THE UNIVERSITY OF CHICAGO

SELF-ADJUSTING MACHINES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

MATTHEW ARTHUR HAMMER

CHICAGO, ILLINOIS

DECEMBER 2012

For my parents, for my sister,

for my friends, and for Nikita

We are the Other,

all arising together,

Self adjusting Self.

Table of Contents

List of Figures

List of Tables

Acknowledgments

As a graduate student, I have been extremely fortunate to have been a member of various research institutions, and to have benefited from the wonderfully gifted people that embody these institutions. These institutions include the The University of Chicago, the Toyota Technological Institute at Chicago, Intel Corporation (specifically, the Programming Systems Lab, in Santa Clara), and the Max Planck Institute for Software Systems (in Kaiserslautern and Saarbrücken, Germany). The faculty, staff and fellow students at these institutions have given me support; over the course of many years, they have shared their advice, wisdom and experiences with me, for which I am grateful and eternally indebted. Through these experiences, I have had unique opportunities to grow as a student, as a researcher and as a human being. My trajectory continues to be shaped for the better by these relationships. This section of acknowledgments can never be sufficient for its intended purpose: To recognize those people from whom I have benefited, and for who I am thankful to know.

First, I thank my advisor, Umut Acar. Our relationship has been one of the most important in my life to date, and one for which I continually feel fortunate. Likewise, for their advice, feedback and support, I thank the rest of my thesis committee: John Reppy, David MacQueen and Rupak Majumdar. John, David and Rupak have each given me very thoughtful perspectives on this work, and have opened my mind to future possibilities.

Several fellow students, Ruy Ley-Wild, Georg Neis, Yan Chen, Roly Perera, Mike Rainey and Duru Turkoglu, have given me feedback on this research, or ideas related to it, and I have grown and benefited tremendously from our discussions together. In the context of this dissertation, I am particularly grateful to Georg Neis for his insights into the abstract machine formalism presented in Chapter 4 and Appendix C, and to Ruy Ley-Wild for his thoughtful guidance throughout our friendship; I have learned much from our collaborations together. Similarly, I have benefited tremendously from collaborations with Joshua

Dunfield, and discussions with Neelakantan Krishnaswami. At various points throughout the years, Derek Dreyer, Matthias Blume and Viktor Vafeiadis have supported me by providing space and time for open-ended discussions, where they always answered my questions thoughtfully. I am thankful to the other students with whom I have worked, and who have provided me with thoughtful feedback on the implementation presented in this dissertation. Some of these students include Pramod Bhatotia, Ezgi Cicek, Reinhard Munz and Mustafa Zengin. For their support and advice, I also thank Anne Rogers, Matthew Fluet, Anwar Ghuloum, Mohan Rajagopalan, Neal Glew, Leaf Petersen, Rose Hoberman, Paul Francis and Bob Harper.

Finally, I thank my parents, my sister and Nikita for their love and emotional support.

Abstract

In computer systems, the interaction of computations and data often causes incremental changes, not wholesale ones. Hence, there exists the possibility of improving the efficiency of a system by recording and reusing computations, rather than blindly remaking them anew. To this end, self-adjusting computation is a technique for systematically constructing computational structures that evolve efficiently and incrementally. Past work has explored several facets of this programming language-based approach, but no prior formulations have given a low-level account of self-adjusting computation. By low-level, we mean an account where machine resources are defined and managed explicitly, e.g., as in the C programming language.

We offer *self-adjusting machines*, a concept based on an operational interpretation of self-adjusting computation with explicit machine resources. By making their resources explicit, self-adjusting machines give an operational account of self-adjusting computation suitable for interoperation with low-level languages; via practical compilation and run-time techniques, these machines are programmable, sound and efficient.

Abstractly, we formally define self-adjusting machines that run an intermediate language; we prove that this abstract machine semantics is sound. Concretely, we give techniques based on this semantics that construct self-adjusting machines by compiling a C-like surface language into C target code that runs within an extensible, C-based run-time system. By creating new programming abstractions, library programmers can extend this C-based system with new self-adjusting behavior. We demonstrate that this extension framework is powerful by using it to express a variety of both new and old self-adjusting abstractions. We give an empirical evaluation showing that our approach is efficient and well-suited for programming space and time-efficient self-adjusting computations.

CHAPTER 1

INTRODUCTION

The world around us is continually changing. The information that our programs consume and produce changes with it. However, not all changes occur at once; rather, they occur gradually over time. Hence, there is a potential for our programs to avoid repeating unaffected work: By recording their past work, and by adjusting that record to account for incremental input changes, programs can avoid repeating redundant work in the future, potentially resulting in improvements that are asymptotically significant.

To exploit this potential, one can develop "dynamic" or "kinetic" algorithms that are designed to deal with particular forms of changing input by exploiting particular structural properties of the problem at hand (Chiang and Tamassia, 1992; Eppstein et al., 1999; Agarwal et al., 2002). This manual approach often yields updates that are asymptotically faster than full reevaluation, but carries inherent complexity and non-compositionality that makes the algorithms difficult to design, analyze, and use.

As an alternative to manual design of dynamic and kinetic algorithms, the programming languages community has developed techniques that either automate or mostly automate the process of translating an implementation of an algorithm for fixed input into a version for changing input. For a survey of this work, we refer the reader to Chapter 2. A recent approach for doing this kind of historical record-keeping and record adjustment, broadly called *self-adjusting computation* (Acar, 2005), builds on prior techniques from the programming languages community.

Specifically, self-adjusting computation generalizes the data dependence graphs of earlier language-based techniques (Section 2.1) by introducing *dynamic dependence graphs* (Acar et al., 2002). Unlike earlier dependence graphs, these graphs are generated from programs written in a mostly conventional way (e.g., using an existing Turing-complete language with general recursion).

Broadly, self-adjusting computation offers a systematic way of extending existing programming languages, as well as their programs, with an additional computational interpretation beyond normal execution. This additional interpretation is equivalent to normal, fixed-input execution from an *extensional* point of view, in terms of input and output, but is distinct from normal execution from an *intensional* view—that is, in terms of computational resources (Ley-Wild et al., 2008). Namely, by making an initial investment (in space), self-adjusting computations offer the potential of responding to incremental input changes significantly faster (in time). The investment of space is proportional to the fraction of the computation's running time that consists of interaction with changing data. In this sense, self-adjusting computation is a technique for amortizing the work associated with past computation across future incremental input changes; specifically, it trades computational space for time. Depending on the program, its environment, and the way that the environment changes, the improvement in update times may be up to a linear factor or better. Indeed, self-adjusting computation has been shown to be effective in a reasonably broad range of areas including computational geometry, invariant checking (Shankar and Bodik, 2007), motion simulation (Acar et al., 2008), and machine learning (Acar et al., 2007) and has even helped solve challenging open problems (Acar et al., 2010).

Being a programming language-based approach, self-adjusting computation relies on programmer help to identify the data that can change over time, called *changeable data*, and the dependencies between this data and program code. This changeable data is stored in special memory cells referred to as *modifiable references* (*modifiables* for short), so called because they can undergo incremental modification. The read and write dependencies of modifiables are recorded in a dynamic *execution trace* (or *trace*, for short), which effectively summarizes the dependency structure of self-adjusting computation. We refer to the implementation of this trace as the dynamic dependency graph (DDG). When modifiables change, the trace is automatically edited through a *change propagation* algorithm: some

portions of the trace are reevaluated (when the corresponding subcomputations are affected by a changed value), some portions are discarded (e.g., when reevaluation changes control paths) and some portions are reused (when subcomputations remain unaffected, i.e., when they remain consistent with the values of modifiables).

We say that a semantics for self-adjusting computation is *sound* (alternatively, *consistent*), if the change propagation mechanism always yields a result consistent with full reevaluation. In practice, the primitives for programming with modifiables can often be misused, in the absence of static checks baring this. In turn, the programmer's misuse of these primitives can break the soundness proven for the system, which makes assumptions of the program regarding the correct-usage of the self-adjusting primitives.

As we survey in Section 2.2, self-adjusting computation was first implemented in the form of libraries (and eventually a compiler) for high-level languages such as SML. These high-level languages act as both the host and implementation languages: Programmers write self-adjusting programs in same language that implements the self-adjusting primitives, including trace generation and change propagation. That these languages are high-level is not accidental: the dynamic dependence graph (execution trace) is a higher-order data structure, and as such, the implementation of self-adjusting computation uses the higher-order features of the high-level languages. In particular, since higher-order languages can natively represent closed functions, they are naturally suitable for implementing traces.

1.1 Problem statement

While in many ways ideal for expressing self-adjusting computation, high-level languages have a central drawback: By being high-level, they give up control over low-level details. In the context of self-adjusting computation, these details include the low-level representation and management of the trace. Indeed, empirical evaluations have shown that

memory management can be a significant overhead both in high-level languages generally (Tarditi and Diwan, 1994), and particularly for their self-adjusting computations (Acar et al., 2009). Moreover, this memory management overhead worsens as free memory becomes more scarce (Hammer and Acar, 2008).

However, there is a potential for a holistic combination of change propagation and memory management, by redesigning the techniques of self-adjusting computation at a lower level of machine abstraction, for a low-level programming language. What is lacking from previous self-adjusting computation techniques is an account of this lower level of abstraction—i.e., that of a low-level host and implementation language, where the programmer and the self-adjusting system cooperatively exert explicit control over machine resources. Our goal is to integrate self-adjusting primitives, originally designed in and for high-level languages, into a low-level, machine-oriented language.

