pohjola, A. Gómez-Londoño, J. Shaker, and M. Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In J. Andronick and L. de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ITP.2022.27. URL https://doi.org/10.4230/LIPIcs.ITP.2022.27.

[16] 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.

[17] 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.

[18] L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi: 10.1145/3371119. URL `https://doi.org/10.1145/3371119`.

1

Kalas: A Verified, End-to-End Compiler for a Choreographic Language

Johannes Åman Pohjola
Alejandro Gómez-Londoño
James Shaker
Michael Norrish

International Conference on Interactive Theorem Proving (ITP) 2022 [1]

Abstract. Choreographies are an abstraction for globally describing deadlock-free communicating systems. A choreography can be compiled into multiple endpoints preserving the global behavior, providing a path for concrete system implementations. Of course, the soundness of this approach hinges on the correctness of the compilation function. In this paper, we present a verified compiler for Kalas, a choreographic 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. 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] This thesis version of the paper exhibits slight differences from the published version.

1.1 Introduction

In recent years, advances in the fields of concurrency theory and systems verification have taken us closer to the idea of a truly correct communicating system. The former abounds with beautiful high-level specification formalisms and reasoning techniques for communicating systems. At the same time, the latter provides detailed correctness proofs of the low-level computing infrastructure (e.g., compilers, language runtimes, and operating systems) needed to implement them. There is then much to be gained by joining these worlds. In particular, high-level descriptions of communicating systems — along with their guarantees — could be propagated down to low-level implementations to create an end-to-end result. One promising approach is choreographic programming, which at a high level describes communicating systems while providing by-construction guarantees.

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 traditional approach of writing separate programs for every `Alice` and `Bob`, 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. Furthermore, through a procedure called *endpoint projection* choreographies can be compiled into separate programs for each endpoint, such that their parallel composition implements the global behaviour.

In this paper, we present a compiler for our choreographic language, Kalas, with a machine-checked, end-to-end proof of correctness. That is, we create an environment based on the HOL4 interactive theorem prover [20] where programmers can write choreographies, and then have the system automatically generate executable code for each endpoint, along with a proof of its correct compilation into machine-code.

Our compiler is structured into five phases, illustrated in Figure 1.1, with associated correctness result for each. The first step is endpoint projection, where the global choreography is projected into a parallel composition of

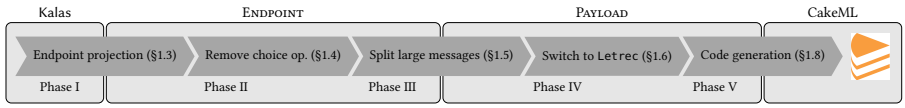


Figure 1.1: Compilation Steps and Intermediate Languages

sequential programs implementing each endpoint, expressed in a process algebra we call Endpoint. Second, the Endpoint operators for branch selection are encoded with more primitive operators. 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. A fourth step compiles Kalas's fixpoint operator (with substitution semantics) into recursive function definitions (with environment semantics), to align better with functional programming idioms. The final compilation phase compiles Payload's endpoints to CakeML [10], a sequential, functional programming language with a verified compiler, giving us semantics preservation down to the machine code.

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

As a convenient byproduct of CakeML's FFI modelcode is parameterised on primitives for sending and receiving messages, making it communication-backend agnostic. Thus, the same CakeML code can be used irrespective of whether the communication happens via (say) TCP/IP, MPI, or IPC, as long as these actions have the same semantics as the corresponding Payload primitives. Like other choreographic 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.

As a proof-of-concept of our approach, we implemented a filter choreography and executed the generated code using an IPC communication backend 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 [6], that this backend is up to the task, even though we do not connect our proofs with the seL4 proofs.

This paper's main contributions are:

- the definition and verification of an end-to-end choreographic compiler, including:
- the proof of *endpoint projection*'s correctness w.r.t. Kalas's asynchronous semantics; and
- the implementation of a proof-of-concept choreography on top of seL4/CAmkES.

All definitions and proofs in this paper are mechanised in HOL4 [20] and available online.[2]

1.2 Kalas: A Choreographic Language

In this section we introduce our choreographic language, Kalas. To build an intuition for how choreographies operate, consider a common situation in component-based systems: a producer wishes to send a stream of messages to a consumer, but the consumer can only receive messages of a certain form. A filter that discards malformed messages is inserted.

Example 9 (Message filter - Choreography)**.**

```
1.  while(true) do
2.    let v@producer = next_msg() in
3.    producer.v -> filter.temp;
4.    let test@filter = test(temp) in
5.    if test@filter then
6.      filter -> consumer[T];
7.      filter.temp -> consumer.v
8.    else filter -> consumer[F]
```

We assume a function `next_msg` to obtain the next message, which is then stored in the `producer`'s local variable v (line 2). The `producer` 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), we inform the consumer that a message is coming (line 6), and forwards the contents of *temp* to the `consumer` (line 7). Otherwise, we inform the `consumer` that a message was dropped (line 8).

[2] `https://github.com/CakeML/choreo/`. Instructions for reviewers here: `https://www.cse.unsw.edu.au/~z3528312/itp2019/artifact.pdf`

This example highlights two important features: a choreography captures 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 local computation, e.g., `test` above. This allows individual endpoints to be translated into complete sequential programs. Second, communication mismatches are impossible by construction: if no message is forthcoming, the `consumer` will never be stuck waiting for one.

1.2.1 Syntax and Semantics

Kalas is similar to Core Choreographies (CC) [1], but features arbitrary local computation and asynchronous communication. The main datatype under consideration in our choreography language is strings or, to be precise, finite sequences of bytes. Strings are used as endpoint names (p_i), variable names (v_i), process variables (X), and as the concrete data that gets bound to variables and transmitted between endpoints (d). The use of strings as the value represents a separation of concerns: after marshalling and unmarshalling, local computations have full access to HOL's strongly typed language, but the choreography language is only concerned with data as it is really transmitted, namely as strings. The booleans (ranged over by b) are written `T` and `F`. When we use booleans where strings are expected, we tacitly identify `T` with [0x01], and `F` 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 (Kalas syntax)**.** Choreographies *in Kalas, ranged over by C, are inductively defined by the grammar*

$$
\begin{array}{lll}
C ::= & p_1.v_1 \twoheadrightarrow p_2.v_2; C & \textit{(com)} \\
 & \textbf{if } v@p \textbf{ then } C_1 \textbf{ else } C_2 & \textit{(if)} \\
 & \mu X.\ C & \textit{(fix)} \\
 & \mathtt{0} & \textit{(nil)} \\
 & p_1 \twoheadrightarrow p_2[b]; C & \textit{(sel)} \\
 & \textbf{let } v@p = f(\widetilde{v}) \textbf{ in } C & \textit{(let)} \\
 & X & \textit{(var)}
\end{array}
$$

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)* is the empty choreography. *(let)* performs local computation, taking all values bound to the variables $\widetilde{v}$ at endpoint p and applies them as arguments to the function f. The result is then stored in v. Note that we do not commit to any particular syntax for functions; rather, f is a function in the meta-language in which Kalas 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 useful 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 be "sufficiently code-like"—otherwise, code generation will fail. This excludes, for example, functions that use Hilbert choice, sets or quantifiers.

Finally, *(fix)* supports choreographies with infinite behaviour. These can be unfolded, taking e.g. $\mu X.\ p_1 \twoheadrightarrow p_2[b]; X$ to $p_1 \twoheadrightarrow p_2[b]; \mu X.\ p_1 \twoheadrightarrow p_2[b]; X$. The above example also illustrates the only use of *(var)*: as a placeholder for fixpoint unfolding. The **while** loop used in Example 9 is syntactic sugar for *(fix)*.

We will use $\mathrm{fv}(C)$ to refer to the free variables of a choreography C, where each variable is paired with the name of the process that owns the variable. The binding operators are **let** and $p_1.v_1 \twoheadrightarrow p_2.v_2; C$, where $(v1, p1)$ is considered free and $(v2, p2)$ is considered bound.

$$\textsc{Com}\ \frac{s\,(v_1,p_1) = d \quad p_1 \neq p_2}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C \xrightarrow[\epsilon]{p_1.v_1 \twoheadrightarrow p_2.v_2} s[(v_2,p_2) := d] \rhd C}$$

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

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

Table 1.1: Kalas semantics: communication rules. The function $\mathrm{wv}(\alpha)$ returns the variable (if any) that is modified by α.

The operational semantics is inductively defined, with some sample rules given in Table 1.1. Transitions are labeled to indicate both the action being performed (upper α), and the trace (lower l) of deferred asynchronous actions. We explain the latter mechanism below. 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: $s(v, p)$, if defined, denotes the value bound to v in p's binding environment. In Hol, we use option types in the range to encode this partiality; much of the following presentation elides the logic's special handling of this (*e.g.*, the `Some` and `None` constructors).

Kalas uses non-blocking, asynchronous communication. Hence, a sender process should be able to perform further actions before the message has arrived at the receiver. The semantics captures this by allowing an action α to occur before other interactions, provided only the sender process is present in α. A trace of every action that was skipped over is kept, to ensure the consistency between asynchrony and concurrency rules. This trace is used in the rule for **if**, which requires that both branches defer the same actions, though not necessarily in the same order. 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 caused by the guard.

We prove that the resulting semantics is locally confluent, which will turn out to be immensely important for taming the proofs. As a sanity check of our rather involved labels, we also show completeness with respect to a similar semantics (not shown here) with structural congruence instead of swapping rules.

1.3 Endpoint Projection

The first phase of our compiler is *endpoint projection*, where we translate Kalas into Endpoint, our first intermediate language.

Continuing with Example 9, when we apply endpoint projection to the producer-consumer-filter choreography (*PCF*), we obtain $[\![PCF]\!]_{\mathsf{E}} = P \mid C \mid F$, comprised of the following endpoints running in parallel:

Example 10 (Message filter — Endpoint)**.**

P (Producer)

$\mu X.$
  let v = `next_msg()` **in**
  send v **to** `filter`. X

C (Consumer)

$\mu X.$
  `filter` **chooses**
    `T` : **receive** v **from** `filter`. X
    or
    `F` : X

F (Filter)

$\mu X.$
  receive *msg* **from** `producer`.
  let v = `test`(*msg*) **in**
  if v **then**
    choose `T` **for** `consumer`.
    send *msg* **to** `consumer`. X
  else choose `F` **for** `consumer`. X

Here the (Filter) endpoint receives a message from the producer and, depending on the output of `test`, communicates its choice of branch to the consumer. Conversely, the (Consumer) decides based on the filter's choice whether it should await a message or whether the message was dropped. Finally, the (producer) obtains a message and sends it to the filter. Note that no branching or choice is required in the producer, since it behaves the same whether `test` succeeds or not.

1.3.1 Endpoint: Syntax and Semantics

Endpoint inherits many design decisions from Kalas, but splits unitary Kalas systems into two layers: the endpoint layer is purely sequential, and the *network* layer is a parallel composition of endpoints, each with its own name, queue and binding environment.

A *queue* q is a function $\mathsf{string} \to \mathsf{string}^*$. The value $q(p)$ is the sequence of messages, from first to last, received from *process* p but not yet read. Let $q + (p, a)$ be q with a appended to the end of $q(p)$, and $q - p$ be q with the first element of $q(p)$ removed; if $q(p)$ is empty, $q - p$ is undefined. An *environment* e is a partial function from variable names to values.

Definition 2 (Endpoint syntax)**.**

P, Q	:=	**send** v **to** $p.P$	*(output)*
		receive v **from** $p.P$	*(input)*
		choose b **for** $p.P$	*(internal choice)*
		p **chooses** T : P **or** F : Q	*(external choice)*
		if v **then** P **else** Q	*(if)*
		let $v = f(\widetilde{v})$ **in** P	*(let)*
		$\mu X.P$	*(fix)*
		X	*(var)*
		$\mathbf{0}$	*(nil)*

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

Table 1.2 shows three representative rules from Endpoint's operational semantics. **send** v **to** $p.P$ 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. **receive** v **from** $p.P$ 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 is blocked until one arrives. Similarly, **choose** b **for** $p.P$ represents an endpoint ready to tell process p that it has chosen the b-branch. The corresponding IntChoice rule (elided) interacts with Enqueue to add the choice to b's message queue. p **chooses** $\mathsf{T} : P$ **or** $\mathsf{F} : Q$ represents a process waiting for p to communicate its choice of branch. If it finds a T from p in the queue, it proceeds as P; if it finds something else from p, it proceeds as Q.

$$
\textsc{Send}\ \frac{e\ v\ =\ d \quad p_1 \neq p_2}{(p_1, e, q) \triangleright \mathbf{send}\, v\, \mathbf{to}\, p_2.P \xrightarrow{p_1 \to p_2 : d} (p_1, e, q) \triangleright P}
$$

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

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

Table 1.2: Endpoint semantics: communication rules.

1.3.2 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

if Alice@v **then** Bob.v -> Alice.v **else** Alice.v -> 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?

We need a projectability criterion that rules out such degenerate cases. 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:

$$\begin{array}{l}\textbf{if}\ \text{Alice}@v\ \textbf{then}\ \text{Alice} \twoheadrightarrow \text{Bob}[\mathtt{T}]; \text{Bob}.v \twoheadrightarrow \text{Alice}.v\\ \textbf{else}\ \text{Alice} \twoheadrightarrow \text{Bob}[\mathtt{F}]; \text{Alice}.v \twoheadrightarrow \text{Bob}.v\end{array}$$

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

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

Fixpoints motivate some additional projectability criteria: (i) orphan *(var)* statements are not allowed, and (ii) the projection of a *(fix)* statement for endpoints that do not appear in its body should be $\mathtt{0}$ (otherwise, the compiler introduces divergence). To enforce these requirements, we use a *fixpoint context* γ, a partial function from process names to sets of endpoint names that keeps track of which endpoints occur in the body of each *(fix)* statement.

We define a single partial function pr that given an endpoint name, a fixpoint context, and a choreography, returns an endpoint (its projection), if it exists.

Definition 4 (Projection and projectability)**.** *A choreography* C *is* projectable *if for all* $p \in \mathsf{procs}(C)$, $\mathsf{pr}_p(\epsilon, C)$ *is defined. The* projection *of the endpoints* $\widetilde{p}$ *from a choreography* C *with state* s *is defined as* $[\![s \triangleright C]\!]_{\mathsf{E}}^{\widetilde{p}} = \Pi_{p_i \in \widetilde{p}}.\ (p_i, s\downarrow_{p_i}, \epsilon) \triangleright \mathsf{pr}_{p_i}(\epsilon, C)$ *where* Π *denotes iterated parallel composition,* $s\downarrow_p$ *denotes* $\lambda v.s(p, v)$*,* ϵ *is an empty fixpoint context, and* pr *is defined inductively by the equations in Table 1.3.* $[\![s \triangleright C]\!]_{\mathsf{E}}$ *abbreviates* $[\![s \triangleright C]\!]_{\mathsf{E}}^{\mathsf{procs}(C)}$*.*

1.4 Refining Choice

In Phase II, we implement Endpoint's choice primitives using send and receive actions. This simplifies reasoning about later compilation phases and the

implementation of communication backends, which only need to consider two message-passing primitives instead of four.

After refining choice from the parallel composition $P \mid C \mid F$ in Example 10, we obtain $[\![P \mid C \mid F]\!]_{\mathsf{C}} = P \mid C' \mid F'$, where the producer P is unchanged because it uses no choice constructs. The filter and consumer are compiled as follows:

Example 11 (Message filter — Refining choice)**.**

F' (Filter)

$\mu X.$
  receive *msg* **from** `producer`.
  let *test* = `test`(*msg*) **in**
  if *test* **then let** v = `T` **in**
    send v **to** `consumer`.
    send *msg* **to** `consumer`. X
  else let v = `F` **in**
    send v **to** `consumer`. X

C' (Consumer)

$\mu X.$
  receive v **from** `filter`.
  if v **then**
    receive v **from** `filter`. X
  else X

The phase is mostly straightforward: internal choice is encoded as sending a boolean value, and external choice is encoded as receiving a value, storing it in a temporary variable v, then branching on it using **if**.

Definition 5 (Phase II)**.** *The compilation function is homomorphic on all operators except internal and external choice, where it is defined as follows for any v not free in P, Q:*

$$[\![\textbf{choose}\, b\, \textbf{for}\, p.P]\!]_{\mathsf{C}} = \textbf{let}\ v = (\lambda x.b)\epsilon\ \textbf{in send}\, v\, \textbf{to}\, p.[\![P]\!]_{\mathsf{C}}$$

$$[\![p\, \textbf{chooses}\, T :\ P\, \textbf{or}\, F : Q]\!]_{\mathsf{C}} = \textbf{receive}\, v\, \textbf{from}\, p.(\textbf{if}\ v\ \textbf{then}\ [\![P]\!]_{\mathsf{C}}\ \textbf{else}\ [\![Q]\!]_{\mathsf{C}})$$

Since v is not used further in the continuation, it can be reused for subsequent choice encodings, meaning that in practice, a single fresh name suffices.

The design of the Endpoint semantics anticipates this compilation phase, by allowing type confusion between boolean values and string values. This feature, which may seem otherwise undesirable, makes branch selection messages indistinguishable from other messages. This makes the difference between N_{E} and $[\![N_{\mathsf{E}}]\!]_{\mathsf{C}}$ unobservable by other processes.

1.5 Splitting Large Messages

In both Endpoint and our source language, transmitting a message of arbitrary size is a single atomic operation, whether it carries one bit or one terabyte of information. This is convenient for the programmer, but doesn't reflect how real communication protocols work.

Our second compiler phase introduces a protocol that divides long messages into chunks (see Figure 1.2), accounting for the fact that real communication protocols have bounds on message size, without burdening the application programmer with the details.

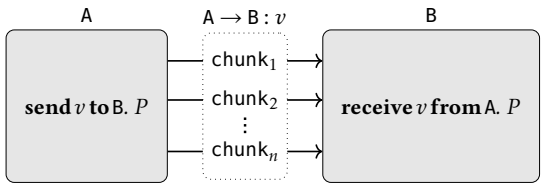


Figure 1.2: Messages split into `chunks`

For these purposes, we introduce another intermediate representation, Payload, which is similar to Endpoint except messages have a fixed size. It turns out that this compiler phase is more proof-relevant than compiler implementation-relevant. Syntactically, the compilation function $[\![\cdot]\!]_{\mathsf{C}}$ from Endpoint to Payload is essentially the identity function. Semantically, send and receive actions are no longer atomic, which leads to a combinatorial explosion in the number of possible interleavings. The associated proof complications are largely mitigated by observing that the target terms are always locally confluent.

Payload is parameterised by a *payload size* $\sigma > 0$. Unlike in previous languages where messages can have arbitrary size, here messages in transit must be exactly $\sigma + 1$ bytes long. Longer messages are transmitted in chunks, and shorter messages are padded; the extra byte encodes the bookkeeping necessary to realise this. In particular, we must track whether a given chunk ends a message, or whether it will be continued in future messages.

Definition 6 (Payload syntax, I)**.** *The syntax of Payload is obtained by removing endpoint, input, output and choice from Endpoint, and adding:*

$$(p, e, \eta, q) \triangleright P \quad \textit{(endpoint)} \qquad \mathbf{send}\, v_n\, \mathbf{to}\, p.P \quad \textit{(output)}$$

$$\mathbf{receive}\, v\, \mathbf{in} \langle d \rangle\, \mathbf{from}\, p.P \quad \textit{(input)}$$

The message-splitting aspect of Payload's semantics is embodied here: the new input and output prefixes record how far along in a transmission we are. Hence $\mathbf{send}\, v_n\, \mathbf{to}\, p.P$ will send the value of v to p, starting from the n-th byte, divided into as many chunks as necessary, one chunk at a time. Similarly $\mathbf{receive}\, v\, \mathbf{in} \langle d \rangle\, \mathbf{from}\, p.P$ will receive chunks from p, recording every

intermediate chunk in the temporary buffer d. When a `final` chunk arrives, all received chunks are concatenated and bound to the variable v.

The state component η is a closure environment; we will discuss it in Section 1.6.1, where the operators that need it are introduced. For space reasons, readers interested in the operational semantics are referred to the formalisation.

1.6 Introducing Closures

Finally, before we transition from Payload to CakeML, we introduce closures. That is, we translate all instances of the fixpoint operator μ into a **letrec** primitive, to better match CakeML's representation of recursive functions. Though CakeML supports global, update-able variables (SML's `ref` types), reasoning is much simpler if one remains "purely functional", and uses parameters with closures and local, immutable bindings. Thus: variables written to in the fixpoint body (ultimately from the Kalas source) must be made function parameters.

Why not just put **letrec** in the source language, if we have to add it later anyway? Briefly, it becomes technically complicated to maintain a consistent view of the global environment in the presence of out-of-order execution. Fixpoint semantics do not need environments. See Section 1.9 for a comparison with related approaches.

1.6.1 Closures: syntax and semantics

Recall from Section 1.5 that endpoint states contain a closure environment η. It is a mapping from function names to *closures*. Closures are triples $(e, \eta, \lambda\widetilde{x}.P)$, where: e, η are the local variable environments and closure environments, respectively; $\widetilde{x}$ is the function's parameters; and P is the function body. Note that closures and environments are mutually recursive.

Definition 7 (Payload syntax, II)**.** *These augment the operators from Definition 6.*

$$\textbf{letrec}\ F(\widetilde{d}) = P \quad \textit{(letrec)} \qquad\qquad F(\widetilde{v}) \quad \textit{(call)}$$

(letrec) and *(call)* are function definitions and function calls, respectively. They are similar to *(fix)*, but use environment semantics instead of substitution semantics. Note that **letrec** has no continuation; instead, the defined function

will be called immediately. This suffices for our purposes, which is to use **letrec** to encode μ.

1.6.2 Compilation

To compile the fixpoint operator into the letrec operator, the basic idea is the obvious one: a fixpoint binder μX becomes a recursive function definition, and a process variable X becomes a function call. But what arguments should we give the function?