Low-level languages are more machine-oriented languages. Specifically, by *low-level language*, we mean a language where:

- The type system describes memory layout
- The program text is a flat, static resource
- The call stack is a distinct resource from heap memory[1]
- Heap memory is managed explicitly

We choose the C language (Kernighan and Ritchie, 1988; Norrish, 1998), the de facto portable assembly language, as our canonical low-level language. Since previous approaches for self-adjusting computation rely on the abstractions of high-level languages (viz. higher-order functions and garbage collection), previous approaches for self-adjusting

1. For instance, programmers in low-level settings such as C often avoid recursion in favor of loops to preserve stack space. Such care avoids over-running whatever stack is preallocated for their process and avoids the details of growing this stack space dynamically.

computation are not directly applicable when C is the host language. Many details—such as how the dynamic dependency graph is created, represented and managed—must be resolved before applying the previous high-level approaches. Moreover, past approaches change the stack behavior of the program, changing tail-recursive fixed-input algorithms into non-tail recursive self-adjusting ones. In C, the call stack is a distinct resource whose use is often carefully managed.

1.2 Challenges

Our goal is to integrate self-adjusting primitives into a low-level language. When attempting to apply known techniques in this new setting, issues arise due to their reliance on the abstractions provided by high-level languages. We survey four interrelated challenges:

Pervasive mutation. The imperative semantics of high-level self-adjusting systems assume that only modifiable references are mutable: All other data is immutable (i.e., pure). Moreover, their type systems distinguish between mutable and immutable data. Beyond guaranteeing a notion of soundness, these type systems either under-pin a higher-order library interface for the programmer, or they guide a compiler as it transforms the program with specially-typed primitives to use the associated run-time library.

By contrast, in a low-level setting, mutation is pervasive. Moreover, all data (both stack and heap-allocated) is implicitly mutable, and the type system only helps to describe the layout of this memory, not what operations on this memory are permissible or illegal. In this context, we must find a programmable, sound and efficient way for the programmer and compiler to distinguish modifiable from non-modifiable data.

Trace representation. Past high-level approaches have a high-level account of traces, which are higher-order data structures. In these languages traces are readily created and

composed via function abstraction and function application; together with garbage collection, these high-level features abstract over the low-level details of closed functions, such as their representation and dynamic management.

By contrast, in low-level languages functions are merely names for statically-fixed, globally-scoped blocks of code. For programmability, we must find a way to replace the necessary higher-order language features with appropriate compiler technology. Moreover, for efficiency, we must mitigate the cost of tracing the comparatively large amount of mutation found in low-level languages, lest our traces become too large.

Stack management. Past approaches for self-adjusting computation tacitly assume that the stack is inexhaustible, and that the stack profile of a program need not be preserved by execution tracing or change propagation. This stack management task is complicated by the *operational knot* of self-adjusting computation, a term we introduce to refer to the recursive control flow induced by the internal techniques of self-adjusting computation.

Unlike high-level languages, low-level languages allocate the call stack as an efficient and distinct run-time resource. Hence, it is desirable that the high-level approaches of traced execution and change propagation be adapted so that they are properly tail-recursive.

Heap management. Finally, past approaches for self-adjusting computation tacitly assume that all garbage is collected automatically. This includes garbage generated by the self-adjusting program, as well as internal garbage that results from updating the trace via change propagation. Traditional garbage collection approaches are badly-suited to the recursive, imperative structure of the execution trace and its complex mutator (viz. the change propagation algorithm). This memory management task is complicated by the *spatial knot* of self-adjusting computation, a term that we introduce to refer to the complex structure of the execution trace.

In contrast to high-level settings, automatic collection is lacking in low-level languages; instead, the heap is managed explicitly. To effectively interoperate with low-level code, we require a simple protocol whereby self-adjusting computations and their execution environments can manage memory cooperatively.

To address the challenges above, we want a sound, programmable and efficient solution. For soundness, we must provide an underlying self-adjusting model that is consistent with full reevaluation. For programmability, we must not overly burden the programmer with heavy annotations, manual transformations or complex programming models. For efficiency, we should exploit known analyses and techniques that are applicable for low-level settings.

While adopting a low level of abstraction usually means being more detailed and explicit, by virtue of this lower level, we gain more control over a machine's resources. With this control, there is an opportunity for a self-adjusting system to incorporate the management of its resources within its incremental rules of self-adjustment. For instance, garbage collection can itself be seen as another incremental problem, arising at the level of the system's implementation, rather than at the level of its applications[2]. More generally, in a low-level setting there is the unique possibility for the trace mechanics, as well as explicit management of other dynamic resources, to be folded into the operational rules of self-adjustment.

We address the challenges posed by a low-level setting by using specialized compilation and run-time techniques, based on a new low-level machine model, called self-adjusting machines, which we introduce below. Our static compilation techniques transform the program to extract information from and perform transformations on the program that

2. For instance, if we define memory liveness generically using pointer reachability, then garbage collection can be seen as an incrementalized graph-reachability problem: As memory incrementally mutates, the problem is to incrementally find blocks that become unreachable.

would be overly burdensome to require of the programmer. This includes providing simple primitives that, through compilation, perform trace construction and management of the program's execution. Our dynamic run-time techniques incorporate resource management, including garbage collection, into the rules of self-adjustment.

1.3 Thesis and contributions

We give an account of self-adjusting computation at a low level of abstraction through the theory and practice of *self-adjusting machines*. We study these machines both abstractly, as a formalism, and concretely, as a closely-related implementation.

Thesis statement

> By making their resources explicit, self-adjusting machines give an operational account of self-adjusting computation suitable for interoperation with low-level languages; via practical compilation and run-time techniques, these machines are programmable, sound and efficient.

Below, we sketch our main contributions, with the intention of showing how these contributions fit together conceptually. In Section 1.4, we give a more detailed account of our specific technical concerns, and a more detailed overview of our corresponding technical results.

Surface language based on C**.** We give a C-like surface language, called CEAL, for writing self-adjusting programs at a low level of abstraction. The CEAL language augments (a large subset of) C with minimal primitives for self-adjusting computation. We provide a

front-end that translates CEAL input into IL, which has structure well-suited for compilation (analysis and transformation) but not well-suited for human readability. For practical purposes, our surface language CEAL interoperates with C: under certain restrictions, CEAL can call and be called by other C code.

Abstract machines. We give an abstract machine semantics for self-adjusting computation for a low-level intermediate language IL. This includes accounts of how change propagation interacts with a control stack, with return values and with memory management. We prove that this semantics is sound. We use this semantics to guide the design of an associated compiler and run-time system.

Compiler and runtime system. Via compilation and run-time techniques, we realize our abstract machine model with a concrete design and implementation based in C.

We design and implement a compiler and runtime system for IL, the intermediate language used by our abstract machines. The compiler emits as a target C, the language of the run-time system and its extensions. Guided by the self-adjusting machine model, our compiler combines standard and new static analyses to transform IL programs into optimized C-based target code. The transformations make IL, and consequently CEAL, more easily programmable. The analyses inform the transformations by relating the control- and data-flow of the program to its trace structure. By extracting this static information, we specialize the transformations and optimize the generated target code.

Our C-based run-time system implements the automatic memory management design of the abstract machine. Furthermore, the system is extensible: it allows library programmers to create new programming abstractions, which the compiler and run-time system incorporate into the self-adjusting programs that they compile and run.

Empirical evaluation. We perform an empirical evaluation of our implementation, comparing language features and optimizations. In many cases our optimizations reduce the overhead of the naive compilation and run-time approach by an order of magnitude in both time and space.

We also compare our approach with competing approaches in other languages. Overall, our implementation is very competitive, but especially so when memory is scarce. In some cases we significantly outperform competing approaches, where we reduce resource usage by an order of magnitude in time, and a 75% reduction in space (Hammer and Acar, 2008; Hammer et al., 2009).

1.4 Technical results

Under the constraint of preserving extensional semantics (a property we refer to variously as *consistency* or *soundness*), the study of self-adjusting computation is concerned with giving a more efficient *intensional semantics* to programs whose input changes incrementally over time. Hence, an *intensional view* of computation (e.g., a cost model) is needed to justify that techniques offered by self-adjusting computation have efficacy. This dissertation gives both abstract and concrete machines for self-adjusting computation. These machines share a cost model that we define in Section 4.6; in this model, the cost of updating a computation is proportional to the sum of new and old work that represents the computed difference between the new and old traces. This cost model is consistent with that of earlier work based on adapting Standard ML (Ley-Wild et al., 2008, 2009; Ley-Wild, 2010).

1.4.1 Contexts of concern

Our high-level domain of interest is in using the approach taken by self-adjusting computation to reduce the cost of adjusting a computation to incremental change in its execution environment. Within this domain, our technical results share a common overarching context of concern: the context where low level models and implementations of self-adjusting computations are desired. For instance, previous intensional models of self-adjusting computation do not consider the costs associated with managing dynamic memory; the corresponding implementations use high-level languages with automatic garbage collection (e.g., Standard ML). While such models and implementations correspond intensionally (in terms of a cost model that relates them), neither the abstract models nor the implementations are concerned with saying how to manage memory, or how its associated costs are quantified. By contrast, in a low level model of computation, the low level nature of the memory model gives programs control over and responsibility for being explicit about how dynamic memory in the heap is allocated and reclaimed, if at all.