To prepare for compilation to CakeML, the main goal is to make sure we use the constructs that mutate variables (let and input) consistently with functional programming idioms. But we have a second, conflicting goal. To simplify proofs, we want the local variable environment of target terms to be identical to the global environment of their source terms. The following table illustrates the options we considered:

Source
let $x, y = \ldots$ **in** μX. **let** $x = \mathtt{f}(x, y)$ **in** **send** x **to** p.**receive** z **from** p.X

Target I	Target II	Target III
let $x, y = \ldots$ **in** **letrec** $X(x, y, z) =$ **let** $x = \mathtt{f}(x, y)$ **in** **send** x **to** p. **receive** z **from** p. $X(x, y, z)$	**let** $x, y = \ldots$ **in** **letrec** $X(x, z) =$ **let** $x = \mathtt{f}(x, y)$ **in** **send** x **to** p. **receive** z **from** p. $X(x, z)$	**let** $x, y = \ldots$ **in** **letrec** $X(x) =$ **let** $x = \mathtt{f}(x, y)$ **in** **send** x **to** p. **receive** z **from** p. $X(x)$

Target I represents the simplest compilation strategy that could possibly work: every program variable becomes a parameter of every function. This, however, is rather wasteful: y is never modified within the function, so there's no need to pass it around; z is modified in the body, but not subsequently read, so there is no need to remember its value between calls.

Taking all this into account would yield Target III, which is to take as function parameters only those variables that may be read before they are written to within a fixpoint's body. Unfortunately, this is not compatible with our second goal above: while the source term retains the value of z between subsequent fixpoint unfoldings, Target III will restore z to its value at the point of X's definition at each recursive call.

As a compromise, we opt for Target II: the function parameters are the variables that may be written to within the body of the fixpoint expression. To

this end, we let the function $\mathrm{wv}(e)$ return all variables that are modified (by **let** or receive) in e.

Definition 8 (Phase IV)**.** *The compilation function $[\![\cdot]\!]_{\mathsf{F}}^{\gamma}$ is homomorphic on all operators except: letrec and call, where it is undefined; and fix and val, where it is*

$$[\![\mu X.P]\!]_{\mathsf{F}}^{\gamma} = \textbf{letrec}\ X(\mathrm{wv}(P)) = [\![P]\!]_{\mathsf{F}}^{\gamma[X:=\mathrm{wv}(P)]} \qquad [\![X]\!]_{\mathsf{F}}^{\gamma} = X(\gamma(X))$$

In the above, γ is a partial function from process variables to lists of local variables.

A further minor complication is that a variable can be used as a function argument before its definition. We could add support for optional arguments, but since CakeML has no such feature we would eventually have to compile them away. Our fix is that before we apply $[\![X]\!]_{\mathsf{F}}$, we add a prelude to each endpoint that initialises all variables to a default value.

1.7 Compiler Correctness

In this section, we discuss the compiler correctness theorem connecting Kalas to Payload with `Letrec`, and its proof.

1.7.1 Theorem Statement

Let $[\![\cdot]\!]^{\widetilde{p}}$ denote the composition $[\![\cdot]\!]_{\mathsf{E}}^{\widetilde{p}} \circ [\![\cdot]\!]_{\mathsf{C}} \circ [\![\cdot]\!]_{\mathsf{P}} \circ [\![\cdot]\!]_{\mathsf{F}}^{\epsilon}$ and let $[\![C]\!] = [\![C]\!]^{\mathsf{procs}(C)}$. We prove weak operational correspondence up-to strong bisimilarity (denoted $\dot{\sim}$) for $[\![\cdot]\!]^{\widetilde{p}}$:

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

1. *(Operational completeness) If $s \triangleright C \implies s' \triangleright C'$ then there exist s'', C'', N_{F} such that $s' \triangleright C' \implies s'' \triangleright C''$ and $[\![s \triangleright C]\!] \implies N_{\mathsf{F}}$ and $N_{\mathsf{F}} \dot{\sim} [\![s'' \triangleright C'']\!]^{\mathsf{procs}(C)}$*
2. *(Operational soundness) If $[\![s \triangleright C]\!] \implies N_{\mathsf{F}}$ then there exist s', C', N_{F}' such that $N_{\mathsf{F}} \implies N_{\mathsf{F}}'$ and $s \triangleright C \implies s' \triangleright C'$ and $N_{\mathsf{F}}' \dot{\sim} [\![s' \triangleright C']\!]^{\mathsf{procs}(C)}$*

Here $\implies$ over networks denotes $\xrightarrow{\tau}{}^{\star}$, and $\implies$ over choreographies denotes $(\bigcup_{a,l} \xrightarrow[l]{a})^{\star}$. Our presentation of *operational completeness* requires a catch-up

transition because projectability is not, in general, preserved by reduction. However, any non-projectable choreography reachable from a projectable choreography can always reduce to a projectable choreography.

One important consequence of Theorem 4 is that the compiler output is deadlock-free:

Theorem 2 (Network-level deadlock-freedom)**.** *If C is a projectable choreography, and* $\mathrm{fv}(C) \subseteq \mathrm{dom}(s)$*, and* $[\![s \triangleright C]\!] \implies N_{\mathsf{F}}$*, then either all endpoints in* N_{F} *are* `Nil`*, or there exists* N'_{F} *such that* $N_{\mathsf{F}} \xrightarrow{\tau} N'_{\mathsf{F}}$

1.7.2 On the proofs

As the reader may expect, we prove soundness and completeness separately for each compilation phase before composing the theorems. A common theme is strategic use of confluence to reduce the number of interleavings we must consider.

The proof of *operational completeness* for Phase I leverages local confluence to simplify reasoning in a major way. The asynchrony and swapping rules in Kalas's semantics, which are otherwise a pain point, play no role in these proofs. This is because any reduction involving them has a common successor with a reduction that only uses the syntax-directed rules (e.g., rule Com from Table 1.1). This yields a simpler proof than, for example, Montesi [16, Appendix C]; his language is also confluent, yet his proof makes no use of this, and includes cases for the swapping and asynchrony rules.

To prove *operational soundness* we use a technique based on *inert reduction*, first conceived by van Glabbeek to study encodings from the synchronous to the asynchronous π-calculus [25]. Intuitively, an inert reduction is one that performs a bookkeeping step without committing to a branch. We say that $N_{\mathsf{E}} \longrightarrow N'_{\mathsf{E}}$ is *inert* if for every $N''_{\mathsf{E}} \neq N'_{\mathsf{E}}$ such that $N_{\mathsf{E}} \longrightarrow N''_{\mathsf{E}}$, there is an N'''_{E} such that $N'_{\mathsf{E}} \longrightarrow N'''_{\mathsf{E}}$ and there is an inert transition $N''_{\mathsf{E}} \longrightarrow N'''_{\mathsf{E}}$. The key insight is that for encodings that only use inert catch-up transitions, operational soundness can be proven by induction on the length of the reduction sequence. Moreover, since inertness is a form of confluence, it suffices to consider just one interleaving of the intermediate steps, namely the one that directly mimics one source-language step at a time. All our catch-up transitions are inert, which makes the proof of *operational soundness* much more tractable, with roughly half the effort going into proving confluence. The same technique is also used to great effect to tame the interleaving explosion of Phase III.

Phase II uses a traditional invariant-based technique, which we found intractable for the other phases with more complicated interleavings. The main headache here is alpha-equivalence considerations arising from the need to invent fresh names.

The proofs for Phase IV are different. In the other phases, the bulk of the effort is chasing transitions. Here, that part is trivial since we have one-to-one transition correspondence (up-to strong bisimilarity). The difficulty is in wrangling the candidate relation used to prove that the continuations of fixpoints and **letrec** unfoldings are bisimilar. The relation, which describes the precise relationship between closure environments and (possibly unfolded) fixpoints, is surprisingly complicated at almost 50 lines of HOL4 script. It is worth pointing out that this complicated relation entails no trust issues; its only use in the overall proof story is to witness an existential quantifier.

1.8 Compilation into CakeML

CakeML [10] 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 [21]. Interaction with the outside world is supported by a foreign function interface (FFI). We 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.

1.8.1 Static compiler

Example 12.
```
let val v =
  let val buff = Word8Array.array (σ + 1) 0
      fun receiveloop d =
        (#(receive) p buff;
         let val m = unpad buff
         in if final buff then concat(reverse(m::d))
            else let fun zerobuf(i) =
                         if i < 0 then ()
                         else (Word8Array.update(buff,i,0);
                         zerobuf(i-1))
                 in zerobuf(Word8Array.length(buff)-1);
                    receiveloop(m::d)
                 end
```

```
        end)
  in receiveloop [] end
in ⟦P⟧ML end
```

The static compilation is performed by the function $[\![\cdot]\!]_{\mathsf{ML}}$, which maps Payload endpoints to CakeML expressions. Its full definition would not fit here, but to show its flavour, $[\![\mathbf{receive}\, v\, \mathbf{in}\, \langle \epsilon \rangle\, \mathbf{from}\, p.P]\!]_{\mathsf{ML}}$ produces the code in Example 12

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

We use a small-step, relational (but deterministic) presentation of CakeML's semantics, allowing a natural expression of our eventual simulation theorem. We write $(p_0, cs_0) \rightarrow_c (p, cs)$, with p_0 the initial CakeML program, and cs_0 its accompanying state, to mean that this pair can evolve in a single step to (p, cs). The states cs_i contain FFI information (see below), the internal program state (variable environment, reference contents), as well as a continuation stack to track what remains to be done. 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 generated CakeML code interacts with the choreography's other endpoints, so our FFI state models the outside world as triple (p, q, N), with p the name of the CakeML endpoint, q its queue, and N a Payload network that p interacts with. There is an unfortunate mismatch here: the FFI model must be a function (CakeML is deterministic), but Payload's semantics 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 us our message, as actions internal to N may or may not fire before N sends the message. However, as long as all endpoints in N have unique names (a reasonable invariant), Payload reductions and send actions are locally confluent. So whether such internal actions fired or not, the resulting states are observationally equivalent from p's point of view.

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

to satisfy $N \xrightarrow{\widetilde{p}\to p_1:\widetilde{d}} \xrightarrow{p\leftarrow p_1:d} \xrightarrow{\widetilde{p'}\to p_1:\widetilde{d'}} N'$. That is, the network component N' records its delivery of some number of messages to us (from $\widetilde{p}$), the delivery of our message d to p_1, followed by its sending us possibly yet more messages (from $\widetilde{p'}$).

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

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

Let N_p denote the endpoint named p in N, and $N - p$ the network with that endpoint removed. Let $\mathsf{FFI}(cs)$ denote the FFI component of the CakeML state cs. Write $cs_1 =_{ffi} cs_2$ when $\mathsf{FFI}(cs_1)$ is bisimilar to $\mathsf{FFI}(cs_2)$ and all other components of the two states are equal.

Theorem 3 (Network Forward Correctness)**.** *Let N be a well-formed Payload network that includes an arbitrary endpoint p. Further, assume a CakeML state cs that is appropriately related to N (see below), with $\mathsf{FFI}(cs) = (p, q, N - p)$. Then, if N can reduce to N', there exist cs', mp (the "merge program"), cs_1 and cs_2 (two "merge states") such that*

- $\mathsf{FFI}(cs') = (p, q', N' - p)$ *and cs' is appropriately related to N' ;*
- $(\llbracket N_p \rrbracket_{\mathsf{ML}}, cs) \to_c^* (mp, cs_1)$*;*
- $(\llbracket N'_p \rrbracket_{\mathsf{ML}}, cs') \to_c^* (mp, cs_2)$*; and*
- $cs_1 =_{ffi} cs_2$*.*

The "appropriate relation" above between a network and a CakeML state requires that: all bindings of N_p are present in the CakeML state's environment; our library functions (e.g., `List.drop`) are defined and have the expected behaviour; and for every function f used in a let expression in N_p, a CakeML function that is a totally correct implementation of f is present in the environment.

As CakeML is deterministic, Theorem 3 gives us that (i) all infinite traces in Payload are necessarily simulated by an infinite trace in CakeML, and (ii)

compilation of a terminating choreography produces CakeML endpoints that will all also terminate successfully.

Though Theorem 3 tells us that every step taken by an endpoint will result in corresponding movement at the CakeML level, we have not transferred deadlock freedom to this level if the original choreography has only infinite paths. This is because currently, our theorems are not strong enough to rule out the possibility of *livelocks*: states where global progress is possible, but some nodes may be stuck waiting to receive. This is impossible by construction in Kalas, so while no such livelocks can occur (under weak fairness), operational correspondence by itself is only strong enough to guarantee global progress. One possible solution is to prove, in addition to operational correspondence, that an invariant stating "every receive can eventually be matched by a send" holds throughout the compilation chain.

1.8.2 Dynamic compiler by example

The dynamic compiler creates the initial environment assumed in Theorem 3, and proves that it is appropriate. 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.

Kalas 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 define the system in Example 9 as a choreography `filter` where the producer has an infinite message stream, and where `test` is a simple function that checks if the message starts with `"A"` or not. To run the compiler, the invocation is

```
project_to_camkes builddir filename "filter";
```

This automatically performs the following tasks: (i) proves that the current environment is appropriate; (ii) evaluates the compiler in the logic to produce CakeML code for each of the three endpoints; (iii) produces end-to-end theorems for each endpoint by composing Theorems 4 and 3, discharging all assumptions; (iv) finally, generates 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. Thus: 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.

1.9 Related Work

Session types [10] have seen extensive use in the π-calculus [11] and other concurrent languages [5, 12, 18, 26]. In recent years, the field has seen more mechanised proofs, perhaps motivated by past mistakes [19, 27]. In Castro *et al.* [3] a revised version of the session-typed π-calculus [27] is formalised in Coq [23]. Furthermore, Thiemann [24] proves type soundness and session fidelity in Agda [1] for an asynchronous functional session type language based on Gay *et al.* [7]. Tassarotti *et al.* [22] develop a higher-order concurrent logic, and verify a refinement procedure for a session-typed language as a case study. More broadly, Hinrichsen et al. [9] draw inspiration from session types to develop Actris, a higher-order concurrent separation logic capable of reasoning about multiple concurrency paradigms like message-passing, process forks, and critical section locks.

Hallal *et al.* [8] synthesise the distributed components of a communicating system from a global choreography. Their result aims only to capture the communication logic of the system; by way of contrast, we consider local computation also.

Carbone and Montesi [6, 16] present a choreographic language with multi-party 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, with a proof—albeit pen-and-paper—of projection correctness. Kalas began as a simplified version of their language.

The most closely related work is two recent Coq formalisations of endpoint projection in different settings, by Cruz-Filipe *et al.* [2], and by Hirsch and Garg [7]. Cruz-Filipe *et al.* verify endpoint projection from CC (Core Choreographies) to a distributed process calculus. Hirsch and Garg [7] formalise endpoint projection from Pirouette, a higher-order functional choreographic language, where functions can return, and be parameterised on, choreographies. The most obvious difference between our work and these other papers is one of scope: both of [2, 7] formalise endpoint projection in isolation; for us, this is just the first step towards our goal of integrating endpoint projection into an end-to-end verified compilation toolchain that can be used to build real, runnable code.

Both CC and Pirouette are parameterised on a local language for describing computation, which is assumed to be available also in the target language.

We achieve similar generality by representing local computation as shallow embeddings (functions in HOL4's logic). This lets us use a more abstract presentation, with no need to carry around an extra syntax, semantics, and associated well-formedness assumptions. The tradeoff is that we need a proof-producing (as opposed to verified) compiler phase to generate CakeML code.

In terms of semantics, one difference is that Kalas has asynchronous communication, whereas both CC and Pirouette are synchronous languages. Another interesting difference between the three languages is their representation of choreographies with infinite behaviour. Pirouette uses function closures. CC does not support the definition of local procedures, but executes in a context where a number of top-level, parameterless procedures are available. Kalas uses a fixpoint operator, which is parameterless, like CC, but supports arbitrary nesting of local procedures, like Pirouette.

CakeML has functions with closure semantics, but we nonetheless chose fixpoints over functions for Kalas. This is because, in an environment semantics, it is difficult to maintain a consistent view of the global environment in the presence of out-of-order execution: the semantics needs to track which local computations should be executed in the caller's environment (if they're ahead) or in the callee's environment (if they're behind). One solution is Cruz-Filipe *et al.* [2]'s approach, which breaks the abstraction of global, atomic actions by introducing an operator representing partially-completed procedure entry into the source language. Hirsch and Garg use an interesting approach, where function calls are considered global *both in source and target language*. In particular, executing a function call in a single endpoint has a CSP-like synchronous semantics where, as a single atomic action, the entire network performs the same function call together. While assuredly simplifying endpoint projection, this comes at the expense of complicated synchronisation when realising this in a distributed setting. In contrast, Kalas's unfolding of fixpoints can be implemented locally.

The target language used by Hirsch and Garg is a parallel composition of nodes expressed in the so-called *control language*. It mixes λ-calculus features with communication-enabling effects like send, receive and choose. This is rather like a functional language, which invites comparisons to our final target language, CakeML; but the role it plays in their development is much more akin to the role ENDPOINT plays in ours. Much like the relationship between Kalas and ENDPOINT, the feature set of Pirouette and the control language are essentially the same, except the latter is a localised representation.

Not all aspects considered by Hirsch and Garg, and by Cruz-Filipe *et al.*, are present in our work. For example, Hirsch and Garg prove progress and preservation for an associated type system, while we do not consider types at all. In a companion paper, Cruz-Filipe *et al.* [3] prove that CC is Turing-complete, by showing that it can implement partial recursive functions. Turing completeness for Kalas is trivial because local computations may use arbitrary HOL functions.

1.10 Conclusion

We have presented what we believe to be the first end-to-end verified compiler for a choreographic language. After passing through five phases and two intermediate languages, our language, Kalas can be compiled to machine-code by reusing existing work from the CakeML project. Further, we have implemented a deployment on top of the micro-kernel seL4, itself also verified software. There, message-passing is implemented by IPC between separate user-processes.

There are a number of interesting directions for future work. Data types other than strings require a framework for verified marshalling and de-marshalling. Our model of the communication backend assumes unboundedly long message queues, which is arguably unrealistic. It would be interesting to investigate if deadlock freedom holds in a model where queues are bounded but not lossy. Alternative ITree-base [18] semantics (i.e., a co-inductive observational semantics) for Kalas and other intermediate languages, could significantly simplify projection proofs and allow for more lax projectability criteria. 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 caveat, which could be discharged using CakeML's verified space-cost semantics [10]. We would also like to deploy on other communication backends, perhaps on top of TCP/IP, which would demonstrate verified distributed computation over the Internet.

$$\mathsf{pr}_p(\gamma, 0) = 0$$

$$\mathsf{pr}_p(\gamma, p_1.v_1 \twoheadrightarrow p_2.v_2; C) = \begin{cases} \bot & \text{if } p_1 = p_2 = p \\ \textbf{send}\, v_1\, \textbf{to}\, p_2.\mathsf{pr}_p(\gamma, C) & \text{if } p = p_1 \neq p_2 \\ \textbf{receive}\, v_2\, \textbf{from}\, p_1.\mathsf{pr}_p(\gamma, C) & \text{if } p \neq p_1 = p_2 \\ \mathsf{pr}_p(\gamma, C) & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_p(\gamma, \textbf{let } v@p_1 = f(\widetilde{v}) \textbf{ in } C) = \begin{cases} \textbf{let } v = f(\widetilde{v}) \textbf{ in } \mathsf{pr}_p(\gamma, C) & \text{if } p = p_1 \\ \mathsf{pr}_p(\gamma, C) & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_p(\gamma, \mu X.C) = \begin{cases} \mu X.\mathsf{pr}_p(\gamma[X := \mathsf{procs}(C)], C) & \text{if } p \in \mathsf{procs}(C) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_p(\gamma, X) = \begin{cases} \bot & \text{if } X \notin \mathrm{dom}(\gamma) \\ X & \text{if } p \in \gamma(X) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{pr}_p(\gamma, \textbf{if } v@p_1 \textbf{ then } C_1 \textbf{ else } C_2) = \begin{cases} \textbf{if } v \textbf{ then } \mathsf{pr}_p(\gamma, C_1) & \text{if } p = p_1 \\ \textbf{else } \mathsf{pr}_p(\gamma, C_2) & \\ p_1\, \textbf{chooses} & \text{if } p \neq p_1 \text{ and } \mathsf{sp}_{p_1,p}(C_1) = (\mathsf{T}, C_1') \\ \quad \mathsf{T} : \mathsf{pr}_p(\gamma, C_1') & \text{and } \mathsf{sp}_{p_1,p}(C_2) = (\mathsf{F}, C_2') \\ \textbf{or} & \\ \quad \mathsf{F} : \mathsf{pr}_p(\gamma, C_2') & \\ \mathsf{pr}_p(\gamma, C_1) & \text{if } p \neq p_1 \text{ and } \mathsf{pr}_p(\gamma, C_1) = \mathsf{pr}_p(\gamma, C_2) \\ \bot & \text{otherwise} \end{cases}$$

Table 1.3: Projection and projectability, with the *(sel)* case, which is similar to *(com)*, elided. Partiality is indicated with a result of $\bot$. Recursive calls (e.g., in the let case) that fail propagate that undefinedness to the top-level.

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.

[6] 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.

[3] 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.

[1] L. Cruz-Filipe and F. Montesi. A core model for choreographic programming. *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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] J. K. Hinrichsen, J. Bengtson, and R. Krebbers. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):6:1–6:30, 2020. doi: 10.1145/3371074. URL https://doi.org/10.1145/3371074.

[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 session 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.