Within this context of concern (a low level machine model), our technical results include a number of designs, techniques and proofs, whose specific contexts of concern vary. To organize them, we consider how our techniques for self-adjusting computation fall address three (interrelated sub-)contexts of concern:

- when one desires efficient *stack-based management* of control flow,
- when one desires efficient *memory management* of trace and heap resources, and
- when one desires *programming abstractions* whose uses are not burdensome.

These contexts are not mutually-exclusive: some of our results are applicable in two or several contexts. Below, we survey our abstract machine design and an associated compiler and run-time system design; these designs are relevant for all the concerns above. We

discuss each of the contexts of concern individually in greater detail, highlighting their associated technical results within this dissertation.

1.4.2 Abstract machine design

Our abstract machine design is central to our entire approach, and is relevant for all the concerns that we consider (viz., for stack-based execution, for efficient trace and heap management, and for a simple programming model). We use the abstract machine design to guide and unify our techniques, and to establish their soundness from a formal (i.e., abstract) view.

From a high-level, our methodology for establishing soundness is as follows. We give two abstract machine models: the *reference machine* serves as a conventional semantics that we use to model ordinary execution; it lacks the features of self-adjusting computation. The self-adjusting machine semantics is modeled formally by an extension of the reference machine that we call the *tracing machine*, since it features an execution trace, as well as transitions that model change propagation of the trace. We characterize the soundness (or *consistency*) of our approach by showing that these machines are related.

In particular, we show that under certain conditions, every run of the self-adjusting machine is consistent with a corresponding run of the reference machine, which lacks self-adjusting features (Theorem 4.4.3 in Section 4.4). Separately, we show a simple compilation technique, *destination-passing-style (DPS) conversion*, that achieves the program property required by our soundness result (Theorem 4.5.1 in Section 4.5). Later, in the context of a compilation framework, we show how to refine this program transformation using further static analysis; the refined approach works to lower the (constant-factor) overhead of the program transformation (Section 5.8).

***Abstract* versus *concrete* machines.** Throughout this dissertation, we use the term *self-adjusting machine* to refer both to the tracing machine formalism, as well as its concrete realization via our compiler and run-time library techniques. We organize this dissertation so that the context of the term clarifies how to interpret it; when ambiguity is possible, we use the terms *abstract machine* and *concrete machine* to respectively disambiguate between the self-adjusting machine formalism and its concrete realization achieved via our compiler and run-time system.

1.4.3 Compiler and run-time system design

We show how to realize concrete, executable self-adjusting machines via compilation and run-time techniques. Our abstract machine model guides and corresponds with these compilation and run-time techniques. Like the abstract machine design, our compilation and run-time techniques are relevant for all the concerns that we consider.

Compilation techniques. Our compilation techniques prepare programs written in a C-like surface language for inter-operation with our run-time system. Our compiler performs two major program transformations: *DPS-conversion* and *normalization*. These transforms automate otherwise tedious program transformations, and are each designed within the context of the concerns listed above. Regarding stack-based execution, we show that our compiler's program transformations preserve the stack profile of the transformed program (i.e., that our compiler preserves the stack space required by the program, up to a constant factor). Regarding dynamic memory management, our the compiler statically places (and optimizes) allocations performed at run-time; hence, it plays a significant role in our approach for efficient heap and trace management. Finally, since it automates otherwise tedious, manual transformations of program text, the compiler greatly simplifies the programming model for our surface language.

Run-time techniques. Our run-time system complements target programs of our compiler; together, they realize the transition semantics of our abstract machine. Our run-time techniques are motivated by all the concerns listed above. Regarding stack-based execution, our run-time library techniques are properly tail-recursive, guaranteeing that they do not consume more than a constant-factor of additional stack space. Regarding dynamic memory management, the run-time library efficiently realizes the heap and trace memory management of the abstract model. In particular, our run-time technique automatically reclaims garbage in amortized constant-time per allocated block. Finally, the run-time system is designed with usability in mind: It implements the simple, orthogonal primitives of the abstract model, as well as a general framework for extending this model with new features. In particular, domain-specific library authors can readily supplement these core primitives with new programming abstractions that interface internally to the run-time system trace and its change propagation algorithm.

1.4.4 Efficient stack-based execution

When one desires efficient stack-based management of control flow, one writes programs consciously with respect to a stack resource that saves and restores local state and local control flow (e.g., across interprocedural boundaries). The primary concern here is the program's complexity with respect to this stack resource. Within this context of concern, features of our surface language, program transformations and run-time techniques are relevant:

- Our surface language gives simple primitives for specifying stack operations that are orthogonal to the syntax for function calls, and which are preserved by our compilation and run-time techniques. That they are orthogonal to function calls and not eliminated or reduced during compilation is crucial: These stack operations encode

programmer intentions and associated intensional properties that are relevant during self-adjustment (Section 3.4).

- We show that our compiler's program transformations preserve the stack requirements of the program, up to a constant factor in required space. Specifically, we first show that DPS-conversion preserves the stack behavior of the program precisely, i.e., in terms of the occurrences of pushes and pops (Section 4.6). Next, we show that normalization also preserves the stack behavior of the program (Section 5.7). Both transformations introduce only a constant-factor overhead in required stack space.

- We give a run-time system's interface and implementation that is properly tail-recursive: Its own usage of the (concrete) machine stack adds only a constant factor of space to that of the compiled program with which it inter-operates (Section 6.5.6).

Beyond low level settings. Stack-based control flow is relevant outside the setting of low level languages and low level machine models. Prior to this work, however, the stack profile of the program (as an intensional property) was not considered in the context of self-adjusting computation. In fact, through library-based instrumentation or through special compilation techniques, all previous approaches for self-adjusting computation alter the direct-style, stack-based mechanics of the program's underlying algorithm. As a direct consequence, these approaches generally affect a program's complexity with respect to the control stack. Hence, the results above give an alternative to previous approaches for self-adjusting computation whenever there is concern for stack complexity. Though we study C specifically, our results in this context are broadly applicable, and are not specific just to low level settings. For instance, they can be readily applied in other stack-based languages, both high and low level (C++, Java, OCaml, etc.). The only requirement we demand is that the underlying machine model properly support function calls in tail-position (i.e., as jumps that do not return).

1.4.5 Efficient memory management

When one desires efficient management of dynamic memory, one writes programs consciously with respect to a heap resource. Within the context of self-adjusting computation, this heap resource is used to store two interrelated structures: the trace structure of the self-adjusting computation, and the user-defined structures created and manipulated by this computation. The management of the heap resource is efficient if the complexity cost associated with reclaiming a previously-allocated block (of either the trace or a user-defined structure) is low. While automatic management techniques (i.e., general-purpose garbage collectors) can often be very efficient, they do not generally provide a complexity guarantee that is independent of available free space. When one wants a stronger guarantee, one uses a specialized approach to managing memory, perhaps resorting to explicit management (i.e., where the program itself says how and when to reclaim blocks).

Within this context of concern, features of our surface language, compilation and runtime techniques are relevant:

- Our surface language provides a single primitive for allocating memory; depending on the context of this allocation, reclamation is either automatic (in the context of self-adjusting code), or manual (in the context of a non-self-adjusting code) (Section 3.2).

- Our compiler performs optimizations that coalesce heap resources, combining trace and user-defined allocations. In doing so, it amortizes the overhead of tracing in terms of both space and time, and simplifies the layout of generated traces by flattening them into larger blocks in the heap (Section 5.8).

- Using our abstract machine model, we describe the automatic reclamation of garbage during change propagation. We formally establish in this abstract model that this technique is sound (Section 4.4). Guided by the technique of the abstract machine,

our run-time system performs an garbage collection analogously within the context of concrete machines. Both abstractly and concretely, we show that the cost of reclaiming an allocated block during garbage collection adds only a constant-factor overhead to the cost of performing change propagation (Sections 4.3 and 6.5.6).

Beyond low level settings. Our automatic memory management techniques are applicable in any setting where the programming model of self-adjusting computation makes reclamation implicit (as in our surface language), while the implementation has the freedom to make reclamation explicit (as in a low level language, such as C). We simplify our model and implementation by insisting that non-self-adjusting code manage its allocated memory explicitly, and that this explicit management is coordinated with the self-adjusting computation with respect to certain restrictions (Section 6.2). Extending our approach to implicitly reclaim non-self-adjusting allocations remains an open problem.

1.4.6 Simple programming model

When one desires a usable set of programming annotations, one desires that the required annotations rely on abstractions that are easy to reason about; moreover, one desires that the annotations themselves are easy to insert, easy to move, and easy to delete or replace. Within the context of low level self-adjusting computation, we provide annotations within a C-like surface language that meet these criteria:

- Our surface language provides annotations whose semantics is extensionally meaningless: They do not alter the input-output behavior of the annotated program. As such, they can be inserted gradually, as the program is developed and maintained. While abstract, these annotations still give the human programmer control over the intensional behavior of the underlying (concrete) self-adjusting machine.

- These annotations correspond to operations over the abstract resources of our (abstract) self-adjusting machines Chapter 4; hence, the programmer can use this model to reason abstractly about their usage.