[10] 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] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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

Dancing in a forest of interactions

Alejandro Gómez-Londoño
Magnus O. Myreen
Johannes Åman Pohjola

Paper draft

Abstract. Choreographic languages provide a convenient model for communicating systems that can be projected to concrete implementations. Previous works have shown, via mechanized proofs, that correspondence between choreographic and target languages can be established while preserving valuable properties like deadlock freedom. The most common approach for achieving these kinds of results is to rely on small-step semantics to describe language behavior and subsequently perform all proofs. However, small-step semantics struggle to capture the unique mix of local and global behavior present in choreographies, leading to complicated correspondence theorems and difficult proof developments. In this paper, we present an alternative semantics for the choreographic language Kalas based on interaction trees—a flexible co-inductive denotation for programming languages. We show how interaction trees can be used to express both the global and local behaviors of choreographies and their correspondence with a target language—all with a reduced proof effort when compared to previous results. This work extends the previous formalization of Kalas with our new interaction tree-base semantics, and all proofs are mechanized using the HOL4 proof assistant.

2.1 Introduction

In recent years choreographies have emerged as a convenient formalism for describing and implementing communicating systems [1, 4, 7]. A choreography expresses the interactions between system components alongside their local computations, creating a complete global view of the whole system. This global perspective is accomplished by using a protocol-like syntax to denote communication and additional syntax for control flow, variable binding, and local computations.

As an example, consider the following simple choreography:

Example 13 (Simple choreography)**.**

```
1.  let l@A = [5,4,7,1,4] in
2.  A.l -> B.xs;
3.  let ys@B = sort(xs) in
4.  B.ys -> A.l;
```

Example 13 illustrates how communication can be seamlessly interleaved with internal computations. First, process `A` sends its local list l (Line 1) to process `B` (Line 2), which stores it in xs. Afterwards, `B` internally performs a sort operation on xs (Line 3) and sends the result back to `A` (Line 4).

Choreographies also guarantee deadlock-freedom *by construction*. Furthermore, one can synthesize the implementation of individual components from choreographies through a process known as endpoint projection. These properties (along with features such as concurrency and asynchrony) make choreographies a robust formalism and an ideal foundation for truly reliable communicating systems.

Rigorous machine-checked formalizations of the semantics and properties of choreographies exists [2, 4, 7]. Moreover, in recent work, a verified end-to-end choreography language compiler (Kalas) was implemented [13]. These results strengthen the guarantees choreographies can provide and expand their potential applications.

The formalization of choreographies, particularly their compilation into other

languages, is not without its challenges. Most formalizations of choreographies use small-step operational semantics, usually labeled transition systems. Small-step semantics are well studied, conveniently capture nondeterminism, and have standard definitions for many desirable properties (e.g., confluence, deadlock freedom, semantic correspondence, etc.); making them, at face value, an excellent choice for choreographies. However, small-step semantics struggle when reasoning about language correspondence is required. Correspondence proofs often require cumbersome operational soundness and completeness results [5], which must account (in the worst cases) for fundamentally different sets of transitions representing the same behavior—e.g., when different communication encoding are used. These shortcomings are present (and abundant) in verified compilers, as the correspondence of different languages' semantics must be obtained to show the compiler's overall correctness. Furthermore, for choreographic compilers, like Kalas', this issue is particularly troublesome as they have to consider a variety of behavior encodings (communication, selection, recursion, etc.) across multiple intermediate languages

In most choreographic semantics, there is almost no distinction between global component interactions and local computations, as both transitions occur at the same global level. Choreographies blend components into a single program structure, making them no longer external to one another and allowing their interactions to be represented similarly to local computations. However, this abstraction disappears as choreographies are compiled into less global representations (e.g., a process calculus). Components' boundaries become explicit through parallel composition; interactions become external from the perspective of individual components, as they require cooperation from others in their global environment. Meanwhile, local computations remain mostly unchanged, only acting within the boundaries of single components. This drastic change in the encoding of global interactions and their sudden disconnect with local computations causes correspondence proofs of choreographic compilers (in a small-step setting) to be challenging. Then, if we could better preserve the various behavior encodings throughout multiple languages, the overall proof effort of establishing semantic correspondence would be significantly reduced.

Interaction trees [18] are a promising new approach for modeling the external interaction of programs in a semantically convenient way. Interaction trees are coinductive structures with a continuation-passing style handling of external interactions. Instead of labeled transitions, the structure of interaction trees contains events (requests to the external environment) alongside

continuations which, given an action (the response from the environment), return the remaining structure. Furthermore, semantic correspondence and other properties can still be proven using powerful notions like bisimulation, coinduction, and observational equivalence. We believe interaction trees can express choreography semantics and support their verified compilation into individual components more flexibly than other traditional approaches.

This paper presents an interaction tree-based semantics for choreographies, which addresses the shortcomings of small-step representations in several ways. First, every component in the choreography is given an interaction tree denotation of their local behavior and intended interactions; all components are then combined into, what we call, *an interaction forest* (a collection of interaction trees), which handles global interaction behaviors separately in a language-agnostic manner. Consequently, establishing semantic correspondence between compilation phases is reduced to showing either the equality or bisimulation of interaction trees, as the interaction forest remains constant. Finally, desirable properties like deadlock-freedom can be generally stated in terms of interaction forests and proven "once and for all" at the choreography level as long as the property under consideration is stable under bisimulation.

Our approach creates a separation of concerns between local and global behaviors that, while seemingly contrary to the choreographic abstraction, provides a consistent semantic notion for every language in the compilation chain. Furthermore, we claim this approach allows for simpler proofs and cleaner semantics in the context of verified choreographic compilers. To support this claim, we developed a new interaction tree-based formalization of the Kalas language and compared it with its previous small-step formalization.

This paper's main contributions are:

- The definition of an interaction tree-based semantics for Kalas with a clear separation between local component behaviors and global component interactions.
- The proof of *endpoint projection correctness* of Kalas's projection function in the context of the new interaction tree-based semantics.
- Definition of deadlock freedom for an interaction forest and proof that our construction for Kalas programs is deadlock-free.

All definitions and proofs in this paper are mechanized in HOL4 [14] and available online[1].

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

2.2 Kalas in a nutshell

This section outlines the original presentation of Kalas based on small-step operational semantics as a prelude to our new interaction tree-based formalization in Section 2.3. Our work reuses the Kalas' syntax and several other key language concepts, which we briefly recap here for convenience.

Kalas is a choreography language and verified compiler implemented in HOL4 on top of the CakeML [10] compiler's toolchain. The language itself is reminiscent of core choreographies [1] with some additions to facilitate mechanized proofs and compilation. First, local computations (besides top-level branching) are shallowly embedded as HOL4 functions and compiled using the CakeML toolchain into executable code. Furthermore, all values are finite sequences of bytes (or simply strings), relying on verified marshalling and unmarshalling at the HOL4 level for type safety. Finally, there is a fixed number of processes, and all values are variable bound.

2.2.1 Syntax

Overall, the language design of Kalas attempts to focus solely on representing global interactions, conveniently offloading other more local aspects of the language. The syntax of Kalas is inductively defined by the following grammar:

Definition 9 (Kalas syntax)**.**

$$
\begin{array}{lcll}
C & ::= & p_1.v_1 \twoheadrightarrow p_2.v_2; C & \textit{(com)} \\
 & & p_1 \twoheadrightarrow p_2[b]; C & \textit{(sel)} \\
 & & \textbf{if } v@p \textbf{ then } C_1 \textbf{ else } C_2 & \textit{(if)} \\
 & & \textbf{let } v@p = f(\widetilde{v}) \textbf{ in } C & \textit{(let)} \\
 & & \mu X.\ C & \textit{(fix)} \\
 & & \textbf{call } X & \textit{(var)} \\
 & & \mathtt{0} & \textit{(nil)}
\end{array}
$$

Component names (p_i), variable names (v_i), and function names (X) are represented using strings. The empty choreography (nil) is denoted by 0. Communication (com) between two components sends the value of local p_1 variable v_1 and binds it to v_2 in p_2. Similarly, component p_1 can select (sel) a boolean branch b and communicate its choice to p_2. Control flow (if) branches over the value of v in p, with [0x01] as true, and [0x00] as false. Local computations (let) bind the result of applying f (a HOL4 function) to the values bounded to the variable list $\widetilde{v}$. Recursion is by fixpoint unfolding

(*fix* and *var*), where the following choreography:

$$\mu X.\ p_1 \twoheadrightarrow p_2[v]; \textbf{call}\ X$$

After one iteration becomes:

$$p_1 \twoheadrightarrow p_2[b]; \mu X.\ p_1 \twoheadrightarrow p_2[b]; \textbf{call}\ X$$

2.2.2 Semantics

The original semantics of Kalas was defined as an operational small-step semantics with support for asynchronous communication. Transitions are of the following form:

$$s_1 \triangleright c_1 \xrightarrow[l]{\tau} s_2 \triangleright c_2$$

Where a choreography (c_i) and a *value store* (s_i) transition under an action label τ and a trace of deferred asynchronous actions l. A *value store* s_i is a partial function $\mathsf{string} \times \mathsf{string} \hookrightarrow \mathsf{string}$ mapping local variables in a component to their corresponding value. Furthermore, action labels τ denote each of the operations that can be performed in a choreography as per Definition 9. Finally, the trace l of deferred asynchronous actions prevents out-of-order execution (e.g., send before receive) from taking place, a challenging property to enforce in the presence of asynchrony and control-flow operations.

The rest of this section discusses the most pertinent aspects of Kalas' semantics; we direct readers to the original article and source repository for the complete formalization.

Communication Distinct components might communicate if a variable binding exists in the sender. After the transition is performed, a new binding is introduced in the receiver. The rule itself is straightforward:

$$\frac{s\,(v_1,p_1) = d \quad p_1 \neq p_2}{s \triangleright p_1.v_1 \twoheadrightarrow p_2.v_2; C \xrightarrow[\epsilon]{p_1.v_1 \twoheadrightarrow p_2.v_2} s[(v_2,p_2) := d] \triangleright C}$$

The interaction between the two components is fully contained within the choreography and occurs in unison—sends and receives always match.

Concurrency Actions between disjoint components might occur in any order. Therefore, an action structurally "behind" another one can be performed first as long as both are component-wise unrelated. While each action has its own concurrency rule, they are all conceptually similar. For example, consider

the rule (below) for "skipping over" a communication action, where function $\mathrm{fp}(\alpha)$ returns the set of all processes present in α.

$$\frac{s \rhd C \xrightarrow[l]{\alpha} s' \rhd C' \quad p_1 \notin \mathrm{fp}(\alpha) \quad p_2 \notin \mathrm{fp}(\alpha)}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C \xrightarrow[l]{\alpha} s' \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C'}$$

Concurrency is fundamental for allowing choreographies to model the interleaving behaviors of most communication systems.

Asyncrony In previous rules, interactions between components seem to occur atomically. However, communication is often implemented using non-blocking send primitives. Therefore, similarly to *concurrency*, a component might perform an action "before" another action in which it is involved as a sender is completed. Delaying a communication due to asynchrony requires that any component overlapping occurs only on the sender. Furthermore, an action writing to a variable can only occur after all remaining readings of that variable have been performed. Concretely, the rule below asynchronously performs an action over a communication, where $\mathrm{wv}(\alpha)$ returns any variable being written by α.

$$\frac{\begin{array}{rll} & & p_2 \notin \mathrm{fp}(\alpha) \\ s \rhd C \xrightarrow[l]{\alpha} s' \rhd C' & \mathrm{wv}(\alpha) \neq (v_1, p_1) & p_1 \in \mathrm{fp}(\alpha) \end{array}}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C \xrightarrow[(p_1.v_1 \twoheadrightarrow p_2.v_2) :: l]{\alpha} s' \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C'}$$

The delayed communication is prepended to l so that asynchronous actions can be safely performed over **if** statements. Conceptually, an action might occur over an **if** statement when its behavior is not affected by branch choice. To enforce this in the presence of asynchronous actions, one must ensure that the delayed actions are the same on each branch up to some reordering ($\simeq$).

$$\frac{s \rhd C_1 \xrightarrow[l]{\alpha} s' \rhd C_1' \quad s \rhd C_2 \xrightarrow[l']{\alpha} s' \rhd C_2' \quad l \simeq l' \quad p \notin \mathrm{fp}(\alpha)}{s \rhd \textbf{if } v@p \textbf{ then } C_1 \textbf{ else } C_2 \xrightarrow[l]{\alpha} s' \rhd \textbf{if } v@p \textbf{ then } C_1' \textbf{ else } C_2'}$$

Deadlock-freedom Choreographies are meant to be deadlock-free by construction; as such, Kalas' semantics provide a standard notion of deadlock-freedom.

$$\vdash c \neq \mathtt{0} \Rightarrow \exists\, \tau\, l\, s'\, c'.\, s \rhd c \xrightarrow[l]{\tau} s' \rhd c'$$

Informally, deadlock freedom guarantee that the choreography as a whole can advance or that it terminates successfully.

2.2.3 Endpoint Projection

Endpoint projection is the process of translating a choreography into individual component implementations. Informally, to project a component from a choreography, one must consider only the actions where the component is present. For example, the projection of $p_1.v_1 \twoheadrightarrow p_2.v_2$ is as a send operation for p_1, a receive for p_2, and is ignored for all other components.

The language used as the projection target is often a CCS-style [12] language with parallel composition and communication primitives. Kalas projects into the Endpoint language, a network of sequential component descriptions joined by parallel composition. In Endpoint, each component has a name (e), its own *value store* (e), and a message *queue* (q).

Definition 10 (Endpoint syntax)**.**

$$
\begin{array}{rcll}
P, Q & := & \textbf{send}\, v\, \textbf{to}\, p.P & \textit{(output)} \\
 & & \textbf{receive}\, v\, \textbf{from}\, p.P & \textit{(input)} \\
 & & \textbf{choose}\, b\, \textbf{for}\, p.P & \textit{(internal choice)} \\
 & & p\, \textbf{chooses}\, \mathsf{T} : P\, \textbf{or}\, \mathsf{F} : Q & \textit{(external choice)} \\
 & & \textbf{if}\ v\ \textbf{then}\ P\ \textbf{else}\ Q & \textit{(if)} \\
 & & \textbf{let}\ v = f(\widetilde{v})\ \textbf{in}\ P & \textit{(let)} \\
 & & \mu X.P & \textit{(fix)} \\
 & & \textbf{call}\ X & \textit{(var)} \\
 & & \mathtt{0} & \textit{(nil)} \\
 & & & \\
N & := & N_1 \mid N_2 & \textit{(parallel)} \\
 & & (p, e, q) \triangleright P & \textit{(endpoint)} \\
 & & 0 & \textit{(nil)}
\end{array}
$$

The syntax of Endpoint in Definition 10 is spit into individual components (P, Q) and their *network* composition (N). Components are described similarly to choreographies (Definition 9), but with communication (input,output) and selection (internal and external choice) split into individual actions for each process involved. The *network* N not only joins components in parallel composition, but extends them with a name, a *queue*, and a *value store*.

Endpoint projection is, however, not always successful, as some choreographies can not be translated directly. The main obstacle is ambiguous branching on **if** statements. For example, consider the following choreography:

$$\textbf{if}\ A@v\ \textbf{then}\ B.v \twoheadrightarrow A.v\ \textbf{else}\ A.v \twoheadrightarrow B.v$$

Here, it is unclear whether component B should send or receive after component A has chosen a branch. Fortunately, choreographies can be repaired and made projectable quite easily:

$$\begin{aligned}&\textbf{if } A@v\\&\textbf{then } A \twoheadrightarrow B[\mathtt{T}]; B.v \twoheadrightarrow A.v\\&\textbf{else } \ A \twoheadrightarrow B[\mathtt{F}]; A.v \twoheadrightarrow B.v\end{aligned}$$

A *projectability criteria* denotes which choreographies can be projected. Defintion 11 states this criterion for Kalas branching—other more straightforward requirements are used to rule out edge-cases.

Definition 11 (Kalas projectability criteria for branching)**.**
Any branching component must communicate its choice to all other components affected.

Endpoint projection for Kalas is defined as a function from choreographies to a pair (ok, ep), where ok is the projectability of the given choreography and ep is the resulting Endpoint network. The definition of this function is quite involved; however, it follows our earlier intuition for the communication case:

$$\begin{aligned}&\mathtt{project}\ p\ dvars\ (p_1.v_1 \twoheadrightarrow p_2.v_2; c) \stackrel{\text{def}}{=}\\&\ \textbf{if } p = p_1 \wedge p = p_2 \textbf{ then } (\mathtt{F}, \mathtt{0})\\&\ \textbf{else if } p = p_1 \textbf{ then } \mathtt{Send}\ p_2\ v_1 < \Gamma > \mathtt{project}\ p\ dvars\ c\\&\ \textbf{else if } p = p_2 \textbf{ then } \mathtt{Receive}\ p_1\ v_2 < \Gamma > \mathtt{project}\ p\ dvars\ c\\&\ \textbf{else } \mathtt{project}\ p\ dvars\ c\end{aligned}$$

The complete *projectability criteria* for Kalas is then embedded into its projection function. Furthermore, combinator $(< \Gamma >)$ lifts the projectability result of recursive calls to the top result. Note, for example, that self-communication is forbidden and leads to unprojectability. Other actions are projected similarly except for branching, which requires careful implementation of Definition 11.

The resulting Endpoint network must preserve all behaviors from the original choreography. Thus, allowing properties like deadlock-freedom to be translated directly to the Endpoint implementation. This correspondence is known as *endpoint projection theorem* and is proved w.r.t the semantics of Kalas and Endpoint.

Theorem 4. *If c is a projectable choreography*

1. *(Operational completeness) If $s \triangleright C \longrightarrow s' \triangleright C'$ then there exist s'', C'' such that $s' \triangleright C' \longrightarrow* s'' \triangleright C''$ and $[\![s \triangleright C]\!] \longrightarrow* [\![s'' \triangleright C'']\!]$*

2. *(Operational soundness) If* $[\![s \triangleright C]\!] \longrightarrow N$ *then there exist* s', C' *such that* $N \longrightarrow\!* [\![s' \triangleright C']\!]$ *and* $s \triangleright C \longrightarrow\!* s' \triangleright C'$

The proof is phrased in terms of two dual properties *operations completeness* and *operations soundness*. In a nutshell, both properties state that a transition in either language can be eventually matched by its counterpart.

While this result successfully connects the behaviors of Kalas and translated Endpoint programs, there is room for improvement. Specifically, using two different semantics makes the proofs more complex as they must grapple with more granular communications and interleaving, despite component behavior being unchanged. Moreover, more relaxed projectability criteria are hard to implement as they would require a complete refactoring of the original proof—a monumental task on its own. These and other issues are commonplace for formalizations of this kind. However, they must not be accepted as unsurmountable facts of life, as will be demonstrated in the upcoming sections.

2.3 Choreographies as Interactions Trees

Interaction trees are coinductive structures that can denote, possibly divergent, programs that interact with the environment. Their definition is deceptively simple:

```
(α, ε, ς) itree =
  Ret ς
 | Tau ((α, ε, ς) itree)
 | Vis ε (α → (α, ε, ς) itree)
```

Interaction trees have then three building blocks:

Results (`Ret` r) computations returning a value r.

Internal computations (`Tau` t) a single internal step followed by an interaction tree t with further computations.

External interaction (`Vis` $e\ f$) a visible events e and continuation function f which expect an action response a from the environment to continue as $f\ a$.

Hol4's simple type system limits the type of events (ϵ), actions (α), and results (σ) to independent type variables. However, more expressive type systems can implicitly derive the type of actions from that of events—i.e., `Vis (A E) (A ->`

`E R itree`). This limitation does not affect interaction trees' expressivity, but it requires some extra considerations when creating specific type instances.

One can use interaction trees to define a program that prints "yes" indefinitely—the `yes` program.

$$\texttt{yes} = \texttt{Vis}\ \text{“yes”}\ (\lambda x.\ \texttt{yes})$$

Events in `yes` are of type `string`, and since no response is expected from the environment, actions are of type unit.

Programs can interact with the environment: when making a visible action, the environment can give an answer. The problem above ignored the environments response, i.e. x. But we can easily make a simple example that uses the response. The following is a program that prints the reverse of any input given —the `rev_echo` program with actions and events of type `string`.

$$\texttt{rev\_echo}\ msg = \texttt{Vis}\ msg\ (\texttt{rev\_echo} \circ \texttt{REVERSE})$$

As showcased by these examples, the type of actions (α) and events (ϵ) determines the interface by which interaction trees "talk" with their environment. Therefore, the environment of an interaction tree can be modeled as the events it can observe and the actions it can give in return. This approach provides excellent flexibility for program reasoning, as the environment model is kept separated from the program's semantics.

Additionally, interaction trees can be interpreted into other representations by traversing their structures while handling events. Interpreters can be seen as executing interaction trees for a given environment, as they must provide actions back to the interaction tree after every event. For instance, an interpreter for `rev_echo` might always give "yes" back as action and produce a possibly infinite list (i.e., a co-list) with only "sey" elements. Similarly, an interpreter for `yes` can produce an infinite trace of the "yes" message by always acting with the unit value—the only possible choice. To better illustrate this idea, consider $\texttt{interp} : [|\alpha|] \times (\alpha,\ \epsilon,\ \varsigma)\ \texttt{itree} \to [|\epsilon|]$, a generic interaction tree interpreter.

Definition 12 (Simple interpreter)**.**

$$\begin{aligned}
&\texttt{interp}\ [||]\ it = [||] \\
&\texttt{interp}\ (msg{:::}rest)\ (\texttt{Ret}\ r) = [||] \\
&\texttt{interp}\ (msg{:::}rest)\ (\texttt{Tau}\ t) = \texttt{interp}\ (msg{:::}rest)\ t \\
&\texttt{interp}\ (msg{:::}rest)\ (\texttt{Vis}\ e\ f) = e{:::}\texttt{interp}\ rest\ (f\ msg)
\end{aligned}$$

Given a co-list of α values, `interp` returns a co-list of ϵ values corresponding to the events generated by the given interaction tree. Using `interp` one can trivially interpret `rev_echo` to be a mapping of the reverse function over the co-list—an accurate representation. This generic interpreter can be used for almost any interaction tree (`interp` *xs it* is unspecified for a divergent *it*). However, `interp`'s environment model is rigid, as it does not react to what events occur; more expressive interpreters are possible, as we shall see in future sections.

2.3.1 A Local Perspective

Choreographies provide a global view of a system by expressing interactions between components and local computations in the same language. Consequently, as choreographies are projected into other languages, component interactions are meant to remain behaviorally equal to match the system specification. As a result, guarantees like deadlock-freedom can be extended to individual component implementations.

Small-step semantics for choreographies tend to be monolithic in their treatment of interactions and computations. A downside of such monolithic semantics is that any transformations to the language (e.g., through projection) must involve new semantics, even if some behaviors are unchanged. In particular, small-step semantics can not take advantage of equivalent component interactions through projection and must "reinvent the wheel" at every step.

In contrast, interaction trees provide a clear separation between external interactions (`Vis`) and local computations (`Tau`). Therefore, language transformations that only concern local computations are denoted by interaction trees with equivalent interactions.

A denotation of choreographies as interaction trees could, in principle, abstract component interactions and remain unchanged through projection. However, representing all components in a single interaction tree would “hide” much of the system’s behavior as `Tau` steps, since the choreography (as a whole) does not interact with an external environment but (locally) between individual components. Therefore, we must look at choreographies in a new light to take full advantage of an interaction tree representation.

A key observation is that an interaction tree can naturally denote a single component in a choreography. From the perspective of a single component, the "external" environment comprises all the other components in the choreography. An interaction tree can represent this perspective by handling any

interactions with the choreography as external. In this approach, component interactions are abstracted away as (`Vis`) events that expect a response from the choreography. Meanwhile, local computations are denoted by `Tau` and possibly `Ret`. As such, in a choreography $p.x \twoheadrightarrow q.y; q.y \twoheadrightarrow r.w$ one could express each component as follows:

Example 14 (Interaction trees for components of p, q and r)**.**

```
p_it = Vis (Send "q" x) done
q_it = Vis (Receive "p") (λ y. Vis (Send "r" y) done)
r_it = Vis (Receive "q") done
done x = Ret ()
```

Component p interacts with its environment by signaling through an event (`Send q x`) that it intends to send x to q and finish its execution, disregarding any reply. Likewise, component r's event (`Receive q`) signals the incoming transmission from q. More interestingly, component q first expects to receive into y a message from p and later relay its value to q. While each component is denoted as an independent interaction tree, their interactions are deeply linked by their common choreographic origin.

However, representing a single component as an interaction tree only captures a fraction of all the behaviors in a choreography.

Looking at Example 14, it appears that the collection of all components, even as interaction trees, somehow denotes the choreography itself. This intuition arises from the fact that all interactions in the original choreography (two communications in this case) appear as matching `Send` and `Receive` events in the corresponding components. Moreover, the action a component expects to receive after each event is evident from the event itself. For example, events `Send` q x in p and *Receive* p in q suggest that the value of x must be applied to the first continuation in q—matching the behavior of the choreography. This observation suggests that it might be possible to construct a complete denotation for a choreography by connecting the interaction trees of all components.

Following this intuition, we present a semantics for choreography languages (Kalas in particular) that uses a form of interaction tree composition to produce the expected global behavior. As a first step, we must denote single choreographic components as interaction trees—the remaining section is devoted to these representations. Later in Section 2.4, the details of their composition and the overall semantics are presented.

2.3.2 Single components as interaction trees

All definitions presented are based on the original Kalas' syntax and semantics; however, the overall approach ought to apply to other choreography formalizations.

The first step in defining an interaction tree for a choreography component is establishing appropriate types for events, actions, and results.

```
event = Send proc datum
      | Receive proc
      | Choose proc bool
      | Select proc
```

The `event` type models the primary forms of interaction (events) that can occur in a choreography—communication and selection. Value `Send p x` signals a component's intent to send value `x` to process `p`. Similarly, a component expecting to receive a message can produce the event `Receive q`. Selection is modeled in the same manner, with `Choose p b` picking a boolean branch `b` to communicate to component `p`, and `Select q` receiving that choice.

```
action = Ok
       | Msg datum
       | Branch bool
```

The environment response is characterized by the `action` type, which provides appropriate responses to each possible event. A component at the receiving end of a communication or a selection expects either `Msg x` or `Branch b` (respectively) as a response; where value `x` and boolean `b` are the interaction's result. When a component sends a value or chooses a branch, it only expects `Ok` as an acknowledgment from the environment.

As previously mentioned, in Hol4's type system the types of events and actions are independent; this in turn allows for event-action mismatches to occur—e.g., a `Receive` event that is given an `Ok` response. Therefore, interaction trees must handle these occurrences explicitly as erroneous results, and it is up to the interpreter to guarantee that such mismatches do not occur.

```
result = End
       | Done
       | Unproj
       | Error
```

In our denotation of choreographies, the result of a well-formed finite interaction tree must always be `End`, indicating that a component has completed

execution. However, the remaining values are used to indicate failures (`Error`), unprojectable behaviors (`Unproj`), and discarded branches (`Done`). All results other than `End` are either handled internally, ruled out by construction (from a choreography), or guaranteed by the interpreter to never occur. The coming sections present the use and appropriate handling of each result variant.

Once the appropriate types are established, a denotation function can be defined (by induction on the choreography's structure) to produce the interaction tree of a single component p. The denotation function filters all operations that relate to p and denote them as `Vis`, `Tau`, or `Ret` to form the corresponding interaction tree.

Given a choreography c, a component identifier p, and an initial variable binding s, the function `chor_itree` produces the interaction tree denoting p in c. The full definition of `chor_itree` is divided below into three distinct categories to aid the presentation: local computations, external interactions and branching.

Local computations Local computations occur only within the component boundary and do not require introducing any new mechanisms. Aside from explicit `End` and `Error` results, local computations are performed in the same fashion as in Kalas' small-step semantics.

$$
\begin{aligned}
&\texttt{chor\_itree}\ p\ s\ \texttt{0} = \texttt{Ret End}\\
&\texttt{chor\_itree}\ p\ s\ (\textbf{call}\ f') = \texttt{Ret Error}\\
&\texttt{chor\_itree}\ p\ s\ (\textbf{let}\ v@q = f(vl)\ \textbf{in}\ c) =\\
&\ \textbf{if}\ p = q\ \textbf{then}\\
&\quad \textbf{if}\ vl\ \subseteq\ \texttt{dom}\ s\ \textbf{then}\\
&\qquad \texttt{Tau}\ (\texttt{chor\_itree}\ p\ s[v := f\ (\texttt{map}\ s\ vl)]\ c)\\
&\quad \textbf{else}\ \texttt{Ret Error}\\
&\ \textbf{else}\ \texttt{chor\_itree}\ p\ s\ c\\
&\texttt{chor\_itree}\ p\ s\ (\mu f'.\ c) =\\
&\ \textbf{if}\ p \in \texttt{procsOf}\ c\ \textbf{then}\\
&\quad \texttt{Tau}\ (\texttt{chor\_itree}\ p\ s\ c[\textbf{call}\ f'/\mu f'.\ c])\\
&\ \textbf{else}\ \texttt{Ret End}
\end{aligned}
$$

The empty choreography (`0`) directly terminates with result `End`. In a well-formed choreography out-of-scope **call** operations can not occur and produce an `Error` result. Fixpoint unfolding is standard (substitution of **call**), but it requires the component to be present in the choreography to guarantee productivity. Finally, let-bindings of local computations related to the component are performed and added to the binding environment.

External interactions Communications and selections are denoted differently depending on the component's role in the interaction.

```
chor_itree p s (q1.v1 -> q2.v2;c) =
 if p = q1 then
  if v1 ∈ dom s then Vis (Send q2 (s v1)) (chor_itree_send p s c)
  else Ret Error
 else if p = q2 then Vis (Receive q1) (chor_itree_recv p s v2 c)
 else chor_itree p s c
chor_itree p s (q1 -> q2[b];c) =
 if p = q1 then Vis (Choose q2 b) (chor_itree_send p s c)
 else if p = q2 then Vis (Select q1) (chor_itree_select p s b c)
 else chor_itree p s c
```

A component sending a message produces an event (`Send` p x) with a recipient (p) and a message (x), followed by a continuation (`chor_itree_send`) expecting (Ok) as an acknowledgment. Conversely, when receiving a message, the expected origin (q) is signaled in an event (`Receive` q) alongside a continuation (`chor_itree_recv`) binding the value received to the appropriate local variable.

Selection interactions follow the same pattern as communications, with homologous events and corresponding continuations. When a component chooses a branch (b) for another component (q_2) it produces an event (`Choose` q_2 b) and, as before, expects an (Ok) acknowledgment—through `chor_itree_send`. In the opposite case, a component expecting a selection (from q_1) signals it with an event (`Select` q_1) and expects to receive a choice of branch matching b. If the choice received is not b, the component returns `Done` as an indicator that the current branch is being discarded.

Branching Branching operations are challenging as they originate in a single component but require the rest of the choreography to react appropriately to achieve projectability. In a projectable choreography all components must either choose a branch (directly or through selection) or have the same behavior regardless of the branch chosen.

```
chor_itree p s (if v@q then l else r) =
 if p = q then
  if v ∈ dom s then
   if s v = ⊤ then Tau (chor_itree p s l)
   else Tau (chor_itree p s r)
  else Ret Error
 else chor_itree_merge (chor_itree p s l) (chor_itree p s r)
```

If a component p is performing a branching operation, it chooses a branch based on the value of its local variable v. Otherwise, both branches are merged as there is no immediate way of locally determining which branch to take. A merge operation expects identical behavior from each branch until one is discarded with a `Done` result. In case of a mismatch, an `Unproj` result is generated, warning of ambiguous behavior under branching.

2.3.3 Revisiting Endpoint Projection

Endpoint projection is a crucial component of any formalization of choreographies. The correctness of this process in Kalas is guaranteed by the *endpoint projection theorem*. Given our new interaction tree-based semantics for Kalas, there is a need to reformulate this theorem.

As we did for choreographies, we must first create an interaction tree denotation for components in Endpoint–our target implementation language. In stark contrast to Theorem 8, we do not need to consider an entirely new semantic for Endpoint; instead, we can reuse our previous interaction tree types as the basis for a new denotation function. The function `ep_itree s n` denotes a single component of an Endpoint network as an interaction tree.

$$
\begin{aligned}
&\texttt{ep\_itree}\ s\ \texttt{0} = \texttt{Ret End}\\
&\texttt{ep\_itree}\ s\ (\textbf{call}\ f') = \texttt{Ret Error}\\
&\texttt{ep\_itree}\ s\ (\texttt{Let}\ v\ f\ vl\ e) =\\
&\ \ \textbf{if}\ vl \subseteq \texttt{dom}\ s\ \textbf{then}\ \texttt{Tau}\ (\texttt{epn\_itree}\ s[v := f\ (\texttt{map}\ s\ vl)]\ e)\\
&\ \ \textbf{else}\ \texttt{Ret Error}\\
&\texttt{ep\_itree}\ s\ (\mu f'.\ e) = \texttt{Tau}\ (\texttt{ep\_itree}\ s\ e[\textbf{call}\ f'/\mu f'.\ e])\\
&\texttt{ep\_itree}\ s\ (\textbf{if}\ v\ \textbf{then}\ l\ \textbf{else}\ r) =\\
&\ \ \textbf{if}\ v \in \texttt{dom}\ (s)\ \textbf{then}\\
&\ \ \ \ \textbf{if}\ s\ v\ =\ \top\ \textbf{then}\ \texttt{Tau}\ (\texttt{ep\_itree}\ s\ l)\\
&\ \ \ \ \textbf{else}\ \texttt{Tau}\ (\texttt{ep\_itree}\ s\ r)\\
&\ \ \textbf{else}\ \texttt{Ret Error}\\
&\texttt{ep\_itree}\ s\ (\textbf{send}\ v\ \textbf{to}\ p.e) =\\
&\ \ \textbf{if}\ v \in s\ \textbf{then}\ \texttt{Vis}\ (\texttt{Send}\ p\ (s\ v))\ (\texttt{ep\_itree\_send}\ s\ e)\\
&\ \ \textbf{else}\ \texttt{Ret Error}\\
&\texttt{ep\_itree}\ s\ (\textbf{receive}\ v\ \textbf{from}\ p.e) \quad = \texttt{Vis}\ (\texttt{Receive}\ p)\ (\texttt{ep\_itree\_recv}\ s\ v\ e)\\
&\texttt{ep\_itree}\ s\ (\textbf{choose}\ b\ \textbf{for}\ p.e) \quad = \texttt{Vis}\ (\texttt{Choose}\ p\ b)\ (\texttt{ep\_itree\_send}\ s\ e)\\
&\texttt{ep\_itree}\ s\ (p\ \textbf{chooses}\ \texttt{T}:\ l\ \textbf{or}\ \texttt{F}:\ r) = \texttt{Vis}\ (\texttt{Select}\ p)\ (\texttt{ep\_itree\_select}\ s\ l\ r)
\end{aligned}
$$

The definition of `ep_itree` is almost identical to that of Kalas' `chor_itree`, if only simpler. An empty component definition returns an explicit `End` result. Local computations are homologously treated, with `Let`, `Call`, and `Fix` being

denoted in the same way using `Tau`. Branching statements in Endpoint are local and, as such, only require a `Tau` step after looking at the branching (local) variable. Furthermore, component interactions use individual operations with explicit roles allowing the one-to-one mapping of events and continuations in `Vis`. Overall, it is remarkable how the denotation of these two languages can be so similarly defined.

This correspondence between denotations hints at a closer link between Kalas and Endpoint. Specifically, while their syntax and operation might differ, the underlying behaviors both languages aim to represent are the same—interactions between components. Therefore, an interaction tree denotation, which excels at capturing this behavior, will necessarily be very similar.

With a common interaction tree denotation, the *endpoint projection theorem* can be stated in a delightfully simple manner.

$$\begin{aligned}&\texttt{project\_ok}\ p\ \ c \Rightarrow \\ &\ \uparrow(\texttt{chor\_itree}\ p\ s\ c) = \texttt{ep\_itree}\ s\ (\texttt{project}\ p\ \ c)\end{aligned}$$

Reusing Kalas' projection function and associated projectability criteria, it is possible to prove that both representations are equal and will produce the same behaviors given the same response from the environment. Operand $\uparrow$ lifts any `Done` result to `End`; this is required as Endpoint has no notion of discarded branches, and the absence of `Done` can not be shown for a single component. Nonetheless, when the whole choreography is considered $\uparrow$ can be easily removed.

An attentive reader might notice that this version of the *endpoint projection theorem* only concerns a single component instead of the whole system—a valid observation. However, since the same form of interaction tree composition can be used to denote global behavior, regardless of language, this result extends naturally to the whole system.

2.4 Interaction Forests

An interaction forest is a collection of interaction trees that are collectively interpreted to produce a global behavior. In general, interaction forests connect two (or possibly more) related events from interaction trees and interpret them with the appropriate actions. For example, consider two interaction trees representing components p and q performing communication events.

$$\begin{aligned}&\texttt{p\_it} = \texttt{Vis}\ (\texttt{Send}\ \text{“q”}\ \texttt{x})\ (\lambda ok.\ \texttt{p\_rest}) \\ &\texttt{q\_it} = \texttt{Vis}\ (\texttt{Receive}\ \text{“p”})\ \texttt{q\_rest}\end{aligned}$$

It is clear from each event that p is attempting to communicate the value of x to q. An interaction forest could react to this and advance both components by providing an `Ok` acknowledgment to `p_rest` (which it disregards) and the value of x to `q_rest`. In essence, interaction forests interpret each interaction tree they hold based solely on their events and a notion of how they relate to each other.

An interaction forest needs to include not just a collection of interaction trees, but also information about how they should behave collectively. This enables the creation of a single interpreter that can be applied to all interaction forest instances. Furthermore, by stating generic properties w.r.t this interpreter, proof efforts can be shifted to the specific details of each interaction forest, rather than its interpretation.

The following record defines an abstract interaction forests value:

$$
\begin{aligned}
&(\alpha, \epsilon, \pi, \varsigma, \sigma)\ \texttt{interp}_\Psi = \langle\!| \\
&\quad \mathsf{forest} : \pi \mapsto (\alpha, \epsilon, \varsigma)\ \texttt{itree}; \\
&\quad \mathsf{st} : \sigma; \\
&\quad \mathsf{act} : \sigma \to \pi \to \epsilon \to \alpha\ \texttt{option}; \\
&\quad \mathsf{upd} : \sigma \to \pi \to \epsilon \to \sigma \\
&|\!\rangle
\end{aligned}
$$

The first and primary field is a mapping (forest) from component identifiers to interaction trees on which to operate. A state (st) holds any bookkeeping necessary for interactions—e.g., message queue, resource pools, broadcast challenges. An actuation function (act) determines given a state, a component identifier, and an event which action to pass to the component to continue. Finally, an update function (upd) appropriately modifies the state after every action taken by act.

The inclusion of a state (st) and its associated update function (upd) is motivated by the goal of creating a generic interpreter. Although it may be possible to adapt the interpreter for specific instances of an interaction forest without utilizing a state, doing so would likely diminish the interpreter's overall generality. For instance, consider an interaction forest where communication is synchronous. In this scenario, it might be feasible to avoid using a state by exchanging the sent value from one interaction tree to another in a single step. However, in an asynchronous setting, the same interpretation would not be feasible as the sent values must be retained until it is received. Therefore, the use of a state becomes crucial for correctly interpreting an interaction forest in such cases.

To facilitate the definition of an interpreter, a minimal interface is provided to perform modifications to interaction forests.

```
del_Ψ ψ p = ψ with forest := ψ.forest \\ p
set_Ψ ψ p i = ψ with forest := ψ.forest[p := i]
get_Ψ ψ p = (ψ.forest p)
upd_Ψ ψ p e f =
 case ψ.act ψ.st p e of
  None ⇒ ψ
 | Some a ⇒ ψ with ⟪forest := ψ.forest[p := f a]; st := ψ.upd ψ.st p e⟫
```

While `delete`, `get` and `insert` are standard map operations over forest, the definition of `update` models the relationship between the fields act, st, and upd. Specifically, `update` encodes how any action picked by act will be used to update st using upd.

Using this interface, we can construct a function that interprets all interaction trees in unison—an interaction forest interpreter. On a high level, an interpreter for an interaction forest operates in three phases: (i) non-deterministically chooses an interaction tree to act on, (ii) performs the chosen action—i.e., interactions or local computations—updating the state accordingly, and (iii) records the behavior that just occurred. By repeatedly performing these phases, we can construct a global denotation of all the behaviors in the forest. The interpreter halts once there are no more available actions or local computations. However, since termination is not guaranteed nor required, the chosen denotation must be able to model non-termination—e.g., by using a co-inductive structure.

We present an interaction forest interpreter implementation by describing each of the three phases followed by a complete definition of the interpreter.

Phase (i) - choosing an interaction tree to act on At every step, an interaction forest interpreter must choose an interaction tree on which to act. However, this choice must not be deterministic to be able to model asynchronous and concurrent behavior—i.e., all the possible orderings in which interactions might occur.

To model non-deterministic choice, our interpreter picks from a trace of choices (a co-inductive infinite list containing component identifiers) that it takes as an argument. This approach considers all possible choices in the interpreter's definition and its subsequent general properties, as it allows universal quantification. Other alternative approaches are Hilbert's choice, oracle functions, or even interaction trees. However, we choose traces of

choices due to their ease of reasoning, but more importantly, because they can naturally express assumptions on choice.

Not all traces of choices are adequate; crucially, some do not allow every component in an interaction forest to act. For example, consider an interpreter execution using a trace that only chooses component a and an interaction forest with more than one component.; clearly, the results would not capture other component's behaviors, only those of a. The expectation that every component is eventually chosen and thus its behavior accounted for is known as *fairness*. To achieve *fairness*, we must restrict any trace of choices to be *fair* w.r.t all components in an interaction forest. Formally, a *fair* trace is co-inductively defined as:

Theorem 5. *Given an interaction forest ψ and a trace of component identifiers l, we say that l is fair w.r.t ψ iff*

- *Every component in ψ is present in a finite prefix of l*
- *Every final segment of l is* fair *w.r.t ψ*

In our presentation, we restrict ourselves to *fair* traces, as it guarantees the *fairness* of our interaction forest interpreter.

Since our interpreter is only interested in choosing components it can act on; we must first define the notion of available actions of an interaction forest.

The available actions of an interaction forest ψ are all the components in ψ.forest it can interpret in the current state ψ.st. The following function describes when an interaction tree p can act in ψ—i.e., it can be interpreted.

$$
\begin{aligned}
&\texttt{can\_act}_{\Psi}\ \psi\ p \iff \\
&\ \textbf{case}\ \texttt{get}_{\Psi}\ \psi\ p\ \textbf{of} \\
&\ \ \texttt{None} \Rightarrow \texttt{F} \\
&\ |\ \texttt{Some}\ (\texttt{Ret}\ v_6) \Rightarrow \texttt{T} \\
&\ |\ \texttt{Some}\ (\texttt{Tau}\ v_7) \Rightarrow \texttt{T} \\
&\ |\ \texttt{Some}\ (\texttt{Vis}\ e\ f) \Rightarrow \texttt{isSome}\ (\psi.\text{act}\ \psi.\text{st}\ p\ e)
\end{aligned}
$$

Components not in ψ.forest can not be interpreted. Local computations `Tau` and result values `Res` can always be interpreted. However, external interactions `Vis` can only be performed if an action can be provided by ψ—via ψ.act.

Finally, to choose between all available actions, we pick the first component identifier in the trace of choices (from phase (i)) for which we can act. The

definition of next_proc returns the next component that can act along with the remaining trace of choices.

```
next_proc ψ xs =
 case LDROP_WHILE ((¬) ∘ can_act_Ψ ψ) xs of
  [||] ⇒ None
 | p:::ll ⇒ Some (p,ll)
```

Using LDROP_WHILE, the identifiers of components that can not act are removed from the front of the trace of choices, and the first component that can act is returned.