- These annotations correspond to operations over the concrete resources of our (concrete) self-adjusting machines Chapters 5 and 6; hence, the compiler uses these annotations to guide its transformation of the surface program, through the intermediate language shared with the abstract model, and into the concrete machine implemented by the run-time system.

1.5 Chapter outline

The outline below consists of a digest of summaries that we collect from each chapter.

Chapter 1 introduces the topic of this dissertation.

Section 1.6 follows this outline and provides additional background on the problem statement and challenges given above (Sections 1.1 and 1.2). We describe the general problem of information maintenance in the presence of dynamic change, and self-adjusting computation as a language-based technique for constructing solutions (Section 1.6.1). We introduce the tools and internal techniques that comprise a self-adjusting language system (Section 1.6.2). We describe how these techniques are inherently higher-order, and how the high-level languages used in previous work have performance shortcomings, especially when traces are large and memory resources are scarce (Section 1.6.3). We describe our use of the terms *high-* and *low-level language* in more detail, focusing on the extra control (and associated burdens) that low-level languages offer the programmer (Section 1.6.4).

Chapter 2 surveys related work and contrasts it with our own.

In this chapter, we focus on the work most closely related to self-adjusting computation generally, and to self-adjusting machines, specifically. First, we characterize a selection of the most closely related early work from the 1980's, the 1990's and earlier (Section 2.1). Next, we characterize the development of self-adjusting computation (Section 2.2). Finally we contrast and compare to other contemporary work on incremental computation (Section 2.3) and garbage collection (Section 2.4).

Chapter 3 describes CEAL, our surface language.

We describe a C-based surface language called CEAL for writing low-level self-adjusting programs. We introduce the programming model used by CEAL (Section 3.1), which is based on the notion of *levels*. We describe the programming abstractions modifiable memory (Section 3.2), and the outer and inner levels of a CEAL program (Sections 3.3 and 3.4). We describe how self-adjusting machine resources interact in the CEAL programming model (Section 3.5). We illustrate the programming model with two simple examples (Sections 3.6 and 3.7). Finally, we describe certain practical concerns (Sections 3.8 and 3.9).

Chapter 4 describes IL and its associated abstract machine semantics.

We present an abstract self-adjusting machine model that runs programs written in an intermediate language for self-adjusting computation that we simply call IL. We give an informal introduction to the design of IL (Section 4.1), and using with examples from Chapter 3, we illustrate example IL programs (Section 4.2). We give the formal syntax of IL, to which we give two semantics, by defining two abstract machines: The *reference machine* models conventional evaluation semantics, while the *tracing machine* models the trace structure and self-adjusting semantics of a self-adjusting machine (Section 4.3).

Our low-level setting is reflected by the abstract machines' configurations: each consists of a store, a stack, an environment and a program. Additionally, the tracing machine's configurations include a trace component. We show that automatic memory management is a natural aspect of automatic change propagation by defining a notion of garbage collection. Under certain conditions, we show that this self-adjusting semantics is (extensionally) consistent (Section 4.4). We give a simple transformation that achieves these necessary conditions (Section 4.5). Finally, we give a cost model framework to relate the intensional semantics of self-adjusting machines to that of the reference semantics (Section 4.6).

Chapter 5 describes the design of an optimizing compiler for IL targeting C.

First, we give an overview of the role of compilation as a bridge from the abstract machine (Chapter 4) to the run-time system (Chapter 6) and outline the reminder of the chapter in greater detail (Section 5.1). We describe how a C-based self-adjusting program consists of multiple levels of control, which our compiler automatically separates (Section 5.2). We give a C-based run-time library interface for constructing execution traces (Section 5.3). We represent programs during compilation using IL and CL (Section 5.4). CL is a representation closely-related to IL that augments programs with additional static control structure. We use static analyses to inform both our basic compilation and our optimizations (Section 5.5). We give a basic translation (from IL to CL) for basic compilation (Section 5.6). We give a graph algorithm for normalization, which statically relocates certain CL blocks as top-level functions (Section 5.7). To improve upon this compilation approach, we describe several optimizations (Section 5.8). Finally, we give further notes and discussion (Section 5.9).

Chapter 6 describes the design of an extensible C-based run-time system.

First, we present an overview of the run-time system's role in the design of the larger system, and in particular, its role vis-à-vis the compiler; we also review the challenges inherent to low-level memory and stack management (Section 6.1). Within the context of memory management, we give a detailed discussion of our run-time system's interaction in memory as implemented with an abstraction we call the *trace arena* (Section 6.2). Based on certain assumptions and restrictions, we describe the basic data structures and memory management techniques for an efficient run-time implementation of self-adjusting machines in C (Section 6.3). With detailed code listings, we describe the design and implementation of the two central abstractions in our run-time system design: trace nodes and self-adjusting machines (Sections 6.4 and 6.5). In doing so, we pay careful attention to how memory and stack resources are created and destroyed, within the context of subtle internal invariants and state changes.

We study the performance of our current and past implementations. We evaluate our current compiler for self-adjusting computation, based on the theory and practice of self-adjusting machines. We empirically evaluate our current system by considering a number of benchmarks written in the current version of CEAL, and compiled with our current compiler. Our experiments are very encouraging, showing that our latest approach still yields asymptotic speedups, resulting in orders of magnitude speedups in practice; it does this while incurring only moderate overhead (in space and time) when not reusing past computations. We evaluate our compiler and run-time optimizations (Section 5.8), showing that they improve performance of both from-scratch evaluation as well as of change propagation. Comparisons with previous work, including our own and the DeltaML language shows that our approach performs competitively. Finally, we briefly survey our current and past implementations Section 7.3.

Chapter 8 concludes. We describe future directions and open problems.

1.6 Background

This section provides additional background on the problem statement (Section 1.1) and challenges (Section 1.2) given above. We describe the general problem of information maintenance in the presence of dynamic change, and self-adjusting computation as a language-based technique for constructing solutions (Section 1.6.1). We introduce the tools and internal techniques that comprise a self-adjusting language system (Section 1.6.2). We describe how these techniques are inherently higher-order, and how the high-level languages used in previous work have performance shortcomings, especially when traces are large and memory resources are scarce (Section 1.6.3). We describe our use of the terms *high-* and *low-level language* in more detail, focusing on the extra control (and associated burdens) that low-level languages offer the programmer (Section 1.6.4).

1.6.1 Self-adjusting structures

Self-adjusting computation uses an amortized analysis[3], as do other self-adjusting structures, such as certain binary search trees. In his 1986 Turing Award lecture, Robert Tarjan explains the organic, naturalistic design principle underlying the concept of self-adjusting data structures (Tarjan, 1987):

3. Self-adjusting computation uses an amortized analysis in at least two ways: First, it relies on data-structures that themselves use an amortized analysis, such as the structure used to maintain the total order of the execution trace (Dietz and Sleator, 1987; Bender et al., 2002); hence, the complexity of an implementation based on those structures, such as the one in this dissertation, can be characterized best in an amortized sense as well, i.e., across sequences of interleaved input changes, change propagation cycles and output observations. Second, in a more informal sense, self-adjusting computation amortizes the (space and time) work of tracing over the savings gained by reusing and adapting the saved results in new execution contexts with changed inputs.

> In designing a data structure to reduce amortized running time, we are led to a different approach than if we were trying to reduce worst-case running time. Instead of carefully crafting a structure with exactly the right properties to guarantee a good worst-case time bound, we can try to design simple, local restructuring operations that improve the state of the structure if they are applied repeatedly. This approach can produce "self-adjusting" or "self-organizing" data structures that adapt to fit their usage and have an amortized efficiency close to optimum among a broad class of competing structures.

To illustrate how self-adjusting structures are related to efficiently maintaining incrementally changing information, we use an analogy between self-adjusting trees and self-adjusting computation. Through this analogy, we highlight the inherent challenges addressed by self-adjusting computation.

Self-adjusting trees. Suppose that a programmer wishes to maintain a sorted view of a collection of data items, whose members change over time. After the collection has changed and one needs the new sorted view, one can fully re-sort the entire collection. In this way, one only need to be concerned with sorting a complete collection, and not with the problem of updating a prior sort to reflect a different set of initial conditions (i.e., the changed data collection). However, if the collection only ever changes gradually, it's more efficient in an amortized sense to do something other than a full resorting[4]. What is needed is to record and repair a dynamic structure that encodes the items' comparisons, so

4. In the *amortized* analysis of an algorithm or data structure, we characterize efficiency in terms of operation sequences, as opposed to the worst-case performance of individual operations. Likewise, in the context of self-adjusting computation, we focus on the aggregate performance of updating a computation across a sequence of input updates and output observations over time; during this time span, we use overall time savings to justify the from-scratch (space and time) overhead of performing fresh computation (*from-scratch computation* is that which is not being reused, but may be stored to be reused and/or adapted in future execution contexts).

that pairs of items that aren't changing need not be re-compared each time the collection changes.

Using any one of the many varieties of standard, textbook algorithms for self-adjusting binary search trees, e.g., *splay trees* (Sleator and Tarjan, 1985; Bender et al., 2002), one can accomplish this maintenance task optimally. That is to say, one first makes an asymptotically linear investment of space, as the self-adjusting tree requires $O(1)$ space for each data item being sorted. In return, this space "remembers" past comparisons between the collection's keys, saving time when and if the collection changes. In particular, as data items are added or removed from the set, the self-adjusting tree allows one to maintain a sorted view in $O(\log n)$ time per update (amortized across the sequence of updates), by injecting these collection updates (e.g., a sequence of insertions to and removals from the collection) into the self-adjusting trees abstract interface (e.g., a sequence of insertions into and removals from the tree).