Phase (ii) - performing the chosen action Once a component with an available action has been chosen, the next step is to advance the interaction forest by performing the given action. To perform an action in a component p it is enough to advance it by one step and update ψ.forest accordingly. The auxiliary function $\texttt{step}_\Psi$ performs one action on the given component.

```
step_Ψ ψ p =
 case get_Ψ ψ p of
  None ⇒ ψ
 | Some (Ret t) ⇒ del_Ψ ψ p
 | Some (Tau t') ⇒ set_Ψ ψ p t'
 | Some (Vis e f) ⇒ upd_Ψ ψ p e f
```

When a component finishes execution and returns a result, it is removed from ψ.forest. Local computations τt are consumed and then replaced in ψ.forest by the remaining interaction tree t. External computations are acted upon using $\texttt{upd}_\Psi$, which updates the internal state ψ.st according to ψ.upd.

Phase (iii) - recording behaviors To encode the kind of behaviors that might occur in an interaction tree, the following data type is used:

```
(α, ς) itree_action = Ext α | Int | Res ς
```

The event in an external computation is recorded inside Ext. Similarly, any resulting values are held in Res. Internal computations are denoted with Int.

Given a component identifier p and an interaction forest ψ, we can denote

the next behavior of p in ψ (assuming it can act) with $\texttt{act}_\Psi$.

$$
\begin{aligned}
&\texttt{act}_\Psi\ \psi\ p = \\
&\ \textbf{case}\ \texttt{get}_\Psi\ \psi\ p\ \textbf{of} \\
&\ |\ \texttt{Some}\ (\texttt{Ret}\ t)\ \Rightarrow\ (p,\texttt{Res}\ t) \\
&\ |\ \texttt{Some}\ (\texttt{Tau}\ t')\ \Rightarrow\ (p,\texttt{Int}) \\
&\ |\ \texttt{Some}\ (\texttt{Vis}\ e\ f)\ \Rightarrow\ (p,\texttt{Ext}\ e)
\end{aligned}
$$

If p is not present in ψ, it can not act, breaking our assumption. Otherwise, the denotation is one-to-one with the corresponding interaction tree as an identifier-behavior pair.

An interaction forest interpreter By combining all of the operations presented in previous phases, we define our interaction forest interpreter as follows:

$$
\begin{aligned}
&\texttt{interp}_\Psi\ \psi\ \mathit{trace} = \\
&\ \textbf{case}\ \texttt{next\_proc}\ \psi\ \mathit{trace}\ \textbf{of} \\
&\ \ \texttt{None}\ \Rightarrow\ [\|] \\
&\ |\ \texttt{Some}\ (p,ll)\ \Rightarrow\ \texttt{act}_\Psi\ \psi\ p :::\texttt{interp}_\Psi\ (\texttt{step}_\Psi\ \psi\ p)\ ll
\end{aligned}
$$

The empty co-list is returned if there are no components on which to act. Otherwise, the behavior of the chosen action is added at the front of the co-list using $\texttt{act}_\Psi$, followed by a recursive call on the updated–trough $\texttt{step}_\Psi$–interaction forest. All three phases are present, with `next_proc` and *trace* providing choice (i), actions being performed by $\texttt{step}_\Psi$ (ii), and $\texttt{act}_\Psi$ recording behaviors (iii).

The resulting denotation is a co-list of identifier-behavior pairs, which we refer to as a behavior trace. More formally, the complete behavior of an interaction forest ψ is the set of behavior traces obtained by applying $\texttt{interp}_\Psi$ to all *fair* traces of choices.

2.4.1 Deadlock-freedom

A notion of deadlock freedom can be established for the resulting behavior trace of an interaction forest.

As a first step, we must rule out any ill-formed behavior traces–i.e., those that do not originate from $\texttt{interp}_\Psi$.

A behavior trace is well-formed if no more behaviors are observed from a component after a result behavior occurs.

The predicate `actions_end` describes well-formed behavior traces by indexing–obtaining the *nth* element of a trace through `LNTH`.

$$\begin{aligned}&\texttt{actions\_end}\ \mathit{actions} \iff \\ &\ \forall\, n\ m\ p\ t\ a. \\ &\quad \texttt{LNTH}\ n\ \mathit{actions} = \texttt{Some}\ (p,\texttt{Res}\ t) \wedge n < m \Rightarrow \\ &\quad\ \texttt{LNTH}\ m\ \mathit{actions} \neq \texttt{Some}\ (p,a)\end{aligned}$$

In a nutshell, if an index n points at a result behavior for p, any index m greater than n does not point at a behavior involving p. As expected, any behavior trace produced by $\texttt{interp}_\Psi$ fulfills *action_end* by construction.

$$\vdash \texttt{actions\_end}\ (\texttt{interp}_\Psi\ \psi\ \mathit{trace})$$

The proof is by induction on the index n, using the definition of $\texttt{step}_\Psi$ to show that components are always removed after producing a result.

Deadlock-freedom requires a system (or its denotation) to show that it either terminates successfully or continues without getting stuck. A behavior trace of an interaction forest that terminates successfully must be finite and record a result behavior for every component. Conversely, an infinite behavior trace signals that the interaction forest does not get stuck. Formally this is described by the predicate `deadlock_freedom`.

$$\begin{aligned}&\texttt{deadlock\_freedom}\ \mathit{procs}\ \mathit{actions} \iff \\ &\ \texttt{actions\_end}\ \mathit{actions} \wedge \\ &\ (\neg\texttt{LFINITE}\ \mathit{actions} \vee \\ &\ \ \forall\, p.\ p \in \mathit{procs} \Rightarrow \\ &\qquad \texttt{exists}\ (\lambda\,(q,a).\ p = q \wedge \exists\, t.\ a = \texttt{Res}\ t)\ \mathit{actions})\end{aligned}$$

Therefore, to show that interaction forest is deadlock-free, one must show that its interpretation–trough $\texttt{interp}_\Psi$–satisfies `deadlock_freedom`.

2.5 Kalas in The Forest

To define a complete semantics for Kalas we can combine our interaction tree denotation of components (Section 2.3) with interaction forests. An interaction forest definition requires us to populate its type structure with appropriate values. We present an interaction forest denotation for Kalas by providing definitions for forest, st, act, and upd.

The mapping of interaction trees The first component, forest, can be constructed by applying `chor_itree` to every component in the choreography and mapping the resulting interaction tree with the corresponding component identifier. The function `chor_forest` takes a choreography and a list of component identifiers and creates a mapping of their interaction tree denotation.

$$\begin{aligned}&\texttt{chor\_forest}\ c\ s\ [\,] = \emptyset\\&\texttt{chor\_forest}\ c\ s\ (p{::}procs) =\\&\ (\texttt{chor\_forest}\ c\ s\ procs)[p := \texttt{chor\_itree}\ p\ (\texttt{projectS}\ p\ s)\ c]\end{aligned}$$

An empty list of component identifiers produces an empty mapping. Alternatively, it recursively performs a call to `chor_itree` with the front component p, an empty binding environment, and the initial choreography c. It is assumed that the list of component identifiers is distinct (no duplicate values) and that choreography c has no free variables.

Bookkeeping state The value of st contains message queues for communication between components. Message queues are represented as a list, while the overall state maps from sender-receiver pairs to message queues. Basic operations over the state are provided to add and retrieve messages.

$$\begin{aligned}&\texttt{message\_add}\ s\ p\ q\ d = s[(p,q) :=\ (s\ (p,q)) +\!\!+ [d]]\\&\texttt{message\_fetch}\ s\ p\ q =\\&\ \textbf{case}\ s\ (p,q)\ \textbf{of}\\&\ \ \texttt{None} \Rightarrow \texttt{None}\\&\ |\,\texttt{Some}\ [\,] \Rightarrow \texttt{None}\\&\ |\,\texttt{Some}\ (x{::}xs) \Rightarrow \texttt{Some}\ x\\&\texttt{message\_drop}\ s\ p\ q =\\&\ \textbf{case}\ s\ (p,q)\ \textbf{of}\\&\ \ \texttt{None} \Rightarrow s\\&\ |\,\texttt{Some}\ [\,] \Rightarrow s \setminus\setminus (p,q)\\&\ |\,\texttt{Some}\ [x] \Rightarrow s \setminus\setminus (p,q)\\&\ |\,\texttt{Some}\ (x{::}v_5{::}v_6) \Rightarrow s[(p,q) := v_5{::}v_6]\end{aligned}$$

Messages are appended to the corresponding queue with `message_add`; while the first message in the queue is retrieved with `message_fetch` and removed with `message_drop`.

Action function Interaction forests use act to determine if an action can be produced for a particular event. In the case of Kalas, events `Send` and `Choose`

can always be acted on, while `Receive` and `Select` events require a value to be present in the messages queue.

$$
\begin{aligned}
&\texttt{chor\_act}_\Psi\ s\ p\ (\texttt{Send}\ q\ d) = \texttt{Some Ok}\\
&\texttt{chor\_act}_\Psi\ s\ p\ (\texttt{Choose}\ q\ b) = \texttt{Some Ok}\\
&\texttt{chor\_act}_\Psi\ s\ p\ (\texttt{Receive}\ q) = \textbf{case}\ \texttt{message\_fetch}\ s\ p\ q\ \textbf{of}\ \texttt{None} \Rightarrow \texttt{None}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \texttt{Some}\ x \Rightarrow \texttt{Some}\ (\texttt{Msg}\ x)\\
&\texttt{chor\_act}_\Psi\ s\ p\ (\texttt{Select}\ q) = \textbf{case}\ \texttt{message\_fetch}\ s\ p\ q\ \textbf{of}\ \texttt{None} \Rightarrow \texttt{None}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \texttt{Some}\ x \Rightarrow \texttt{Some}\ (\texttt{Branch}\ (x = \top))
\end{aligned}
$$

An `Ok` acknowledgment is always given as the action for `Send` and `Choose` events. For `Receive` and `Select` events, the first message retrieved from the queue is used as the action.

Update function After an action is produced by act, changes to the state of the interaction forest might be required. Given that Kalas' state is composed of message queues and its events always signal some form of communication, changes to the state are direct message operations.

$$
\begin{aligned}
&\texttt{chor\_upd}_\Psi\ s\ p\ (\texttt{Send}\ q\ d) = \texttt{message\_add}\ s\ q\ p\ d\\
&\texttt{chor\_upd}_\Psi\ s\ p\ (\texttt{Choose}\ q\ b) =\\
&\quad \texttt{message\_add}\ s\ q\ p\ (\textbf{if}\ b\ \textbf{then}\ \top\ \textbf{else}\ [0w])\\
&\texttt{chor\_upd}_\Psi\ s\ p\ (\texttt{Receive}\ q) = \texttt{message\_drop}\ s\ p\ q\\
&\texttt{chor\_upd}_\Psi\ s\ p\ (\texttt{Select}\ q) = \texttt{message\_drop}\ s\ p\ q
\end{aligned}
$$

Events `Send` and `Choose` add their corresponding values to the message queue. Messages are meant to be retrieved and read when acting on `Receive` and `Select` events; therefore, they must be removed from the message queue.

A Kalas interaction forest As the last step, we combine our previous definitions to construct an interaction forest for a given Kalas program.

$$
\begin{aligned}
&\texttt{chor}_\Psi\ c\ s =\\
&\quad \langle\!|\texttt{forest} := \texttt{chor\_forest}\ c\ s\ (\texttt{procsOf}\ c);\ \texttt{st} := \emptyset;\\
&\qquad \texttt{upd} := \texttt{chor\_upd}_\Psi;\ \texttt{act} := \texttt{chor\_act}_\Psi |\!\rangle
\end{aligned}
$$

2.5.1 Deadlock freedom

Deadlock freedom holds by construction on any Kalas program denoted as an interaction forest.

Theorem 6.

$$\vdash \mathit{compile\_network\_ok}\ s\ c\ \wedge\ \mathit{fair\_trace}\ (\mathit{procsOf}\ c)\ \mathit{procs} \Rightarrow \mathit{deadlock\_freedom}\ (\mathit{procsOf}\ c)\ (\mathit{interp}_{\Psi}\ (\mathit{chor}_{\Psi}\ c\ s)\ \mathit{procs})$$

The proof is by contradiction: we assume that the interaction forest that one gets from a choreography is not deadlock free. This means that there must be a finite trace for some process p such that the trace does not contain a `Res` for that process. We perform induction on this finite trace. In particular, we show that every step of the trace preserves an invariant (explained below) and that, at the end of the trace, the invariant implies that the forest must represent a choreography without any processes, which means that there must have been a `Res` in the trace somewhere for process p. We use the following invariant in the proof: one can take some number of steps from the current interaction forest in order to reach an interaction forest that describes some choreography. The tricky part is that we have to show that some such new steps can be found whatever step is taken from the current interaction forest. Similarly to Theorem 2.3.3, the $\uparrow$ operand and a homologous version for interaction forests were required for this proof, as `compile_network_ok` is reliant on the structure of Endpoint and can not properly distinguish between `Done` and `End` results. Nonetheless, the use of $\uparrow$ was removed from the final *deadlock freedom* lemma as the whole choreography is considered, and the absence of `Done` can be shown.

2.5.2 Endpoint projection

An interaction forest denotation for Endpoint can be trivially derived from that of Kalas.

As with `chor_forest`, a mapping from component identifier to interaction trees must be created from an initial Endpoint program.

$$\begin{aligned}
&\mathtt{epn_forest}\ \mathtt{NNil} = \emptyset \\
&\mathtt{epn_forest}\ (\mathtt{NEndpoint}\ p\ s\ ep) = \emptyset[p := \mathtt{ep_itree}\ \emptyset\ ep] \\
&\mathtt{epn_forest}\ (N_1 \mid N_2) = (\mathtt{epn_forest}\ N_1 \sqcup \mathtt{epn_forest}\ N_2)
\end{aligned}$$

An empty network produces an empty mapping. Single endpoints are denoted using `ep_itree` and a singleton map. The parallel compositions of endpoints are merged recursively.

All other interaction forest fields can be reused from Kalas.

$$\begin{aligned}
&\mathtt{epn}_{\Psi}\ epn = \\
&\ \langle\!|\mathsf{forest} := \mathtt{epn_forest}\ epn;\ \mathsf{st} := \emptyset;\ \mathsf{upd} := \mathtt{chor_upd}_{\Psi}; \\
&\quad \mathsf{act} := \mathtt{chor_act}_{\Psi}|\!\rangle
\end{aligned}$$

A denotation this similar can be accomplished since i) Kalas and Endpoint use the same interaction tree denotation, and ii) the interaction model is compatible with both languages, i.e., communication through message queues.

It comes as no surprise that an *endpoint projection theorem* between Kalas' and Endpoint's follows trivially as an equality from our previous (Section 2.3.3) single component result. Where `compile_network` projects every component in the choreography into an Endpoint, creating a complete network

Theorem 7.

$$\begin{aligned}&\vdash \texttt{project\_ok}\ p\ c \wedge \texttt{fair\_trace}\ (\texttt{procsOf}\ c)\ procs \Rightarrow \\ &\quad \texttt{interp}_\Psi\ (\texttt{chor}_\Psi\ c\ s)\ procs = \\ &\quad\ \texttt{interp}_\Psi\ (\texttt{epn}_\Psi\ (\texttt{compile\_network}\ c\ ))\ procs\end{aligned}$$

This form of correspondence is considerably stronger than what can be achieved with small-step semantics (see Theorem 4). Interaction forests allow the preservation of behaviors between Kalas and Endpoint to be direct—via equality–obviating the need for "catch-up" transitions or notions like *operational completeness* and *operational soundess*. The use of equality allows properties like *deadlock freedom* (Theorem 6) to be trivially translated from Kalas to Endpoint; as well as any other property over $\texttt{chor}_\Psi\ c$.

Theorem 8. *Deadlock freedom for projected endpoint networks.*

$$\begin{aligned}&\vdash \texttt{compile\_network\_ok}\ s\ c\ \wedge \texttt{fair\_trace}\ (\texttt{procsOf}\ c)\ procs \Rightarrow \\ &\quad \texttt{deadlock\_freedom}\ (\texttt{procsOf}\ c) \\ &\quad\ (\texttt{interp}_\Psi\ (\texttt{epn}_\Psi\ (\texttt{compile\_network}\ c\ ))\ procs)\end{aligned}$$

More generally, given the coinductive nature of interaction forests and interaction trees, bisimulation relations can be conveniently used to express correspondence and translate properties between denotations.

2.6 Related work

Ever since their introduction, interaction trees have been a viable tool for proving the correctness of computer programs. Koh et al. [9] used interaction trees to denote and verify a swap server and relate its specifications at different levels of abstraction. In subsequent work, Zhang et al. [16] implemented a verified HTTP key-value server, refining its interaction tree-based specification of system calls to match that of CertiKOS [6]. More broadly in

the field of programming languages, Kanabar et al. [8] utilized interaction trees as the semantics of PureCake, a lazy functional language, and other intermediate languages used in PureCake's verified compiler. While in the context of concurrency, Lesani et al. [11] introduced *verified transactional objects*, a hybrid approach combining concurrent data structures and transactional memory, whose methods are expressed as interaction trees and can guarantee atomic execution.

Choreographies and endpoint projection have been mechanically formalized before in a variety of settings, albeit only using small-step semantics. Hirsch and Garg [7] mechanized the semantics of a higher-order functional choreographic language, Pirouette, and verified its endpoint projection into a concurrent λ-calculus. Moreover, Cruz-Filipe et al. [2, 3] formalized the semantics and projection of Core Choreographies [1], a minimal foundational model of choreographies, into a distributed process calculus. These results highlight choreographies' relevance and potential as a formalism for concurrency, but they also show how common the use of small-step semantics is, despite its downsides. Our work breaks with this paradigm and takes advantage of the versatility of interaction trees to develop a simple and expressive semantics that can be used as a basis for further development of choreographies.

2.7 Conclusions

We have presented a novel interaction tree-based semantics for Kalas, along with interaction forests as a compositional notion for interaction trees. Our approach improves on previous small-step formalizations in several ways. First, it creates a separation of concerns between local computations, denoted as interaction trees, and global interactions, represented as interaction forests. Second, the presentation of the *endpoint projection theorem* (Section 2.3.3) is significantly strengthened, as Kalas and its projection target, Endpoint, share denotations and interaction models. Finally, a general notion of *deadlock freedom* based on interaction forest provides the guarantees choreographies are known for with a significantly reduced proof burden.

More broadly, interaction trees brought some practical advantages that greatly aided development. Definitions were concise and, in many cases, could be reused; as such, this paper includes the complete interaction tree denotation of Kalas and Endpoint, along with all definitions for interaction forests. Many theorems saw an order of magnitude reduction in their proof's length, and statements became much easier to understand—compare, for example, The-

orems 4 and 8. Overall, in the authors' opinion interaction trees made the formalization of choreographies a much more manageable endeavor.

The new formalization of Kalas presented in this paper creates new routes for the development of the language. In particular, introducing less restrictive projectability criteria could allow for more expressive choreographies. While extending the *endpoint projection theorem* to support bi-similarity could broaden the scope of projectable languages.

The introduction of interaction forests also creates several directions for future work. One promising idea is to generalize the notion of deadlock freedom as a property over interaction forests (i.e., their components) rather than their behavior trace. Such characterization could allow proofs of deadlock freedom for similar denotations to be derived. Also promising is the use of linear temporal logic (LTL) to describe specific properties of interaction forests. This connection arose during development as predicates over traces were often reminiscent of common LTL concepts.

Bibliography

[1] L. Cruz-Filipe and F. Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. URL https://doi.org/10.1016/j.tcs.2019.07.005.

[2] L. Cruz-Filipe, F. Montesi, and M. Peressotti. Certifying choreography compilation. In *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, pages 115–133, 2021. doi: 10.1007/978-3-030-85315-0_8. URL https://doi.org/10.1007/978-3-030-85315-0_8.

[3] L. Cruz-Filipe, F. Montesi, and M. Peressotti. Formalising a Turing-complete choreographic language in Coq. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021. Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2021. URL http://doi.org/10.4230/LIPIcs.ITP.2021.15.

[4] L. Cruz-Filipe, E. Graversen, L. Lugović, F. Montesi, and M. Peressotti. Functional choreographic programming. In *International Colloquium on Theoretical Aspects of Computing*, pages 212–237. Springer, 2022.

[5] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. In F. van Breugel and M. Chechik, editors, *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 492–507. Springer, 2008. doi: 10.1007/978-3-540-85361-9_38. URL https://doi.org/10.1007/978-3-540-85361-9_38.

[6] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Im-*

plementation (OSDI 16), pages 653–669, Savannah, GA, Nov. 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[7] A. K. Hirsch and D. Garg. Pirouette: Higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL), Jan. 2022. doi: 10.1145/3498684. URL https://doi.org/10.1145/3498684.

[8] H. Kanabar, S. Vivien, O. Abrahamsson, M. O. Myreen, M. Norrish, J. Å. Pohjola, and R. Zanetti. Purecake: A verified compiler for a lazy functional language. 2022.

[9] N. Koh, Y. Li, Y. Li, L. Xia, L. Beringer, W. Honoré, W. Mansky, B. C. Pierce, and S. Zdancewic. From C to interaction trees: specifying, verifying, and testing a networked server. In A. Mahboubi and M. O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 234–248. ACM, 2019. doi: 10.1145/3293880.3294106. URL https://doi.org/10.1145/3293880.3294106.

[10] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *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, 2014. doi: 10.1145/2535838.2535841. URL https://doi.org/10.1145/2535838.2535841.

[11] M. Lesani, L.-y. Xia, A. Kaseorg, C. J. Bell, A. Chlipala, B. C. Pierce, and S. Zdancewic. C4: Verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022. doi: 10.1145/3527324. URL https://doi.org/10.1145/3527324.

[12] 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.