Self-adjusting trees and self-adjusting computations serve as an illustrative pairing for comparison and contrast. Table 1.1 shows how the two concepts parallel each other.

Table 1.1: Self-adjusting trees and computations: Parallel Concepts

Parallel concept	**Self-adjusting trees**	**Self-adjusting computation**
Problem specification	An ordering	A program
Problem instance	A collection of keys	A program input
Generic maintenance rules	Splaying algorithm	Change-propagation algorithm[†]
Spatial record	Self-adjusting tree	Self-adjusting program trace[†]
Input changes	Key insertion or deletion	General input mutation[‡]
Invariant maintained	Sorted keys	Output consistent with input

[†] We introduce these techniques in greater detail in Section 1.6.2.
[‡] Depending on the system, different restrictions apply to this mutation; see Section 2.2 for details.

Both structures are parameterized by both a problem specification and a problem instance.

In this sense, they are both parametric structures. They both employ generic maintenance rules, which are independent of both the problem specification, and the problem instance. The underlying techniques of both use a spatial record to save time when re-establishing the maintained invariant. In both cases, the techniques maintain an invariant that reflects a high-level property of the problem instance.

While parallel in many ways, the two concepts are also quite different: a self-adjusting tree is parameterized by only an ordering with certain simple mathematical properties (reflexivity, transitivity and antisymmetry), while a self-adjusting computation is parameterized by a program, whose complexity is of an entirely richer nature, since it can describe anything that is computable. Table 1.2 summarizes how, out of this gap, important contrasts arise. In particular, the known lower bounds and simple information interaction used for sorting become complex and unknown for general computational patterns.

Table 1.2: Self-adjusting trees and computations: Contrasting Concepts

Contrasting concept	**Self-adjusting trees**	**Self-adjusting computations**
Invariant maintained	Sorted keys	Output consistent with input
Dynamic dependence structure	*Simple*: Binary tree of pairwise comparisons	*Complex*: Any computable pattern is possible
Lower bounds	Established and well-known	Unknown / input-sensitive

If we consider information-maintenance problems that are more complex than the sorting problem from above, the question arises: How do we best systematize the specification and implementation of these problems' solutions?

Designing, analyzing, and implementing dynamic (or kinetic) algorithms can be complex, even for problems that are relatively simple in the conventional setting. As an exam-

ple, consider the problem of planar convex hulls, whose conventional version is straightforward. The incremental version, which allows discrete changes in the set of input points[5], has been studied for over two decades (Overmars and van Leeuwen, 1981; Brodal and Jacob, 2002). In particular, Jacob's dissertation focuses solely on this topic (Jacob, 2002), describing and analyzing an efficient (albeit considerably complex) algorithmic approach over the course of one hundred pages. By contrast, self-adjusting computation can be used to give a practical implementation based on the programmer supplying little more than the conventional, fixed-input algorithm (Acar et al., 2006).

Self-adjusting computation. Self-adjusting computation is a language-based approach for solving dynamic information-maintenance problems. What these solutions have in common is that they are all based around the approach of extracting the desired behavior from a program that also (and primarily) describes non-incremental input-output behavior.

The qualities that make self-adjusting computation appealing follow from it being programming language-based: since programmers already know how to program for the non-incremental case, we can see self-adjusting computation as just another extension of our programs' original meaning:

- two programs' composition leads to their self-adjusting computations' composition.
- a program's correctness leads to its self-adjusting computations' correctness.
- a program's optimization (for time and space) leads to its self-adjusting computations' optimization (for space and time, perhaps in different proportions).

5. Despite the term *incremental*, we mean to consider both insertions and deletions from the set of input points. In this dissertation, we do not differentiate between problems whose input is *incremental* (allowing insertions) versus *decremental* (allowing deletions).

Listing these parallels is not to say that self-adjusting efficiency always comes "for free" from ordinary programming principles, but that in many cases, common patterns for efficiently handling dynamic changes can be swept under programming abstractions that resemble conventional programming (higher-order functions and mutable reference cells).

However, to realize more efficiency improvements, some incremental patterns may require specialized abstractions of their own. For this purpose, self-adjusting computation offers *extensibility*, again drawing on the benefits of being language-based. Just as programming languages can encapsulate domain-specific knowledge into libraries of reusable abstractions, self-adjusting computations can also be extended in a similar fashion. First, the library extension programmer uses special knowledge of the domain in question to define the efficient incremental behavior that is internal to their extension. Then, once written, these extensions are exposed to the programmer as new abstract data types (Acar et al., 2010).

1.6.2 Tools and techniques

From the client programmer's viewpoint, a *self-adjusting language system* (such as the one presented in this thesis) consists of:

1. a set of primitive programming abstractions for interacting with modifiable data,
2. a compilation strategy for compiling programs containing these primitives,
3. an efficient run-time implementation of these primitives, and
4. a framework for extending these primitives with new abstractions.

In recent years, the basic primitives have become better understood and easier to use. Despite the differences in the exact facade presented to the programmer, all of these variations on the programming abstraction share a set of conceptual themes and internal techniques, which we introduce below. We give a more detailed survey of the development of self-adjusting computation in Chapter 2.

Abstractly, the self-adjusting programming primitives provide some family of queryable, modifiable structures for storing and representing program state. As the input undergoes incremental change (e.g., caused by external, environmental changes), the self-adjusting techniques work in concert to maintain the program's input-output relationship across this state. To accomplish this efficiently, self-adjusting computation combines three techniques:

Traced execution to maintain a record of dynamic dependencies among modifiable data.

Change propagation is a a scheduling algorithm, designed in conjunction with the trace data structure. It edits the trace, replacing inconsistent portions with consistent ones.

Memoization caches and reuses portions of execution traces, to avoid reexecution.

Given an initial environment, the program executes and generates an initial execution trace. Internally, this trace is represented as a *dynamic dependency graph* (DDG) that relates incrementally modifiable data to the execution steps that depend on it. The DDG is a higher-order data structure: each traced step stores a suspended computation that can be re-executed to adjust the execution to newly-modified data. Managing this reexecution is the purpose of the change propagation algorithm.

Change propagation. The change propagation algorithm removes inconsistent parts of the trace and installs new, consistent replacements. It is a worklist algorithm: Using the DDG trace data structure, change propagation consists of scheduling inconsistent intervals of the trace for reexecution. Each interval begins with a traced action that is no longer

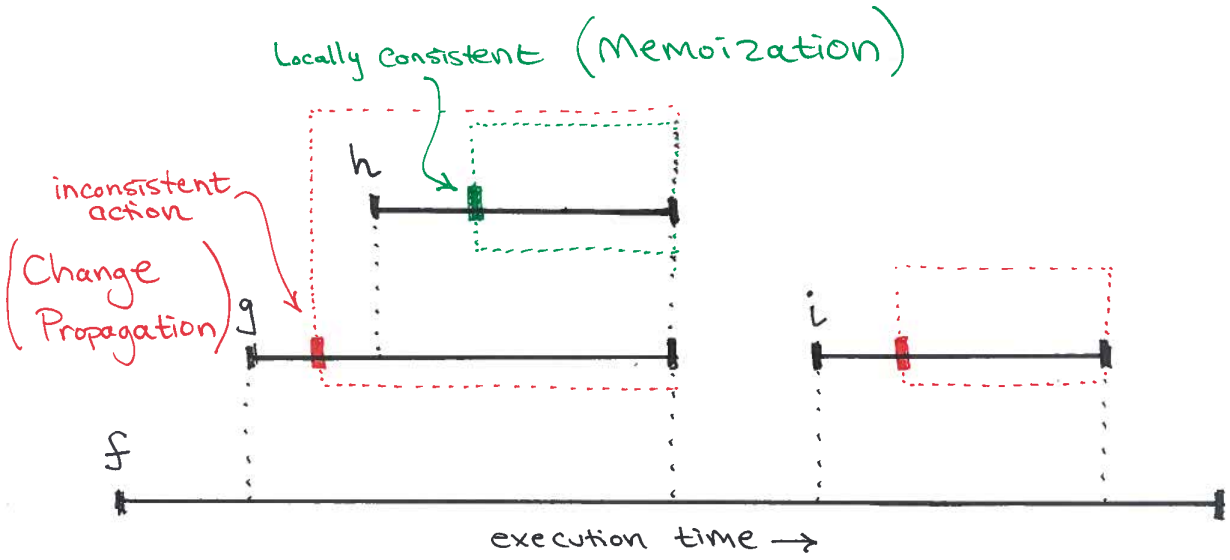

Figure 1.1: The operational knot of self-adjusting computation.

consistent with the current execution environment. Change propagation reexecutes (the suspended versions of) these actions monotonically with respect to their original execution order. In doing so, it replaces the affected portions of the trace, making them consistent.

The operational knot. In a self-adjusting system, the change propagation algorithm, memoization and traced execution are coroutines, operationally. That is, these techniques are knotted in mutual reference: change propagation consists of intervals of traced reexecution, which, due to memoization, may each consist of intervals of change propagation. Hence, from an operational perspective, there is a recursive dependency between the algorithmic decisions of change propagation and the reexecution of inconsistent portions of the program under consideration.