[13] J. Å. Pohjola, A. Gómez-Londoño, J. Shaker, and M. Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In *13th International Conference on Interactive Theorem Proving (ITP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[14] K. Slind and M. Norrish. A brief overview of HOL4. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs*. Springer, 2008.

[18] L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: representing recursive and impure

programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi: 10.1145/3371119. URL https://doi.org/10.1145/3371119.

[16] H. Zhang, W. Honoré, N. Koh, Y. Li, Y. Li, L.-Y. Xia, L. Beringer, W. Mansky, B. Pierce, and S. Zdancewic. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-188-7. doi: 10.4230/LIPIcs.ITP.2021.32. URL https://drops.dagstuhl.de/opus/volltexte/2021/13927.

3

Do You Have Space for Dessert? A verified space cost semantics for CakeML programs

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

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2020 [1]

[1] This thesis version of the paper exhibits slight differences from the published version.

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.

3.1 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 [17] 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:

machine_sem *ffi* (compile *c prog*) ⊆
extend_with_resource_limit (source_sem *ffi prog*)

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.

is_safe_for_space *ffi c prog* ... ⇒
machine_sem *ffi* (compile *c prog*) = source_sem *ffi prog*

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 [26] 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.

3.2 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.

3.2.1 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.

3.2.2 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 3.3).
- 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.

3.2.3 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 [23] 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.

3.2.4 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 3.3).

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 3.5).

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 [20], and `Generational` for a generational collector [16]. 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.

3.2.5 A note on semantics

The semantics of CakeML, and all of its intermediate languages, is defined in the functional big-step semantics style [14]. 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.

$$\begin{aligned}
&\texttt{extend_with_resource_limit}\ \mathit{behaviours} \stackrel{\text{def}}{=} \\
&\ \mathit{behaviours} \cup \\
&\ \{\ \texttt{Terminate Resource_limit_hit}\ \mathit{io_list} \\
&\quad \mid \exists\, t\ l.\ \texttt{Terminate}\ t\ l \in \mathit{behaviours} \wedge \mathit{io_list} \preccurlyeq l\ \} \cup \\
&\ \{\ \texttt{Terminate Resource_limit_hit}\ \mathit{io_list} \\
&\quad \mid \exists\, ll.\ \texttt{Diverge}\ ll \in \mathit{behaviours} \wedge \mathit{io_list} \preccurlyeq ll\ \}
\end{aligned}$$

3.2.6 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 3.2.1 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 3.2.1; 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`.

3.3 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.

3.3.1 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. The timestamps of the blocks are 8, 7 and 0, respectively; we will explain the purpose of timestamps in Section 3.3.2.

```
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 3.1: The definition of the DataLang state.

The runtime state of DataLang's semantics is represented by a record type `state` shown in Figure 3.1. 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 3.1) pertain to the code store, the state of the foreign function interface, and the semantic clock, respectively.

DataLang's abstract syntax (see Figure 3.2) 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 3.2, `var` (a type alias for the 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

```
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 3.2: DataLang's abstract syntax.

(`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.

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

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

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

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`.

$$\begin{array}{l}\texttt{evaluate}\,(\texttt{Assign}\; dest\; op\; args\; names\_opt, s) \stackrel{\text{def}}{=} \\ \quad \textbf{case}\;\texttt{cut\_state\_opt}\; names\_opt\; s\;\textbf{of} \\ \quad\; \texttt{Some}\; s \Rightarrow \\ \quad\; \textbf{case}\;\texttt{get\_vars}\; args\; s.\texttt{locals}\;\textbf{of} \\ \quad\;\; \texttt{Some}\; xs \Rightarrow \\ \quad\;\; \textbf{case}\;\texttt{do\_app}\; op\; xs\; s\;\textbf{of} \\ \quad\;\;\; \texttt{Rval}\,(v, s) \Rightarrow (\texttt{None}, \texttt{set\_var}\; dest\; v\; s) \\ \quad\;\;\; \mid \texttt{Rerr}\; e \Rightarrow (\texttt{Some}\,(\texttt{Rerr}\; e), s) \\ \quad\;\; \mid \ldots \Rightarrow (\texttt{Some}\,(\texttt{Rerr}\,(\texttt{Rabort}\;\texttt{Rtype\_error})), s)\end{array}$$

For a more detailed description of the DataLang semantics, including `MakeSpace` and `Call`, we refer to Tan et al. [17].

3.3.2 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 3.5.

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 = ⟪
  ...
  safe_for_space : bool
  stack_frame_sizes : num num_map;
  limits : limits;
⟫
```

```
limits = ⟪
  heap_limit : num;
  length_limit : num;
  stack_limit : num;
  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:

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

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:

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

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.

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

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.

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

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.

$$\begin{aligned}
&\texttt{size_of}\,[]\;\mathit{refs}\;\mathit{seen} \stackrel{\text{def}}{=} (0, \mathit{refs}, \mathit{seen})\\
&\texttt{size_of}\,[\texttt{Number}\;i]\;\mathit{refs}\;\mathit{seen} \stackrel{\text{def}}{=}\\
&\quad (\textbf{if}\;\texttt{is_smallnum}\;i\;\textbf{then}\;0\;\textbf{else}\;\texttt{bignum_size}\;i, \mathit{refs}, \mathit{seen})\\
&\texttt{size_of}\,[\texttt{Block}\;\mathit{ts}\;\mathit{tag}\;\mathit{vs}]\;\mathit{refs}\;\mathit{seen} \stackrel{\text{def}}{=}\\
&\quad \textbf{if}\;\mathit{vs} = [] \vee \mathit{ts} \in \mathit{seen}\;\textbf{then}\;(0, \mathit{refs}, \mathit{seen})\\
&\quad \textbf{else}\\
&\qquad \textbf{let}\;(n, \mathit{refs}', \mathit{seen}') = \texttt{size_of}\;\mathit{vs}\;\mathit{refs}\;(\{\mathit{ts}\} \cup \mathit{seen})\;\textbf{in}\\
&\qquad\quad (n + |\mathit{vs}| + 1, \mathit{refs}', \mathit{seen}')\\
&\texttt{size_of}\,(x :: \mathit{xs})\;\mathit{refs}\;\mathit{seen} \stackrel{\text{def}}{=}\\
&\quad \textbf{let}\;(n_1, \mathit{refs}_1, \mathit{seen}_1) = \texttt{size_of}\;\mathit{xs}\;\mathit{refs}\;\mathit{seen}\;;\\
&\qquad (n_2, \mathit{refs}_2, \mathit{seen}_2) = \texttt{size_of}\,[x]\;\mathit{refs}_1\;\mathit{seen}_1\;\textbf{in}\\
&\quad (n_1 + n_2, \mathit{refs}_2, \mathit{seen}_2)
\end{aligned}$$

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 =
 case lookup r refs of
  None ⇒ (0, refs, seen)
 | Some (ValueArray vs) ⇒
   (let (n, refs', seen') = size_of vs (delete r refs) seen
    in (n + |vs| + 1, refs', seen'))
 | Some (ByteArray v2 bs) ⇒
   (|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 =
 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.

3.4 Proving soundness of heap cost

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

```
... ⇒
semantics_word ffi (compile c prog)
   ⊆ extend_with_resource_limit (semantics_data ffi prog)
```

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 =.

```
... ∧ data_lang_safe_for_space ffi prog ... ⇒
semantics_word ffi (compile c prog) = semantics_data ffi prog
```

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

3.4.1 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 [17]. 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.

$$
\begin{array}{l}
\vdash \mathtt{evaluate}_{\mathsf{data}}\ (prog, s) = (res, s_1) \wedge \\
\quad \mathtt{state\_rel}\ c\ s\ t \wedge \\
\quad res \neq \mathtt{Some}\ (\mathtt{Rerr}\ (\mathtt{Rabort}\ \mathtt{Rtype\_error})) \Rightarrow \\
\quad \exists\, t_1\ res_1. \\
\qquad \mathtt{evaluate}_{\mathsf{word}}\ (\mathtt{comp}\ c\ prog, t) = (res_1, t_1) \wedge \\
\qquad (res_1 = \mathtt{Some}\ \mathtt{NotEnoughSpace} \Rightarrow \\
\qquad\quad t_1.\mathsf{ffi.io\_events} \preccurlyeq s_1.\mathsf{ffi.io\_events} \wedge \\
\qquad\quad \boxed{(c.\mathsf{gc\_kind} \neq \mathtt{None} \Rightarrow \neg s_1.\mathsf{safe\_for\_space})}\ ) \wedge \\
\qquad (res_1 \neq \mathtt{Some}\ \mathtt{NotEnoughSpace} \Rightarrow \\
\qquad\quad \mathtt{state\_rel}\ c\ s_1\ t_1 \wedge \ldots\ )
\end{array}
$$

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 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 Sec-

tion 3.2.1) 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.

3.4.2 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 3.4.3.

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_element`*s*.

$$
\begin{aligned}
&(\alpha,\ \beta)\ \texttt{heap\_element}\ = \\
&\quad \texttt{Unused num} \\
&\ \mid \texttt{ForwardPointer num}\ \alpha\ \texttt{num} \\
&\ \mid \texttt{DataElement}\ (\alpha\ \texttt{heap\_address list})\ \texttt{num}\ \beta \\
&\alpha\ \texttt{heap\_address}\ =\ \texttt{Pointer num}\ \alpha \mid \texttt{Data}\ \alpha
\end{aligned}
$$

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:

$$
\begin{aligned}
&\texttt{heap\_lookup}\ a\ [] \stackrel{\text{def}}{=} \texttt{None} \\
&\texttt{heap\_lookup}\ a\ (x :: xs) \stackrel{\text{def}}{=} \\
&\quad \textbf{if}\ a = 0\ \textbf{then}\ \texttt{Some}\ x \\
&\quad \textbf{else if}\ a < \texttt{el\_length}\ x\ \textbf{then}\ \texttt{None} \\
&\quad \textbf{else}\ \texttt{heap\_lookup}\ (a - \texttt{el\_length}\ x)\ xs \\
&\\
&\texttt{el\_length}\ (\texttt{Unused}\ l) \stackrel{\text{def}}{=} l + 1 \\
&\texttt{el\_length}\ (\texttt{ForwardPointer}\ n\ d\ l) \stackrel{\text{def}}{=} l + 1 \\
&\texttt{el\_length}\ (\texttt{DataElement}\ xs\ l\ data) \stackrel{\text{def}}{=} l + 1
\end{aligned}
$$

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`.

$$
\begin{array}{l}
\texttt{v\_inv}\ c\ (\texttt{Number}\ i)\ (x, f, t, heap) \stackrel{\text{def}}{=} \\
\quad \textbf{if}\ \texttt{is\_smallint}\ i\ \textbf{then}\ x = \texttt{Data}\ (\texttt{Word}\ (\texttt{Smallnum}\ i)) \\
\quad \textbf{else} \\
\quad\quad \exists\, ptr. \\
\quad\quad\quad x = \texttt{Pointer}\ ptr\ (\texttt{Word}\ 0w) \wedge \\
\quad\quad\quad \texttt{heap\_lookup}\ ptr\ heap = \texttt{Some}\ (\texttt{Bignum}\ i)
\end{array}
$$

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 box.

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

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

$$
\begin{array}{l}
\texttt{get\_len}\ heap\ p \stackrel{\text{def}}{=} \\
\quad \textbf{case}\ \texttt{heap\_lookup}\ p\ heap\ \textbf{of}\ \texttt{None} \Rightarrow 0 \mid \texttt{Some}\ x \Rightarrow \texttt{el\_length}\ x
\end{array}
$$

3.4.3 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.

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

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:

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

which is equivalent to:

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

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:

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

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 3.3. 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 last rule, it is worth noting

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

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

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

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

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

Figure 3.3: Definition of `traverse`.

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 $\leq$ the first component of the result of `size_of`.

$$\begin{aligned}
&\vdash \texttt{size_of}\ \mathit{values}\ \mathit{refs}\ \texttt{empty} = (n, r, s) \wedge \\
&\quad \texttt{v_inv_list}\ c\ \mathit{roots}\ (\mathit{values}, f, t, \mathit{heap}) \wedge \\
&\quad (\forall n.\ n \in \texttt{reachable_refs}\ \mathit{values}\ \mathit{refs} \Rightarrow \texttt{ref_inv}\ c\ n\ \mathit{refs}\ (f, t, \mathit{heap}, be)) \Rightarrow \\
&\qquad \exists p_2.\ \texttt{traverse}\ \mathit{heap}\ []\ \mathit{roots}\ p_2 \wedge \texttt{sum}\ (\texttt{map}\ (\texttt{get_len}\ \mathit{heap})\ p_2) \leq n
\end{aligned}$$

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:

$$\texttt{sum}\,(\texttt{map}\,(\texttt{get\_len}\,\mathit{heap})\,p_2) \leq \texttt{size\_of\_heap}\,s \wedge \\ \texttt{traverse}\,\mathit{heap}\,[]\,\mathit{roots}\,p_2$$

With this knowledge, it suffices to prove:

$$\texttt{sum}\,(\texttt{map}\,\texttt{el\_length}\,\mathit{heap}) \leq \texttt{sum}\,(\texttt{map}\,(\texttt{get\_len}\,\mathit{heap})\,p_2)$$

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:

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

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.

3.4.4 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.

3.5 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 3.3, 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.[2]

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

[2]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.

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:

$$
\begin{aligned}
&\texttt{stack\_size}\,(\texttt{StackFrame}\;n\;l\;\texttt{None::}\mathit{stack}) \stackrel{\text{def}}{=} \\
&\;\texttt{option\_binop}\,(+)\;n\;(\texttt{stack\_size}\;\mathit{stack}) \\
&\texttt{stack\_size}\,(\texttt{StackFrame}\;n\;l\;(\texttt{Some}\;\mathit{handler})\texttt{::}\mathit{stack}) \stackrel{\text{def}}{=} \\
&\;\texttt{option\_binop}\,(+)\,(\texttt{option\_map}\,((+)\;3)\;n)\;(\texttt{stack\_size}\;\mathit{stack}) \\
&\texttt{stack\_size}\,[\,] \stackrel{\text{def}}{=} \texttt{Some}\;1
\end{aligned}
$$

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 3.7 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:

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

The 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 [21] 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.

3.5.1 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.

3.6 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.

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

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 $\subseteq$ 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*.

3.7 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.

3.7.1 Is yes safe for space?

As a first example, we use a CakeML implementation of the `yes` command, shown in Figure 3.4. 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 3.4: 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.

3.7.2 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 3.2.4. 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 to the expression $\lambda\, ret.$ `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:

$$\begin{aligned}\vdash\ & \texttt{the}\,(\texttt{size\_of\_stack}\ \texttt{s.stack}) + 17 \leq \texttt{s.limits.stack\_limit} \wedge \\ & \texttt{size\_of\_heap}\ \texttt{s} + 11 \leq \texttt{s.limits.heap\_limit} \wedge \ldots \Rightarrow \\ & (\texttt{snd}\,(\texttt{evaluate}(\texttt{Seq}\,(\texttt{Call}\ldots)\,(\texttt{Call}\ldots))))\texttt{.safe\_for\_space}\end{aligned}$$

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:

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

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 3.6, can be instantiated to read:

$$\begin{aligned}\vdash\ & 56 \leq \textit{stack\_limit} \wedge 89 \leq \textit{heap\_limit} \wedge \\ & \texttt{read\_limits}\ \texttt{yes\_x64\_conf}\ \textit{ms} = (\textit{stack\_limit}, \textit{heap\_limit}) \wedge \\ & \texttt{installed}\ \texttt{yes\_code}\ldots\ \textit{ms} \wedge \ldots \Rightarrow \\ & \texttt{machine\_sem}\ldots\ \textit{ms} = \texttt{semantics\_prog}\ldots\ \texttt{yes\_prog}\end{aligned}$$

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`.

$$\begin{aligned}&\texttt{semantics\_prog}\ldots\ \texttt{yes\_prog} = \\ &\ \{\texttt{Diverge}\,(\texttt{lrepeat}\,[\texttt{put\_str\_event}\ \texttt{"y"}])\}\end{aligned}$$

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:

$$
\begin{aligned}
\vdash\ & 56 \leq \mathit{stack\_limit} \wedge 89 \leq \mathit{heap\_limit} \wedge \\
& \texttt{read\_limits}\ \texttt{yes\_x64\_conf}\ \mathit{ms} = (\mathit{stack\_limit}, \mathit{heap\_limit}) \wedge \\
& \texttt{installed}\ \texttt{yes\_code}\ \ldots\ \mathit{ms} \wedge \ldots \Rightarrow \\
& \quad \texttt{machine\_sem}\ \ldots\ \mathit{ms} = \\
& \quad\ \{\texttt{Diverge}\ (\texttt{lrepeat}\ [\texttt{put\_str\_event}\ \texttt{"y"}])\}
\end{aligned}
$$

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.

3.7.3 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 3.5.

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 3.3 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.

```
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 3.5: Implementation of `lcg`. The definition of `put_char` is elided.

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 $O(|\texttt{cs}|^2)$ heap space

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 3.5 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).

3.7.4 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 3.6: 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 3.7: 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:

$$\vdash \texttt{evaluate}\,(s, \texttt{reverse\_body}) = (res, s') \wedge \ldots \Rightarrow$$
$$\exists k.\, s'.\texttt{stack\_max} = \texttt{option\_map}\,(+\,k)\,s.\texttt{stack\_max}$$

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.

3.8 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. [29] 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. Madiot and Pottier [19] present a separation logic for reasoning about heap usage in the presence of garbage collection. 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 [28] 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 [24] 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 [12] has also been employed to formally estimate resource bounds for imperative C programs. Carbonneaux et al. [5] 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 [2, 4, 5]. 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. [30] 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.

3.9 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.

[2] 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.

[5] 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.

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

[19] J. Madiot and F. Pottier. A separation logic for heap space under garbage collection. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi: 10.1145/3498672. URL `https://doi.org/10.1145/3498672`.

[20] 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.

[21] 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.

[14] 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.

[23] 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.

[24] 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.

[16] 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.

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

[17] 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.

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

[29] 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.

[30] 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.

4

A flat reachability-based measure for CakeML's cost semantics

Alejandro. Gómez-Londoño
Magnus. O. Myreen

Symposium on Implementation and Application of Functional Languages (IFL) 2021

Abstract. The CakeML project has recently developed a verified cost semantics that allows reasoning about the space safety of CakeML programs. With this space cost semantics, compiled machine code can be proven to have tight memory bounds ensuring no out-of-memory errors occur during execution. This paper proposes a new cost semantics which is designed to make proofs about space safety significantly simpler than they were with the original version. The work described here has been developed in the HOL4 theorem prover.

4.1 Introduction

Functional languages aid the development of complex programs by providing programmers with many abstractions (e.g., polymorphism, garbage collection, ADTs, among others). However, these abstractions often come at the cost of increased memory usage and compiler complexity. These drawbacks make it difficult to judge space safety, i.e., how much memory a program will need in order to run without encountering out-of-memory errors.

To avoid out-of-memory errors, the CakeML project has recently developed a verified cost semantics [8] that makes it possible to prove the space safety of programs generated by the CakeML compiler. CakeML's cost semantics predicts when a program runs out of memory using a space measuring function, `size_of`. The `size_of` function is used at all allocation sites to check if the memory usage has surpassed a given limit. To prove that a program does not run out of memory, it is enough to show that `size_of`, as used by the formal semantics, stays within the limits.

At its core, the measuring function `size_of` uses a single recursive descent to discover reachable nodes in the heap and compute their sum (while taking aliasing into account in order to avoid gross over-approximations). Space safety proofs must then carefully use the definition and properties of `size_of` to reach Q.E.D. Unfortunately, the current formulation of `size_of` is not naturally compositional, making its definition tricky to use, and forcing related properties to require complicated assumptions to hold.

The non-compositionality of `size_of` makes reasoning about space safety a cumbersome endeavor. The majority of a space safety proof is dedicated to tedious by-hand accounting of arguments and establishing complicated assumptions. Overall, while space safety can be established using CakeML's cost semantics, its utility is severely limited by the amount of effort necessary to complete these `size_of` proofs.

This paper makes it easier to reason about CakeML's cost semantics by defining (and proving soundness of) an alternative space measuring function, called

`flat_size_of`. The new function `flat_size_of` is defined in two steps: first, it computes the set of reachable nodes, and then computes the sum of the size of the data at those nodes. The new formulation is compositional and, thus, one can express properties and conduct proofs more naturally than with `size_of`. Our initial experiments suggest that `flat_size_of` makes space safety proofs less cumbersome (i.e., require less assumptions) and more manageable (i.e., shorter theory files) than `size_of` equivalents.

This paper makes the following contributions:

- Defines `flat_size_of` (in Section 4.3) as a new reachability-based measuring function, which is significantly simpler to work with than the original `size_of` (Section 4.2.3).
- Demonstrates (in Section 4.4) how `flat_size_of` overcomes some of the most significant problems of `size_of`.
- Discusses (in Section 4.5) how `flat_size_of` was proved sound, and how future space safety proofs can use it.

All the work presented in this paper is machined-checked using the HOL4 theorem prover in the context of the CakeML compiler verification project; artifacts and example proofs can be found here.

4.2 A verified cost semantics

The cost semantics for the CakeML compiler [8] is expressed at the level of its DataLang intermediate language.

DataLang is an intermediate language approximately in the middle of the CakeML compiler. It is an imperative intermediate language with nested tuple-like values and reference pointers, but no function values. It appears right before memory becomes finite and the garbage collector is introduced. The semantics of DataLang is expressed in the form of a (functional) big-step semantics.

The semantics for DataLang acts as a cost semantics for CakeML by maintaining a boolean-valued `safe_for_space` field in the semantic state of the operational semantics. This field is set to false whenever a semantic space-cost measurement predicts that the current use of space might exceed the configured space limits for heap or stack space.

This paper focuses on the measurement of heap space. At each allocation of new memory, the semantics for DataLang computes the size of the currently

live data using a measuring function called `size_of`. This `size_of` function computes the space consumption of all values that are reachable from the root values obtained from the stack and global variables. This `size_of` function is defined to carefully track aliasing by keeping track of pointer-equal values, and is unchanged by garbage collection as it only consider live (reachable) data.

To prove the space safety of a CakeML program, one must show that for some (concrete or abstract) limit that the semantics of its DataLang representation never sets `safe_for_space` to `false`. Once space safety is established, it can be extended all the way to the level of machine code, thanks to the soundness proof for the cost semantics w.r.t. to the CakeML compiler.

The rest of this section introduces: the DataLang semantics, the space semantics, and the definition of the original heap space measure, i.e. `size_of`. More details on the original DataLang's operational and costs semantics can be found in prior work [8, 15].

4.2.1 DataLang at a glance

DataLang is an imperative language with abstract values, stateful storage of local variables, and a call stack. In the compiler-stack, it sits between the more abstract functional languages and the low-level languages with word-based value representations.

To give a sense of how CakeML programs look when compiled into DataLang, consider the following CakeML function expressed in CakeML source syntax (which is very similar to SML syntax).

```
fun app123 x = let a = [1,2,3] in a ++ x end
```

This function appends its input to the list `[1,2,3]`. The result of compiling this function to DataLang is shown in Figure 4.1.

At first, the DataLang presentation of the code might seem significantly different. However, on closer inspection, we hope the reader will see the similarity. In DataLang, the result of a primitive operation is always assigned (:=) to a local variable, which is represented as a natural number. On line 0, argument 0 corresponds to the source code binding `x`. Line 1 allocates 9 slots of space, since for each of the three `Cons` space for the constructor, head, and tail is required. Line 2 creates a value representing an empty list using the primitive operation `Cons` and a number tag (`nil_tag`) denoting the `nil` constructor for lists. On line 3, a `Const` operation creates the number literal 3. Line 4 combines local variables 1 (`[]`) and 2 (3) into the singleton list

```
line 0 :    app123 [0] evaluates as
line 1 :      MakeSpace 9
line 2 :      1 := Cons nil_tag []
line 3 :      2 := Const 3
line 4 :      3 := Cons cons_tag [2; 1]
line 5 :      4 := Const 2
line 6 :      5 := Cons cons_tag [4; 3]
line 7 :      6 := Const 1
line 8 :      7 := Cons cons_tag [6; 5]
line 9 :      8 := ListAppend [7; 0]
line 10 :     return 8
```

Figure 4.1: DataLang code for a function that appends its argument to `[1,2,3]`

`[3]` using `Cons` and the corresponding list constructor tag (`cons_tag`); using the same process, lines 5 through 8 create the DataLang representation of the list `[1,2,3]`. Then, Line 9 applies `ListAppend`—which appends the two lists-shaped values—variables 0 (the argument) and 7 (`[1,2,3]`).

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

Figure 4.2: DataLang's abstract values

Primitive values in DataLang are modeled by the data type presented in Figure 4.2. Here `Number` is 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 arrays). The `Block` constructor represents contiguous values in memory, and encodes datatype constructors, tuples and vectors.

An example of a DataLang value is shown in Figure 4.3 which shows the DataLang representation of the CakeML list `[1,2,3]`. This is the value resulting from a call to `app123` with the empty list as the argument. `Block` values, with `cons_tag` and `nil_tag` indicate the source-level constructor that each Block represents. Furthermore, timestamp values 8, 7, and 6 uniquely identify

each block.

```
app123_nil ≝ Block 8 cons_tag [Number 1;
               Block 7 cons_tag [Number 2;
                 Block 6 cons_tag [Number 3;
                   Block 0 nil_tag []]]]
```

Figure 4.3: `Block` representation of CakeML list `[1,2,3]`

The semantics state is defined as the record type shown in Figure 4.4. 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 stack frames, each frame containing only the relevant variables that should be restored after a call is completed. The `global` field contains an optional reference to an array of global variables. Space limits are kept in a record with fields for heap and stack limits. The boolean flag `safe_for_space` is set to false when space limits have been exceeded. The remaining fields are not of relevance for the presentation here.

```
α state = ⟪|
  locals : v num_map;
  refs : v ref num_map;
  stack : stack list;
  global : num option;
  limits : limits;
  safe_for_space : bool;
  clock : num;
  ...
 |⟫

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

limits = ⟪| heap_limit : num; stack_limit : num; ... |⟫
```

Figure 4.4: The definition of the DataLang state.

The semantics of DataLang is defined as a functional big-step semantics [14]. In this style of semantics, a clocked big-step evaluation function, `evaluate`, takes a (program, state) pair as input, and returns a (result, state) pair as output. As an example, consider the evaluation of `app123` with the empty list as argument, which results in value `app123_nil`. Note that the program is

given to `evaluate` as a DataLang AST (`app123_prog`) and arguments are local variables in the state.

$$\begin{aligned}&\texttt{evaluate}\,(\texttt{app123\_prog}, s\ \textbf{with}\ \text{locals} := \{\,0 \mapsto \texttt{Block}\ 0\ \texttt{nil\_tag}\ []\,\})\\ &= (\texttt{app123\_nil}, s')\end{aligned}$$

To better visualize intermediate steps of evaluation, the DataLang semantics can also be expressed as a shallowly embedded state-exception monad. This is the representation used in `app123` and by partially evaluating the first three operations we can inspect its intermediate state:

$$\begin{aligned}&\texttt{app123}\,(s\ \textbf{with}\ \text{locals} := \{\,0 \mapsto \texttt{Block}\ 0\ \texttt{nil\_tag}\ []\,\})\\ &= (\,4 := \texttt{Const}\ 2\\ &\quad 5 := \texttt{Cons}\ \texttt{cons\_tag}\ [4;\,3]\\ &\quad 6 := \texttt{Const}\ 1\\ &\quad 7 := \texttt{Cons}\ \texttt{cons\_tag}\ [6;\,5]\\ &\quad 8 := \texttt{ListAppend}\ [7;\,0]\\ &\quad \texttt{return}\ 8)\\ &\quad s\ \textbf{with}\ \langle\!|\ \text{locals} := \{\,0 \mapsto \texttt{Block}\ 0\ \texttt{nil\_tag}\ []\\ &\qquad\qquad\qquad 1 \mapsto \texttt{Block}\ 0\ \texttt{nil\_tag}\ []\\ &\qquad\qquad\qquad 2 \mapsto \texttt{Number}\ 3\\ &\qquad\qquad\qquad 3 \mapsto \texttt{Block}\ 3\ \texttt{cons\_tag}\\ &\qquad\qquad\qquad\qquad [\texttt{Number}\ 3;\\ &\qquad\qquad\qquad\qquad\ \texttt{Block}\ 0\ \texttt{nil\_tag}\ []]\,\};\\ &\qquad\qquad \ldots\\ &\qquad\quad |\!\rangle\end{aligned}$$

4.2.2 Embedded cost semantics

As previously stated, DataLang's costs semantics is embedded into its operational semantics. Therefore, proving space safety of `app123` is a matter of proving the following statement:

$$\begin{aligned}&\vdash s.\text{limits.heap\_limit} = mh \wedge\\ &\quad s.\text{limits.stack\_limit} = ms \wedge\\ &\quad s.\text{safe\_for\_space} \wedge\\ &\quad \texttt{evaluate}\,(\texttt{app123\_prog}, s) = (res, s') \implies\\ &\quad\ s'.\text{safe\_for\_space}\end{aligned}$$

This is, given stack space mh and heap space ms; the evaluation of `app123_prog` preserves safe_for_space, thus signalling that the program's memory consumption falls within the given bounds.

Internally, the safe_for_space flag is updated at every space-consuming operation, for example, at function calls and whenever new values are created. Auxiliary functions `size_of_heap` and `size_of_stack` are used to update safe_for_space in one of two ways. If k slots of new heap space are to be used (e.g. as part of `MakeSpace`), then `safe_for_space` is updated as follows:

$$
\begin{array}{l}
s \textbf{ with } \text{safe\_for\_space} := \\
\quad (s.\text{safe\_for\_space} \wedge \\
\quad \texttt{size\_of\_heap}\ s + k \leq s.\text{limits.heap\_limit})
\end{array}
$$

Similarly, if k slots of new stack space are to be consumed (e.g. as part of a function call), then `safe_for_space` is updated as follows:

$$
\begin{array}{l}
s \textbf{ with } \text{safe\_for\_space} := \\
\quad (s.\text{safe\_for\_space} \wedge \\
\quad \texttt{size\_of\_stack}\ s + k \leq s.\text{limits.stack\_limit})
\end{array}
$$

The important work is performed by the `size_of_heap` and `size_of_stack` functions. This paper focuses on improving the formulation of the heap space measure and thus `size_of_heap`.

The original formulation of `size_of_heap` is shown below. Here `stack_to_vs` is a function that computes a list of root values from local variables (s.locals), the call-stack (`extract_stack`), and global references (`global_to_vs`). The root values are given to the measuring function `size_of`, which computes the size of heap elements reachable from these initial elements.

$$
\begin{array}{l}
\texttt{size\_of\_heap}\ s \stackrel{\text{def}}{=} \\
\quad \textbf{let } (n,\_,\_) = \\
\quad\quad \texttt{size\_of}\ (\texttt{stack\_to\_vs}\ s)\ s.\text{refs}\ \emptyset \textbf{ in} \\
\quad n \\
\\
\texttt{stack\_to\_vs}\ s \stackrel{\text{def}}{=} \\
\quad \texttt{toList}\ s.\text{locals} +\!+ \\
\quad \texttt{extract\_stack}\ s.\text{stack} +\!+ \\
\quad \texttt{global\_to\_vs}\ s.\text{global}
\end{array}
$$

The main workhorse of this definition is the `size_of` function, which is the topic of the next section.

4.2.3 The original heap measure: size_of

At the core of DataLang's cost semantics is the heap space measuring function `size_of`. This function is responsible for computing the space consumed by

all values reachable from the initial list of root values. Figure 4.5 shows its definition with *seen* (a set of timestamps), and *refs* as additional arguments.