Figure 1.1 illustrates an example of how this operational knot arises. The figure shows an execution trace with nested intervals that correspond to nested function invocations (where function f calls g, g calls h and afterward function f calls i). The trace interval for g contains an inconsistency. Since it occurs before (to the left of) the other inconsistency,

change propagation schedules its reexecution first. During its reexecution, the program generates a new (consistent) interval for g to replace the old (inconsistent) one.

Within this new execution, g again calls h; in doing so, say that it reaches a state that is locally-consistent with a state in the old trace. At this point, reexecution uses memoization to locate and reuse this nested interval from the old trace. Though not shown in the figure, this interval may itself contain internal inconsistencies that demand repair from change propagation. Hence, the operations of change propagation, reexecution and memoization are nested and recursively interdependent.

Generally speaking, this operational knot makes it difficult to tell what reexecution is required, since each update can create a cascading flow of inconsistencies that ripple through the entire execution trace. However, for certain categories of programs, input data and input changes, the techniques offered by self-adjusting computation lead to optimal incremental update times.

Higher-order language features. As we survey in Chapter 2, self-adjusting computation primarily arose in the form of libraries (and eventually a compiler) for high-level host languages such as SML. This is not accidental: self-adjusting computation critically relies on higher-order features of high-level languages. In particular, since higher-order languages can natively represent closed functions, they are naturally suitable for implementing traces.

Function closures, or simply *closures*, consist of a record containing a function and values for some subset of its free variables. When these functions are completely closed (all free variables have an associated value recorded in the closure), they are commonly referred to as *thunks* (Ingerman, 1961).

Thunks are ubiquitous in functional programming. In lazy languages (e.g., Haskell), thunks are shared by otherwise-pure data structures, and their role is to delay computation (in time) and share computational resources (namely, space and time) (Hudak et al., 1992;

Rudiak-Gould et al., 2006). Thunks are also used for the same purpose in eager higher-order languages, such as Standard ML (SML) (Milner et al., 1990a; Appel and MacQueen, 1991). In effectful higher-order languages (e.g., SML), thunks are commonly stored in data structures for the purpose of suspending and/or replaying computational side effects. These effects may include incrementing a counter, checking and setting environmental flags or accessing some other globally shared data structure. Effectful thunks are also relevant to the efficient implementation of self-adjusting computation.

Thunks and self-adjusting computation. As a somewhat complex example of an imperative, higher-order data structure, we describe the role that thunks play in implementing self-adjusting computation. As introduced above, self-adjusting computation is represented as an imperative higher-order data structure, variously referred to as the trace, or the dynamic dependence graph of the computation.

The trace consists of modifiable data and the thunks that depend on that data. Each thunk is crafted such that, when invoked, it will reperform the local computation, but with the new modifiable value in the place of the previous one. To do this, the thunk records the modifiable location as well as other contextual information required to later restore the local state, modulo a different value for the access.

The spatial knot. As introduced in Section 1.6.2, change propagation is operationally knotted with new execution. Given a data modification, change propagation finds the thunks in the trace that first depend on the modified data, and reexecutes them to update the computation and the output. To ensure that change propagation is efficient it uses a DDG representation where the modifiable data itself records the thunks that depend on it, which can change over time as the structure evolves. Hence, the modifiable data and the code on which it depends are knotted spatially, since each refers to the other.

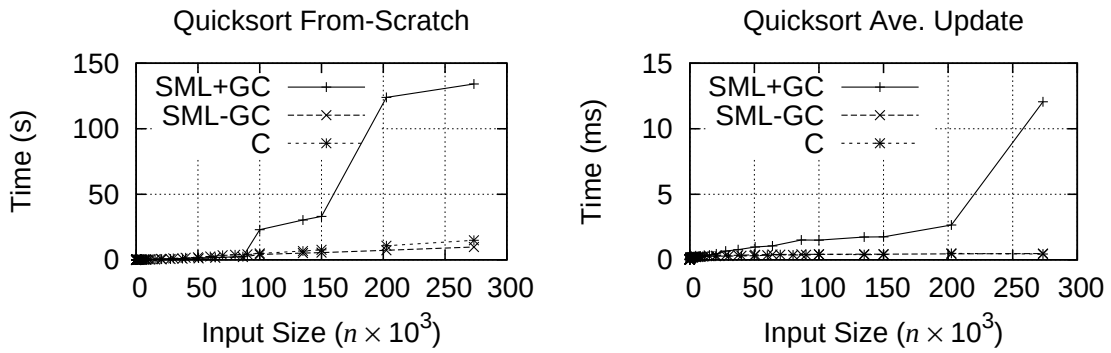

Figure 1.2: Multi-language comparison with **quicksort**.

As another example of how the higher-order and imperative features of traces are subtly knotted, consider how these traces are edited during change propagation. In particular, each traced thunk used for reexecution comes with a related, but distinct thunk, call it its "undo" thunk, whose role is to uninstall the first thunk from the trace. Change propagation uses these "undo" thunks to reverse an earlier effect on the trace; including the effect of installing the original thunk (that is, the one that generates replacement traces).

1.6.3 Limitations of high-level languages

While in many ways ideal for expressing self-adjusting computation, high-level languages have a central drawback: by being high-level, they give up control over low-level details. Specifically, detailed practical evaluations of self-adjusting computation show that memory management via traditional automatic garbage collection (GC) can be a significant cost.

As an example, Figure 1.2 shows timings for from-scratch runs and change propagation of a self-adjusting program. The program, a standard quicksort implementation, is taken directly from a recent implementation of self-adjusting computation in Standard ML (SML) (Acar et al., 2006). The program is compiled with the MLton compiler (Weeks, 2006) and run with a fixed heap of two gigabytes. The MLton compiler employs a mix of copying, generational, and mark-and-compact strategies based on the workload to deliver

efficient garbage collection. The graph on the left shows the running time of a from-scratch execution of the program on inputs with up to 275,000 elements. The graph on the right shows the average time for change propagation under an insertion/deletion of an element (average taken over all possible positions in the input). The lines labeled "SML+GC" show the time including the GC time and the lines labeled "SML-GC" shows the time excluding the GC time. In both graphs, GC time increases significantly with the input size. For comparison, we include a line labeled "C", showing timings with the approach described in this dissertation (Chapter 7 gives a detailed evaluation of our approach).

Self-adjusting programs are challenging for traditional automatic garbage collection for several reasons:

1. the total amount of live data at any time can be high,
2. many allocated objects have long life spans, and
3. old and new data interact in complex ways.

The total amount of live data is large because a self-adjusting computation stores the trace, which records the dependences between computation data, in memory. As a result, even for moderately-sized inputs the total amount of live data can be quite high. For example, with the SML quicksort example, when the input has 100,000 elements, the amount of live data can be as high as 650 megabytes. Next, trace elements tend to be long-lived because change-propagation is generally successful in reusing them. Finally, since the trace itself is a highly cyclic structure (see Section 1.6.2), after performing change propagation, old parts of the trace point to the new parts and vice versa; this establishes cyclic dependences between old and new data.

The cost of memory traversal. The characteristics described above make self-adjusting programs pessimal for memory-traversing collectors. To see this, recall that for traversal-based GC, the cost for each reclaimed memory cell is $\frac{1}{1-f} - 1$, where $f \in [0, 1]$ is the fraction

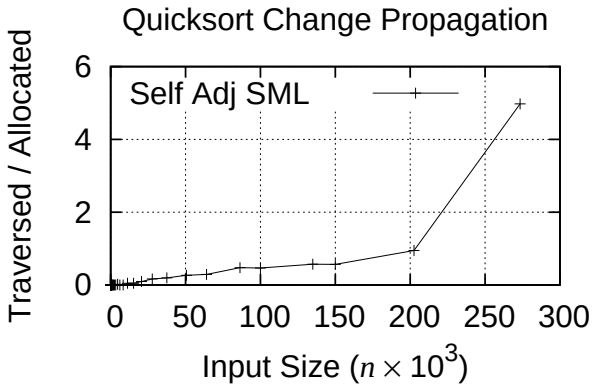

Figure 1.3: GC cost for quicksort in SML.

of memory that is live (Jones, 1996). We note that this function grows very quickly. For instance, consider the folllowing equation:

$$\frac{1}{1-f} = 1 + f + f^2 + f^3 + \cdots$$

As f approaches one (as memory fills and free space becomes vanishingly scarce), the traversal cost per reclaimed memory block approaches infinity.

We confirm the correlation between f and the GC costs by collecting some GC statistics. Figure 1.3 shows the GC cost for each allocated byte during the change-propagation experiments with quicksort (Figure 1.2). We measure the GC cost as the total number of bytes traversed by GC during the experiment divided by the total number of bytes allocated during the experiment. As can be seen, as the input size increases (requiring more live memory to hold the trace), the cost increases dramatically, essentially following the function $\frac{1}{1-f}$.

Avoiding memory traversal. Because of its high cost, automatic memory management management techniques try to avoid memory traversal when possible. However, these general approaches are not well-suited to the context of self-adjusting computation.

Reference counting. First, reference counting may perform better than traversal when free space is scarce, but it also has a large, constant-factor overhead (Jones, 1996). This overhead results from maintaining counters for each dynamically allocated block (where each counter counts the number of incoming pointers to the block), and frequently updating these counts to reflect changes in the pointer structure of memory. Furthermore, reference counting requires additional facilities for dealing with cyclic memory topologies.

Generational collection. Next, generational collectors mitigate the traversal cost by separating data into generations to avoid traversing all of memory. Self-adjusting computation, however, does not observe the *generational hypothesis*, which says that most garbage is newly created data (Hammer and Acar, 2008). Moreover, especially in the context of functional languages, generational assumptions often also say that pointer topology is related to the age of memory, positing that new data points to old data, but rarely the other way around. By virtue of its large, recursively-knotted execution trace, this assumption is also violated by self-adjusting computation.

Explicit management using domain knowledge. Intuitively, the automatic techniques discussed above stand in for lack of domain-specific knowledge: they attempt to automatically detect, using the global memory configuration of the system, when memory blocks are guaranteed to be dead (i.e., never accessed again). In some cases, it is possible to know this event with only local knowledge, by using domain-specific insights and reasoning principles. In these cases, the programmer can be explicit about how to manage the computation's memory resources. However, to gain control over these low-level details, they typically program in a low-level language, introduced below.

1.6.4 Low-level languages

As first explained in Section 1.1, by *low-level* language, we mean languages where:

- The type system describes memory layout.
- The program text is a flat, static resource.
- The call stack is a distinct, bounded resource.
- The program's management of the heap is explicit.

Concretely, we mean that both the programs and the underlying implementation are expressed in C (Kernighan and Ritchie, 1988). We note that while C does have a type system, it is essentially just a description language for describing the format of memory; it provides no safety guarantees.

Table 1.3 summarizes the contrasts between low-level and high-level languages, as they pertain to our context of self-adjusting computation. We explore the key aspects of these contrasts below.

Table 1.3: Low-level versus high-level languages

Language aspect	**Low-level language** (i.e., C)	**High-level language** (e.g., ML, Haskell)
Functions	Flat, globally-scoped	Nested, lexically-scoped
Data representation	Words (e.g., pointers)	Algebraic, higher-order
Data types	Describe memory format	Have a logical interpretation
Stack space	Distinct, bounded	Indistinct, unbounded
Heap space	Explicit reclamation	Implicit reclamation

In both high-level and low-level languages, programs consist of collections of functions and data type definitions. Sharp distinctions between low-level languages and high-level languages center around when information is known, how it is allowed to change, and how it is discarded. These differences give rise to the following dichotomy:

- Low level: statically weak, and dynamically explicit
- High level: statically strong, and dynamically implicit

We elaborate on this characterization below.

Program representation. In low-level languages, programs are dynamically explicit: functions are flat and globally-scoped; they cannot be partially applied (and further, in this dissertation we assume that their code is static and immutable); and a function invocation's local variables are stack-allocated, meaning that their lifetimes end when the corresponding function invocation returns. Programmers may construct closures, but must do so explicitly, i.e., by declaring top-level functions, declaring records for closing these functions, and constructing and managing space for each closure instance.

In high-level languages, programs are dynamically implicit: functions can nest and variables are lexically scoped; functions can be partially applied, and a function invocation may construct nested functions that escape, making lifetimes of this higher-order data generally undecidable. Hence, many operational (machine-oriented) details are unspecified, especially those related to the construction and reclamation of dynamic, higher-order data.

Data representation. Compared with a high-level language, the data that programmers represent in a low-level language often has a very machine-oriented structure. First, nearly everything is mutable (except the program text, usually). Next, no uniform representation

for data exists[6]. These two facts allow for complex memory structures: sharing[7], unboxing[8], cycles, pointer aliasing[9], address arithmetic[10] and bit-packing[11] are all common.

By contrast, in a high-level setting, the system imposes abstractions, under which it assumes that data has a uniform, boxed representation, and that only certain parts of it are mutable. These assumptions simplify the implementation of polymorphically-typed code generation and automatic garbage collection (Jones, 1996; Appel, 1998a).

Data types. As a result of these contrasts in data representation, the type system used in each level is afferent. In low-level settings, types just express the format of memory, and the type system does not enforce any strong, static properties. In fact, the "type" of a pointer (i.e., the format of the memory that it points at) may not even be known statically, or it may change dynamically, or both.

By contrast, in high-level settings, the Curry-Howard correspondence views a program's type as a logical proposition, and it views the program itself as a candidate proof term for this proposition. Operationally, this logical interpretation yields powerful, sound reasoning about what operations can occur to what operands, and in what contexts. Because it yields such strong reasoning principles, approaches to self-adjusting computation in high-level languages often revolve around a special extension of the underlying (higher-order) type

6. By *uniform representation* we mean that the system uses a memory representation such that the garbage collector's traversal of memory can precisely find and traverse every pointer. Languages such as C do not observe any uniform representation: they allow pointer arithmetic or other bit-level manipulations of pointers, of which the traversal may be unaware.

7. Sharing consists of common sub-structures being shared among super-structures; e.g., consider two trees that share a subtree, making the super-structure into a DAG.

8. Unboxing consists of storing in an aggregate block the elements themselves, not pointers to the elements; e.g., storing a tuple of tuples as a flat, array-like structure, rather than a tree-like structure.

9. Pointer aliasing consists of multiple pointers pointing at shared content, which we say is *aliased*. For instance, in low-level languages such as C, local variables can be aliased by pointers.

10. Instead of locations being abstract names, they are numbers that programs manipulate arithmetically.

11. For instance, when the least-significant bits of a pointer are used to store another field of either the containing structure, or the pointed-at structure.

system. See Section 2.2 for more details on work combining self-adjusting computation and type systems. Ensuring that the programmer uses self-adjusting primitives correctly relies on the soundness of the extended type system. By contrast, in low-level settings, there exists no sound type system to extend[12].

Heap and stack space. Programmers in low-level languages take control of explicitly managing the heap, and must be careful not to overuse the call stack, lest they overrun it. By contrast, automatic memory management is typically assumed in a higher-order setting, since automatic techniques are often viewed as a necessary consequence of higher-order programming: since function closures often capture pointers to other structures, the liveness of objects in a higher-order program is difficult to judge locally, without a global traversal of memory. Hence, manual management is thus not typically feasible, since this global information is lacking (Jones, 1996; Appel, 1998a), and because general, explicit management cannot be provided safely[13].

12. We note that a large and growing body of research consists of using powerful type systems or program logics to verify that the precarious programming patterns found in low-level languages can be used without fear of danger. However, at present, formal verification of low-level programs poses many challenges for researchers and proof engineers alike. In the meantime, we want a practical programming model that does not depend on the existence or usability of a powerful type system, even if it means that the system primitives can potentially be misused for lack of static checks, or other proofs of correct usage.

13. In Section 2.4 we discuss techniques that mix automatic memory management with some explicit control, in the form of region-based memory management.

CHAPTER 2

RELATED WORK

General-purpose, programming language-based techniques for building incremental computations have been studied since the early 1980's. We refer the reader to the survey by Ramalingam and Reps (1993) for a more detailed set of references for earlier work. In this chapter, we focus on the work most closely related to self-adjusting computation generally, and to self-adjusting machines, specifically. First, we characterize a selection of the most closely related early work from the 1980's, the 1990's and earlier (Section 2.1). Next, we characterize the development of self-adjusting computation (Section 2.2). Finally we contrast and compare to other contemporary work on incremental computation (Section 2.3) and garbage collection (Section 2.4).

2.1 Incremental computation: Early work

Of the many earlier language-based techniques proposed to support incremental computation, the most effective ones are dependence graphs, memoization, and partial evaluation. When surveying this early work, we characterize both *how* incremental constraints are specified by the programmer, and *what* incremental computations result.

Incremental constraints via dependence graphs. Broadly speaking, *dependence graphs* record the data dependency structure of an incremental computation. These structures come equipped with an associated change-propagation algorithm that updates the incremental computation when its input data is modified (Demers et al., 1981; Hoover, 1987). Depending on the system and the strategy taken by its change propagation algorithm, the dependency structures of the computation are limited to some class of expressiveness. The way that these dependencies are specified varies from system to system; below, we survey various approaches taken in the past.

Incremental constraints via attribute grammars. First proposed by Knuth (1968), attribute grammars offer a declarative programming model for giving context-sensitive semantics to otherwise context-free languages. That is, attribute grammars declaratively specify sets of constraints that relate data items whose inter-dependencies are defined by a tree structure (e.g., the context-sensitive attributes that decorate an abstract syntax tree, such as its typing information). By virtue of their declarative nature, attribute grammars deliberately sweep away the implementation details of attribute evaluation (i.e., they sweep away the underlying method used for solving the constraints). Different systems for attribute grammars support different attribute evaluation strategies, both conventional and incremental. These strategies give rise to different classes of attribute grammars, where each class is defined by the way it limits the dependency structure of its attributes. In the systems that support incremental attribute evaluation, after the attributed tree is changed, the system can sometimes update the affected attributes in optimal time (Reps, 1982a,b; Reps and Teitelbaum, 1984, 1989; Efremidis et al., 1993). The definition of *optimal* in this case is input-sensitive: it is based on counting of the minimum number of attributes that must be updated after a tree edit occurs; in general this is not known until such an incremental reevaluation is completed. In this sense, the change propagation algorithm used in these systems is similar to that of self-adjusting computation (see Section 1.6.2).