```
size_of [] refs seen ≝ (0,refs,seen)
size_of (x::xs) refs seen ≝
 let (n1,refs1,seen1) = size_of xs refs seen ;
     (n2,refs2,seen2) = size_of [x] refs1 seen1 in
  (n1 + n2,refs2,seen2)
size_of [Word64 v0] refs seen ≝ (3,refs,seen)
size_of [Number i] refs seen ≝
 (if is_smallnum i then 0 else bignum_size i,refs,seen)
size_of [CodePtr v1] refs seen ≝ (0,refs,seen)
size_of [RefPtr r] refs seen ≝
 case lookup r refs of
  None ⇒ (0,refs,seen)
 | Some (ValueArray vs) ⇒
  (let (n,refs',seen') = size_of vs (delete r refs) seen in
    (n + |vs| + 1,refs',seen'))
 | Some (ByteArray v2 bs) ⇒
  (|bs| div (arch_size lims div 8) + 2,delete r refs,seen)
size_of [Block ts tag vs] refs seen ≝
 if vs = [] ∨ isSome (lookup ts seen) then (0,refs,seen)
 else
  let (n,refs',seen') = size_of vs refs (insert ts () seen) in
   (n + |vs| + 1,refs',seen')
```

Figure 4.5: Definition of `size_of`.

The measurement of most values (`CodePtr`, `Word64`, and `Number`) is straightforward, as it is either constant, already accounted for within another structure (e.g. stack frames), or measured by a function without considering other values. By contrast, the handling of `Block` and `RefPtr` requires additional bookkeeping to avoid counting the same value twice (aliasing), and as such, is where most of the complexity of `size_of` lies. In the case of `Block` values, a set of already-measured (*seen*) timestamps is kept to avoid counting identical blocks multiple times; this mechanisms relies on a bijection between timestamps and the blocks in memory. For `RefPtr`, pointers are removed from references map (*refs*) once they are counted, this is to only follow a reference once.

To illustrate how `size_of` handles aliasing consider the following examples:

With x equal to `Block` *ts tag* [`Number` 1] throughout:

(B1) $ts \notin seen \Rightarrow$ `size_of` $[x]\ refs\ seen = (2, refs, \{ts\} \cup seen)$

(B2) $ts \in seen \Rightarrow$ `size_of` $[x]\ refs\ seen = (0, refs, seen)$

(B3) $ts \notin seen \wedge ts \in seen' \wedge$
`size_of` $xs\ refs\ seen = (n, refs', seen') \Rightarrow$
`size_of` $(x{::}xs)\ refs\ seen = (n, refs', seen')$

Intuitively, blocks whose timestamps have not been "seen" (i.e., $ts \notin seen$) are always counted (B1). Moreover, blocks with already "seen" timestamps are ignored (B2), as this hints at the block being present in previous values (B3) at the back of the list — since `size_of` operates from the back.

The definition of `size_of` succeeds at providing tight bounds, mitigating the effects of aliasing, and traversing only live data; however, perhaps due to its precise and concrete nature, it can be challenging to reason about. The main hurdle with `size_of` is the linearity of its traversal, where initial measurements at the back of the argument list directly affect subsequent ones through pointers or timestamps — as in example (B3). Thus, conceptually simple properties (e.g., the reordering of values) are hard to prove and apply. The shortcomings of `size_of` are explained in more detail in Section 4.4.

4.3 A new flat reachability-based measurement

This section shows the definition of our new heap cost measuring function, `flat_size_of`, which improves on the original `size_of`. In a nutshell, `flat_size_of` takes a set of root addresses, computes the set of all addresses reachable from that initial set (Section 4.3.1), and then sums the sizes of all heap elements at those addresses (Section 4.3.2). We refer to this new formulation as "flat" because operations occur mostly over sets and avoid recursing into the structure of values. The rest of this section goes into the details of the definition of `flat_size_of`.

4.3.1 The set of all reachable addresses

DataLang has no immediate notion of heap address. For the purposes of the definition of `flat_size_of`, we define a type for DataLang addresses. Intuitively, an address is meant to represent any value that might contain or reference other values. Therefore, we represent an address as either the

timestamp (`TStamp`) of a Block (remember each block has a unique timestamp) or the pointer to a reference (`RStamp`).

$$\texttt{addr} = \texttt{TStamp num} \mid \texttt{RStamp num}$$

From a list of DataLang values, we can compute, using `to_addrs`, a set of corresponding addresses. Note that `to_addrs` does not recurse into `Block` values, because it only wants to collect the immediately reachable addresses of the given values.

$$\begin{aligned}
&\texttt{to_addrs}\ [] \stackrel{\text{def}}{=} \emptyset \\
&\texttt{to_addrs}\ (\texttt{Block}\ ts\ tag\ []::xs) \stackrel{\text{def}}{=} \texttt{to_addrs}\ xs \\
&\texttt{to_addrs}\ (\texttt{Block}\ ts\ tag\ (v::vs)::xs) \stackrel{\text{def}}{=} \\
&\quad \{\ \texttt{BlockAddr}\ ts\ \} \cup \texttt{to_addrs}\ xs \\
&\texttt{to_addrs}\ (\texttt{RefPtr}\ ref::xs) \stackrel{\text{def}}{=} \\
&\quad \{\ \texttt{RefAddr}\ ref\ \} \cup \texttt{to_addrs}\ xs
\end{aligned}$$

Omitted value kinds in the definition of `to_addrs` do not have addresses in this representation; in those cases, recursion directly continues through the list.

As a precursor to reachability, we define the `next` relation which consider pairs of addresses that are one-step reachable. When provided with value mappings for pointers (*refs*) and timestamps (*blocks*), the relation `next` *refs blocks a b* holds only if *b* is immediately reachable from *a* using one of such mappings.

$$\begin{aligned}
&\texttt{next}\ refs\ blocks\ (\texttt{TStamp}\ ts)\ r \stackrel{\text{def}}{=} \\
&\quad r \in \texttt{block_to_addrs}\ blocks\ ts \\
&\texttt{next}\ refs\ blocks\ (\texttt{RStamp}\ ref)\ r \stackrel{\text{def}}{=} \\
&\quad r \in \texttt{ptr_to_addrs}\ refs\ ref
\end{aligned}$$

$$\begin{aligned}
&\texttt{block_to_addrs}\ blocks\ ts \stackrel{\text{def}}{=} \\
&\quad \textbf{case}\ \texttt{lookup}\ ts\ blocks\ \textbf{of} \\
&\quad \mid \texttt{Some}\ (\texttt{Block}\ _\ _\ vs) \Rightarrow \texttt{to_addr}\ vs \\
&\quad \mid _ \Rightarrow \emptyset
\end{aligned}$$

$$\begin{aligned}
&\texttt{ptr_to_addrs}\ refs\ p \stackrel{\text{def}}{=} \\
&\quad \textbf{case}\ \texttt{lookup}\ p\ refs\ \textbf{of} \\
&\quad \mid \texttt{Some}\ (\texttt{ValueArray}\ vs) \Rightarrow \texttt{to_addr}\ vs \\
&\quad \mid _ \Rightarrow \emptyset
\end{aligned}$$

Therefore, from an initial set of addresses we can neatly describe all reachable addresses using the reflexive transitive closure (*) of `next`. Note that

this approach implicitly handles aliasing by declaratively defining the set of reachable addresses; avoiding non-termination concerns and ruling out duplicated values.

$$\texttt{reachable\_v}\ \mathit{refs}\ \mathit{blocks}\ \mathit{roots} \stackrel{\text{def}}{=} \\ \{\, y \mid \exists x.\ x \in \mathit{roots} \wedge (\texttt{next}\ \mathit{refs}\ \mathit{blocks})^*\ x\ y \,\}$$

With these functions we can state the set of all reachable addresses from a list of root values as follows.

$$\texttt{reachable\_v}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{to\_addrs}\ \mathit{roots})$$

Crucially, the result of `reachable_v` is only finite if the initial set of roots is finite; a requirement to iterate on the resulting set in subsequent functions. Fortunately, the result of `to_addrs` *roots* is know to be finite as it only turns the finitely many elements of *roots* into addresses. Thus, one can prove the following.

$$\vdash \texttt{FINITE}\ (\texttt{reachable\_v}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{to\_addrs}\ \mathit{roots}))$$

4.3.2 Adding it all up

In order to sum the sizes of all the reachable values, we need a function that can compute the heap space consumed by a heap element at a specific address. For this purpose, we define a function `size_of_addr` which given an address returns the size of that heap element.

$$\begin{array}{l}
\texttt{size_of_addr}\ \mathit{lims}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{TStamp}\ \mathit{ts}) \stackrel{\text{def}}{=} \\
\quad \textbf{case}\ \texttt{lookup}\ \mathit{ts}\ \mathit{blocks}\ \textbf{of} \\
\quad\quad \texttt{Some}\ (\texttt{Block}\ _\ _\ \mathit{vs}) \Rightarrow \\
\quad\quad\quad 1 + |\mathit{vs}| + \texttt{sum}(\texttt{map}\ (\texttt{flat_measure}\ \mathit{lims})\ \mathit{vs}) \\
\quad \mid _ \Rightarrow 0 \\
\texttt{size_of_addr}\ \mathit{lims}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{RStamp}\ p) \stackrel{\text{def}}{=} \\
\quad \textbf{case}\ \texttt{lookup}\ p\ \mathit{refs}\ \textbf{of} \\
\quad\quad \texttt{None} \Rightarrow 0 \\
\quad \mid \texttt{Some}\ (\texttt{ValueArray}\ \mathit{vs}) \Rightarrow \\
\quad\quad\quad 1 + |\mathit{vs}| + \texttt{sum}(\texttt{map}\ (\texttt{flat_measure}\ \mathit{lims})\ \mathit{vs}) \\
\quad \mid \texttt{Some}\ (\texttt{ByteArray}\ _\ \mathit{bs}) \Rightarrow \\
\quad\quad\quad |\mathit{bs}|\ \texttt{div}\ (\texttt{arch_size}\ \mathit{lims}\ \texttt{div}\ 8) + 2
\end{array}$$

In the definition above, we see that an address of a `Block` *t n vs* has size $1 + |\mathit{vs}| + \texttt{sum}\ (\texttt{map}\ (\texttt{flat\_measure}\ \texttt{lims})\ \mathit{vs})$. Here 1 is the space for the header

$$\begin{aligned}
&\texttt{flat_measure}\ lims\ (\texttt{Word64}\ v_0) \stackrel{\text{def}}{=} 3\\
&\texttt{flat_measure}\ lims\ (\texttt{Number}\ i) \stackrel{\text{def}}{=}\\
&\ \textbf{if}\ \texttt{small_num}\ lims.\text{arch_64_bit}\ i\ \textbf{then}\ 0\\
&\ \textbf{else}\ \texttt{bignum_size}\ lims.\text{arch_64_bit}\ i\\
&\texttt{flat_measure}\ lims\ (\texttt{Block}\ v_5\ v_6\ v_7) \stackrel{\text{def}}{=} 0\\
&\texttt{flat_measure}\ lims\ (\texttt{CodePtr}\ v_8) \stackrel{\text{def}}{=} 0\\
&\texttt{flat_measure}\ lims\ (\texttt{RefPtr}\ v_9) \stackrel{\text{def}}{=} 0
\end{aligned}$$

Figure 4.6: The definition of `flat_measure`

of the heap element; $|vs|$ is for the length of the payload of the heap element; and `flat_measure lims` vs is to account for the heap elements that are immediately reachable from this block, but have no address. The definition of `flat_measure`, shown in Figure 4.6, counts `Block` and `RefPtr` values as having zero size, because they are already counted elsewhere.

Now we have a way to compute the set of reachable addresses and a way to compute the size of a heap element at each address. Our final definition makes use of $\sum$ which sums the application of a given function f to all elements of a finite set s.

$$\sum f\ s \stackrel{\text{def}}{=} \texttt{fold\_set}\ (\lambda e\ acc.\ f\ e + acc)\ s\ 0$$

The top-level definition of the new heap measure is the following. This definition sums the size of all `Word64` and large `Number` values in the roots using `flat_measure`. This is added to $\sum$ of `size_of_addr` applied to every reachable address in the heap.

$$\begin{aligned}
&\texttt{flat_size_of}\ lims\ refs\ blocks\ roots \stackrel{\text{def}}{=}\\
&\ \texttt{sum}\ (\texttt{map}\ (\texttt{flat_measure}\ lims)\ roots) +\\
&\ \textstyle\sum\ (\texttt{size_of_addr}\ lims\ refs\ blocks)\\
&\quad (\texttt{reachable_v}\ refs\ blocks\ (\texttt{to_addrs}\ roots))
\end{aligned}$$

Even though this definition is very different in formulation from the original `size_of`, shown in Figure 4.5, it computes the same number while providing various advantages. Aliasing is implicitly handled and there is no need for book-keeping of pointers and timestamps. Moreover, the clear separation between the gathering (`reachable_v`) and measuring (`size_of_addr`,`flat_measure`) of heap elements makes for a more concise definition than combining both operations in a single recursive descent. More generally, the main advantage of the `flat_size_of` approach is that it abstracts the structure of the heap into

a model (the set of all reachable addresses) that is considerably easier to operate over; this is in stark contrast of `size_of`, which operates directly on the structure of the heap and therefore must deal with its associated complexity.

4.3.3 Requirements

In order for `flat_size_of` to be a viable replacement of `size_of`, some support in DataLang's semantics is required. Specifically, the semantic state must provide suitable values for the auxiliary arguments *lims*, *refs*, and *blocks*. However, in the current semantics, only s.limits (*lims*) and s.refs (*refs*) are available.

To add support for `flat_size_of` to the semantics, we extended the semantic state to include a mapping from timestamps to blocks: s.all_blocks. This field is updated every time a block is created, adding a mapping between the block's timestamp and the block itself (i.e., $ts \mapsto \texttt{Block}\ ts\ tag\ l$). Since timestamps uniquely identify blocks, the mapping in s.all_blocks is always consistent with all blocks in the heap, and by extension, all addresses derived by `reachable_v`.

Given this set up, one can define the top-level cost measuring function `flat_size_of_heap` in a way similar to `size_of_heap`.

$$\begin{aligned}&\texttt{flat\_size\_of\_heap}\ s \stackrel{\text{def}}{=}\\ &\ \texttt{flat\_size\_of}\ s.\text{limits}\ s.\text{refs}\ s.\text{all\_blocks}\ (\texttt{stack\_to\_vs}\ s)\end{aligned}$$

4.4 flat_size_of is better than size_of

To illustrate the challenges of reasoning about `size_of`, consider the following reordering property:

$$\texttt{size\_of}\ [x,y]\ refs\ \emptyset\ =\ \texttt{size\_of}\ [y,x]\ refs\ \emptyset$$

Intuitively, this property must hold for a measuring function as the values considered are the same. However, with `size_of` both sides of the equality might perform completely different traversals:

$$\begin{aligned}&\texttt{size\_of}\ [y]\ refs\ \emptyset\ =\ (n_{y1}, refs_{y1}, seen_{y1})\ \wedge\\ &\texttt{size\_of}\ [x]\ refs\ \emptyset\ =\ (n_{x1}, refs_{x1}, seen_{x1})\ \wedge\\ &\texttt{size\_of}\ [y]\ refs_{x1}\ seen_{x1}\ =\ (n_{y2}, refs_{y2}, seen_{y2})\ \wedge\\ &\texttt{size\_of}\ [x]\ refs_{y1}\ seen_{y1}\ =\ (n_{x2}, refs_{x2}, seen_{x2})\ \Rightarrow\\ &\ (n_{y1}+n_{x2}, refs_{x2}, seen_{x2})\ =\ (n_{x1}+n_{y2}, refs_{y2}, seen_{y2})\end{aligned}$$

This mismatch exposes the following problems:

- There is no straightforward relation between the two measurements of $[x]$ (or those of $[y]$) as `size_of` is applied to different arguments.

- All blocks in $[x]$ and $[y]$ with the same timestamps must have the same contents; otherwise, the order in which blocks are counted will affect the result due to aliasing mitigation.

These issues can be overcome by introducing well-formedness conditions on $[x]$ and $[y]$, and by generalizing the property statement to one more suited for induction (e.g. list permutations). However, these kinds of hurdles appear more often than one might want for such a crucial function.

In stark contrast, reordering can be trivially proved for the new `flat_size_of` function. First, a call to `flat_measure` traverses a list to add non-root values, and is thus unaffected by permutations. Similarly, the initial root set computed by `to_addr` is the union of all addresses in the list of values and is again unaffected by reordering. Therefore, the remaining application of $\sum$ is being applied to the same arguments.

This ease of reasoning is what makes `flat_size_of` better suited for proofs of space safety as shown in the rest of this section.

4.4.1 A layout for space safety proofs

As mentioned before, to prove the space safety of a DataLang program one must show the preservation of safe_for_space through its evaluation (Section 4.2.2). As most DataLang programs are composed of multiple recursive functions, it is often necessary to separately prove space safety for some of them. To prove a function is space safe, one generally needs three kinds of assumptions:

(A1) The space consumption before the function call is below the limits or roughly `size_of_heap` $s + M \leq$ heap_limit, where M is any extra space the function body needs.

(A2) A description of the arguments to the function, e.g., a list-shaped block, a number within 0 and 255, among others.

(A3) That the function is defined in s.code and its body corresponds with the code being evaluated

Resulting in the following layout:

$$\begin{array}{l}\vdash \mathsf{A1} \wedge \mathsf{A2} \wedge \mathsf{A3} \wedge \\ \quad s.\mathsf{safe\_for\_space} \wedge \\ \quad \mathtt{evaluate}\,(\mathtt{fun\_body}, s) = (res, s') \Rightarrow \\ \quad\; s'.\mathsf{safe\_for\_space}\end{array}$$

Proofs are by complete induction on the semantic clock and symbolic evaluation of the function body. Assumption (A2) should allow the evaluation of most of the function body. Moreover, intermediate updates to safe_for_space can be resolved using (A1). Once the recursive call is reached, assumption (A3) replaces the function call with the function's body such that the inductive hypothesis can be applied. At this point in the proof, assumptions must be established again for the state at the function call. (A3) is trivial as s.code does not change. (A2) might require work, but well-formed function code correctly operates on its values and thus provides good arguments. The proof of (A1) shown below is where things are most likely to become tricky:

$$\begin{array}{l}\vdash \ldots \\ \quad \mathtt{size\_of\_heap}\, s + M\, s \leq s.\mathsf{limits.heap\_limit} \Rightarrow \\ \quad\; \mathtt{size\_of\_heap}\, s' + M\, s' \leq s'.\mathsf{limits.heap\_limit}\end{array}$$

Here, we must show that the space required at the recursive call ($\mathtt{size\_of\_heap}\, s' + M\, s'$) is still less than heap_limit, assuming the space was enough in the original call. This amounts to proving that the required space decreases as the function recurses:

$$\begin{array}{l}\vdash \ldots \Rightarrow \\ \quad \mathtt{size\_of\_heap}\, s' + M\, s' \leq \mathtt{size\_of\_heap}\, s + M\, s\end{array}$$

This follows the intuition that function calls should take either progressively less space, or require an extra amount of memory bounded by M.

4.4.2 A hypothetical tail-recursive example

Consider a hypothetical tail-recursive function `ftail` with the following features:

- Takes a list of numbers as argument.
- Operates over the head of the list consuming constant space.
- Makes a tail-recursive call with the tail of the list.

Now assume we want to prove `ftail` space safe for concrete argument `[1,2,3]`. Instantiating the proof layout from the previous section, we arrive at the proof goal shown below:

$$
\begin{array}{l}
\vdash \texttt{size\_of\_heap}\ s\ +\ C\ \leq\ s.\text{limits.heap\_limit}\ \wedge \\
\quad \texttt{lookup}\ ''\text{ftail}''\ s.\text{code} = \texttt{Some}\ \texttt{ftail\_body}\ \wedge \\
\quad s.\text{locals} = \\
\qquad \{\, 0\ \mapsto\ \texttt{Block}\ 8\ \texttt{cons\_tag}\ [\texttt{Number}\ 1, \\
\qquad\qquad \texttt{Block}\ 7\ \texttt{cons\_tag}\ [\texttt{Number}\ 2, \ldots]]\} \ \wedge \\
\quad s.\text{safe\_for\_space}\ \wedge \\
\quad \texttt{evaluate}\ (\texttt{ftail\_body}, s) = (\mathit{res}, s')\ \Rightarrow \\
\qquad s'.\text{safe\_for\_space}
\end{array}
$$

Above, C is the (constant) space the function uses to operate.

Using assumptions (A1), (A2), and (A3), most of the proof can proceed by evaluation; until the tail recursive call to `ftail` is reached and we must establish assumption (A1) again, leading to an inequality of the form:

$$\texttt{size\_of\_heap}\ s' \ \leq\ \texttt{size\_of\_heap}\ s$$

Which by definition of `size_of_heap` and *rest* (an abbreviation of expression `extract_stack` s.stack ++ `global_to_vs` s.global) simplifies to:

$$
\begin{array}{l}
\texttt{size\_of}\ ([\texttt{Block}\ 7\ \texttt{cons\_tag}\ [\texttt{Number}\ 2, \ldots]]] {+}{+}\ \mathit{rest}) \\
\qquad s.\text{refs}\ \emptyset \\
\leq \\
\texttt{size\_of}\ ([\texttt{Block}\ 8\ \texttt{cons\_tag}\ [\texttt{Number}\ 1, \\
\qquad\quad \texttt{Block}\ 7\ \texttt{cons\_tag}\ [\texttt{Number}\ 2, \ldots]]] {+}{+}\ \mathit{rest}) \\
\qquad s.\text{refs}\ \emptyset
\end{array}
$$

Moreover, since `size_of` operates from the back of the list, we can abstract away *rest* at both sides as `size_of` *rest* s.refs $\emptyset = (n, \mathit{refs}, \mathit{seen})$, and rewritten to:

$$
\begin{array}{l}
\texttt{size\_of}\ [\texttt{Block}\ 7\ \texttt{cons\_tag}\ \ldots]\ \mathit{refs}\ \mathit{seen}\ \leq \\
\quad \texttt{size\_of}\ [\texttt{Block}\ 8\ \texttt{cons\_tag}\ \ldots]\ \mathit{refs}\ \mathit{seen}
\end{array}
$$

At this point, it would appear that the proof is almost done, as we are essentially testing if the space occupied by a list (`[1,2,3]`) is greater than that of its tail (`[2,3]`), a rather intuitive claim. However, due to `size_of`'s handling of timestamps and the fact that *seen* is symbolic, one can not show this inequality without additional assumptions. Concretely, one can think of a scenario

where only 8 is in *seen* and no other timestamps in the block is in *seen*. Such a situation will result in the measurement being 0 at the right of the inequality and 4 on the left, a clear falsehood.

$$
\begin{aligned}
&8 \in \mathit{seen} \wedge 7 \notin \mathit{seen} \wedge \ldots \wedge \\
&\texttt{size\_of}\ [\texttt{Block}\ 7\ \ldots]\ \mathit{refs}\ \mathit{seen} = (4, \mathit{refs}', \mathit{seen}')\ \wedge \\
&\texttt{size\_of}\ [\texttt{Block}\ 8\ \ldots]\ \mathit{refs}\ \mathit{seen} = (0, \mathit{refs}'', \mathit{seen}'')\ \Rightarrow \\
&4 \leq 0
\end{aligned}
$$

Therefore, the proof goal must be extended with a predicate ensuring that if timestamps 8 is in *seen* it must be the case that 7 and all other subsequent timestamps in the block are also in *seen*.

Proving such results and all their associated lemmas takes considerable work, to the point that, similar mechanisms in existing space safety proofs take around 25% of the proof script. The issue is further aggravated by the fact that these kinds of results can not be easily generalized for all types of values and must be re-written every time a new type is used.

By switching our reasoning to `flat_size_of`, our proof goal is greatly simplified:

$$
\begin{aligned}
&\texttt{flat\_size\_of}\ s.\mathsf{refs}\ s.\mathsf{all\_blocks}\ ([\texttt{Block}\ 7\ \ldots] +\!\!+ \mathit{rest}) \\
&\leq \\
&\texttt{flat\_size\_of}\ s.\mathsf{refs}\ s.\mathsf{all\_blocks}\ ([\texttt{Block}\ 8\ \ldots] +\!\!+ \mathit{rest})
\end{aligned}
$$

While we can no longer "drop" *rest* from the roots, `flat_size_of` more than makes up for this with its use of sets and relations to represent the reachable memory. To showcase this, consider the following lemma, which states that if the reachable set of addresses from two roots x and y are subsets, and `flat_measure` then the space measurement of x done by `flat_size_of` must be less than that of y.

$$
\begin{aligned}
&\texttt{flat\_measure}\ \mathit{lims}\ x \leq \texttt{flat\_measure}\ \mathit{lims}\ y\ \wedge \\
&\texttt{reachable\_v}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{to\_addrs}\ x) \subseteq \\
&\quad \texttt{reachable\_v}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{to\_addrs}\ y)\ \Rightarrow \\
&\texttt{flat\_size\_of}\ \mathit{lims}\ \mathit{refs}\ \mathit{blocks}\ x\ \leq \\
&\quad \texttt{flat\_size\_of}\ \mathit{lims}\ \mathit{refs}\ \mathit{blocks}\ y
\end{aligned}
$$

Using this lemma the proof goal becomes trivial:

$$
\begin{aligned}
&\{\texttt{TStamp 7},\ldots\} \cup \texttt{reachable\_v}\ \ldots\ (\texttt{to\_addrs}\ \mathit{rest}) \\
&\quad \subseteq \{\texttt{TStamp 8}, \texttt{TStamp 7}, \ldots\} \cup \\
&\qquad \texttt{reachable\_v}\ \ldots\ (\texttt{to\_addrs}\ \mathit{rest})
\end{aligned}
$$

One can then conclude the proof using basic set reasoning.

It is this ease of reasoning in the presence of (possibly) aliased values that makes `flat_size_of` a suitable measuring function for a cost semantics. In particular, the reachability-based approach to gathering live data aids the function, and its reasoning, to not be concerned with where in the heap structure a value is located, and focus solely on its effect on the space measurement. In contrast, reasoning about `size_of` constantly requires additional safeguards and guarantees on the heap structure to be able to relate two measurements, as seen in our previous example.

4.4.3 A concrete tail-recursive example

Consider the CakeML function `sum` defined below:

```
fun sum xs = foldl (+) 0 xs
```

Where `xs` is a list of (unbounded) integers and `(+)` is integer addition with support for bignum arithmetic. As expected, a call to `sum` computes the addition of all the elements of `xs`.

The space safety of `sum` follows from a similar intuition as the one presented for `ftail` (Section 4.4.2) even after considering the space consumption of bignum arithmetic (+) and accumulator arguments (`foldl`). This relation is made evident by the space safety proof of `sum` currently available in the CakeML project, which shares `ftail`'s proof structure, and thus, its issues regarding the use of `size_of`. Specifically, the proof requires additional assumptions and theorems to enforce the timestamps in `xs`'s representation are correctly traversed — i.e., it is never the case that a timestamp at the head of the list has been "seen" and one in the tail has not.

Fortunately, as with `ftailm` the space safety proof of `sum` can also be improved by switching to `flat_size_of`. As an experiment we updated the proof of the `sum` example to use `flat_size_of` and the following quantitative improvements (in LOC) where archived:

- Assumptions outside of the scope of (A1), (A2), and (A3) were removed.
- 14 auxiliary lemmas and definitions were removed.
- The section of the proof dedicated to re-establishing (A1) shrunk by 32%.
- The proof of space safety shrunk by 13%.
- The entire file for this proof and all auxiliary lemmas shrunk by 28%.

Furthermore, the new proof text for `flat_size_of` only utilized definitions and standard set reasoning leading to a nicer proof.

In summary, this updated space safety proof demonstrates the advantages of `flat_size_of` over `size_of`.

4.5 Soundness

We have proved `flat_size_of_heap` sound. More specifically, we have proved that, under reasonable assumptions `size_inv` s, the number computed by `flat_size_of_heap` is equal to the number computed by `size_of_heap`.

$$\vdash \texttt{size\_inv}\ s \Rightarrow\ \texttt{size\_of\_heap}\ s = \texttt{flat\_size\_of\_heap}\ s$$

The `size_inv` assumption ensures that the values in s.all_blocks are consistent with those in the heap (i.e,. s.refs and `stack_to_vs`). Specifically, that for any `Block` *ts tag l* reachable in the heap, there is an entry $ts \mapsto$ `Block` *ts tag l* in s.all_blocks. Our proof of soundness requires `size_inv` because `flat_size_of`, unlike `size_of`, does not recurse over block values and instead must rely on an accurate block mapping to obtain the same results.

Using the proof of equality between `size_of_heap` and `flat_size_of_heap` one can rephrase any space safety proof, previously involving `size_of`, to be in terms of `flat_size_of`.

4.5.1 Updates to CakeML's cost semantics

The CakeML's cost semantics was updated to facilitate the usage of `flat_size_of` in space safety proofs.

The main hurdle when switching to `flat_size_of` is establishing `size_inv` (so the soundness theorem can apply). To address this, we extended (at the DataLang level) how s.safe_for_space is updated to include `size_inv` as an antecedent.

$$\begin{array}{l} s \textbf{ with} \\ \text{safe\_for\_space} := \\ \quad (s.\text{safe\_for\_space} \wedge \\ \quad\ (\texttt{size\_inv}\ s \Rightarrow \\ \quad\quad \texttt{size\_of\_heap}\ s + k \leq\ s.\text{limits.heap\_limit})) \end{array}$$

With this change, if one starts a typical space safety proof (Section 4.4.1) and uses `flat_size_of_heap`, instead of `size_of_heap`, for the (A1) assumption

(i.e., heap measurement are within the limits), then, whenever *s*.safe_for_space needs to be re-established `size_inv` will be available as an assumption.

The addition of `size_inv` to the cost semantics was proven sound w.r.t. the rest of the compiler, as an update to `size_of_heap`'s original soundness proof. Informally, if `size_inv` holds for the initial semantic state and is preserved by the semantic as an invariant, then, its addition as an antecedent in *s*.safe_for_space does not affect the field's value. Therefore, the addition of `size_inv` makes proofs more convenient while keeping the semantics essentially unchanged.

4.6 Related Work

Verified cost semantics are available for the CompCert [12] and CakeML [11] verified compilers. Carbonneaux et al. [5] develop a source level logic for stack space reasoning that translates to the CompCert compiler output. Besson *et al.* extends CompCert's memory model with finite memory and integer pointers in CompCertS [2, 4, 5], which allows for memory usage estimates of C functions that are proven to be bounds of the compiled code.

In recent work, Madiot and Pottier [13] develop a separation logic for conveniently reasoning about heap space usage in the presence of garbage collection. However, their cost semantics is not proved correct w.r.t. a concrete compiler.

There have been many other approaches to source-level analysis of space cost. For example, resource-aware type systems based on refinement types [6, 7, 10] can be used to obtain bounds for source programs. Moreover, a program's resource usage can be directly encoded as a refinement type in compilers with support for such type systems [9]. Time-complexity annotations and indexes in types [16] can also be used to express costs. Another approach is for proof-carrying code to be equipped with a resource usage proof w.r.t. a resource-aware program logic [1]. In general, theses approaches provide formal estimates of costs for source-level programs, however, they forgo the effects compilation and program transformation can have on resource consumption. Source-level cost analysis techniques could be used on DataLang programs to facilitate reasoning further, however, we have not yet investigated this approach.

4.7 Conclusion

In this paper we have proposed a new reachability-based measure for CakeML's verified cost semantics. The examples explored here suggest that the new

formulation is better suited for space safety proofs. We found that the need for extra assumptions and auxiliary lemmas has been greatly reduced and, as a consequence, proof scripts are more concise and easy to read, making the whole proving process more scalable. Overall, we hope that by making space safety reasoning easier, more ambitious verification projects that prevent out-of-memory errors can be undertaken.

Bibliography

[1] D. Aspinall, L. Beringer, M. Hofmann, H. Loidl, and A. Momigliano. A program logic for resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007. doi: 10.1016/j.tcs.2007.09.003. URL `https://doi.org/10.1016/j.tcs.2007.09.003`.

[2] 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.

[5] 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.

[6] E. Çiçek, D. Garg, and U. A. Acar. Refinement types for incremental computational complexity. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 2015. doi: 10.1007/978-3-662-46669-8_17. URL `https://doi.org/10.1007/978-3-662-46669-8_17`.

[7] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational cost analysis. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages,*

POPL 2017, Paris, France, January 18-20, 2017, pages 316–329. ACM, 2017. doi: 10.1145/3009837.3009858. URL https://doi.org/10.1145/3009837.3009858.

[8] A. Gómez-Londoño, J. Å. Pohjola, H. T. Syeda, M. O. Myreen, and Y. K. Tan. Do you have space for dessert? a verified space cost semantics for cakeml programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):204:1–204:29, 2020. doi: 10.1145/3428272. URL https://doi.org/10.1145/3428272.

[9] M. A. T. Handley, N. Vazou, and G. Hutton. Liquidate your assets: reasoning about resource usage in liquid haskell. *Proc. ACM Program. Lang.*, 4(POPL):24:1–24:27, 2020. doi: 10.1145/3371092. URL https://doi.org/10.1145/3371092.

[10] J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ML. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012. doi: 10.1007/978-3-642-31424-7_64. URL https://doi.org/10.1007/978-3-642-31424-7_64.

[11] 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.

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

[13] J.-M. Madiot and F. Pottier. A separation logic for heap space under garbage collection. *Proceedings of the ACM on Programming Languages*, 6 (POPL), Jan. 2022. URL http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf. to appear.

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

[15] 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.

[16] 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, 2017. doi: 10.1145/3133903. URL https://doi.org/10.1145/3133903.