Incremental constraints over collection operations. The INC language (Yellin and Strom, 1991) allows programmers to work with aggregate data structures that incrementally change (e.g., sets, lists, etc.). By programming with these collections' operations, programmers implicitly specify constraints for incrementally-updated computations. However, the INC language does not permit general recursion, and consequently, the computations that it can describe are limited.

Incremental reuse via memoization. Memoization, also called *function caching* (Pugh and Teitelbaum, 1989; Liu and Teitelbaum, 1995; Abadi et al., 1996; Heydon et al., 2000), is a technique that can improve efficiency of any purely functional program wherein functions are applied repeatedly to equivalent arguments. In fact, this idea dates back to at least the late 1950's (Bellman, 1957; McCarthy, 1963; Michie, 1968). Due to its rather strict reliance on argument equivalence, however, sometimes even small data modifications can prevent memoization-based reuse in general incremental computations (Acar et al., 2006). These restrictions can be overcome, however, by integrating memoization with other complementary incremental techniques; several examples of this integration can be found in work on self-adjusting computation (Acar, 2005; Ley-Wild, 2010; Ley-Wild et al., 2011).

Efficiency improvements via partial evaluation. Partial evaluation has an incremental flavor: It consists of techniques where a program is given some if its input (which is fixed) and not given other input (which can change over time). Using this information, certain analyses and compilation techniques specialize the program to the fixed input, which speedup responses when the unfixed, changing input is modified (Sundaresh and Hudak, 1991; Field and Teitelbaum, 1990). The main limitation of this approach is that it allows input modifications only within a predetermined partition, and the user must determine the partitioning of the fixed and changing input parts before running the program. That is, a partition that is fixed statically, at *binding time* (viz., at compilation time).

2.2 Self-adjusting computation

Self-adjusting computation generalizes the data dependence graphs of earlier techniques (see above) by introducing *dynamic dependence graphs* (DDGs) (Acar et al., 2002). Unlike

earlier dependence graphs, these graphs are generated from programs written in a conventional programming language with general recursion. Later, Acar et al. (2004) combined these dynamic graphs with a form of memoization where partially inconsistent subgraphs can be rearranged, reused and consistently repaired, making the approach even more efficient and broadly applicable.

Self-adjusting computation first arose in high-level languages, where it benefited from their higher-order features. They offer programming interfaces within existing functional languages, namely, SML (Milner et al., 1990b) and Haskell (Hudak et al., 1992), either via a library (Acar et al., 2002; Carlsson, 2002; Acar et al., 2009) or with special compiler support (Ley-Wild et al., 2008a). Below, we survey the evolution of self-adjusting computation techniques, including those that support both purely functional and imperative programming models.

Monadic approaches, without memoization. Using Standard ML as their implementation and host language, Acar et al. (2002) were the first to formulate a general programming interface and algorithmic implementation for self-adjusting computation. Soon afterward, Carlsson (2002) showed how to adapt this programming model for Haskell, by exploiting certain Haskell features such as monads and type classes. In both cases, the early approaches of Carlsson (2002) and Acar et al. (2002) use monadic types in Haskell and SML, respectively; these systems both encapsulate incrementally modifiable state[1]. In both cases, this work did not incorporate the memoization aspect of self-adjusting computation, which Acar (2005) later showed was crucial for writing a large class efficient self-adjusting programs. We discuss the incorporation of memoization below.

Compared with later approaches, the monadic library interfaces place a comparatively heavy cognitive burden on programmer. This results from the programmer having to pro-

1. Acar et al. (2002) characterize incremental state as *modifiable*, a notion that they define.

gram in a higher-order, modal way, with one-shot modifiable references. A *one-shot reference* is one that is written exactly once. We say that this interface is modal because modifiables cannot be written or read unless in a special mode, represented by an abstract monadic type. Entering this monadic type requires programmers to supply a function argument that contains their program's local continuation. The write-once protocol is enforced using a higher-order, modal style: each local continuation of a read ends with a write operation; the write populates a modifiable reference and carries a monadic type. As a simple illustrative example, consider the pseudo code to compute the maximum value of two modifiable inputs, x and y, given in Figure 2.1.

```
let
  val m_max = mod (              -- make mod. ref., enter changeable mode
      read m_x ( fn x =>           -- read from modifiable holding x
      read m_y ( fn y =>           -- read from modifiable holding y
      write max ( x, y ) ) ) )     -- write to modifiable, leave chg. mode
in ···                           -- use modifiable holding max
```

Figure 2.1: $\max(x, y)$ expressed with modal, one-shot modifiable references.

To read a modifiable requires the population of another modifiable reference, which holds the incremental information gathered. Though stateful, these earlier systems enforced a kind of correlated pure execution, in the sense that all data structures built from modifiable references are acyclic. This acyclicity invariant is enforced by a combination of syntactic techniques which includes a special modal type system (Acar et al., 2002). Associated formal semantics and their proofs of consistency relied on the essential elements of this modal design with its purely-functional correspondence (Acar et al., 2007).

In the case of the early library-based SML implementations of self-adjusting computation, this higher-order modal/monadic style comes without any special assistance from the compiler. Specifically, if they want to access incremental state, the programmer must provide a function that populates a new piece of incremental state; Figure 2.1 gives an

example of this, where the programmer creates `m_max` to hold the result of accessing modifiable references `x` and `y` and computing their maximum.

To later access the newly created modifiable state of `m_max`, the programmer needs to provide another function and another modifiable reference. By using this higher-order interface, the programmer manually performs a compiler transformation of their program, explicitly creating λ-abstractions to record and suspend the control flow that follows each modifiable access (i.e., each modifiable access's continuation).

The reader may recognize this pattern as one that can be expressed through a monad (Wadler, 1995). Indeed, the recognition of this connection between incremental programming and monadic programming is at least a decade old (Carlsson, 2002). While Haskell has specific tool and language support for monads, the the incorporation of memoization into change propagation also requires a higher-order style, and to this author's knowledge, has not been incorporated into the work of Carlsson (2002).

Monadic approaches, with memoization. Memoization of dynamic dependency graphs is a key ingredient in self-adjusting computation (Acar et al., 2004), one that sharply distinguishes it from other work on programming with incrementally-changing state. In particular, DDG memoization gives rise to a larger class of efficient execution trace edits, and consequently, a larger class of efficient input changes. For instance, in recursive loops consuming recursive input structures (such as lists), memoization within the recursive loop allows for individual recursive iterations to be inserted or removed from the sequence, without having to reexecute recursive iterations that are not affected (Acar et al., 2004).

As with libraries for incremental state, libraries implementing memoization are naturally higher-order: given a client's function, and arguments to be applied (viz., a thunk) the library tries to search for a viable match in the prior execution; the client's function is applied only if no such match is found.

The initial approach with the combined features of incremental state and memoization were implemented as library-based approaches in SML (Acar et al., 2006). When programmers want to memoize a function call, as when they access modifiable memory, this library interface demands that they η-expand the code being memoized[2], essentially making all the higher-order plumbing of the memoized code explicit (Acar et al., 2006).

Hence, early approaches benefited heavily from being hosted by higher-order languages: recording thunks, reusing thunks and re-executing thunks constitutes the core behavior of self-adjusting computation. This programming style is expressive, in that it is general-purpose, but it is also inexpressive, in that doing simple tasks is highly cumbersome: Essentially, the programmer manually subdivides their code's logic into several, smaller pieces organized by the placement of modifiable accesses and memoization points.

The created boundaries are sometimes, but not always, in close correspondence with the natural conceptual boundaries of the program's logic. When the code boundaries required by the primitives are numerous, or misaligned with the code boundaries in the conceptual view, the code becomes harder to understand. The cognitive burden of programming in this style comes from having to think about and spell out all the higher-order plumbing, and mix it with the conceptual logic of the program. This mixing is burdensome to initially create; moreover, making small changes is harder compared with the uninstrumented version of the program. As such, some recent work on self-adjusting computation focuses on improving the programmer's experience.

Easing the programmer's burdens. In contrast to Acar et al. (2006), who use a library of modal self-adjusting primitives, Ley-Wild et al. give a more programmer-friendly approach where a compiler does the work to translate certain self-adjusting parts of an SML program into a specially-instrumented continuation-passing style (CPS) (Ley-Wild

2. An η-expansion of a arrow-typed term encloses the original term with an additional λ-abstraction.

et al., 2008a; Ley-Wild, 2010). A key feature of this program representation, as with the monadic style of earlier work, is that its higher-order control-flow becomes explicit (Friedman et al., 1984; Appel, 1991; Danvy and Hatcliff, 1993). Unlike earlier work, however, this representation is generated by the compiler, not by the programmer.

While Ley-Wild et al. (2008a) still uses a type system to delimit the self-adjusting code and enforce invariants in the way that incremental state is used, this type system gives first-order types to the operations that access and modify incremental state, as opposed to the higher-order types used in previous monadic systems[3]. Hence, a programmer can more directly see the conventional semantics of the program, without having to be explicit about the higher-order plumbing required to construct and save thunks. Though much easier to use than earlier approaches, the programmer must still be explicit about what is incrementally modifiable.

Chen et al. (2011) recently showed how a Hindley-Milner-style type system can be extended with implicit self-adjusting computation primitives, using type annotations to implicitly and polymorphically indicate where incremental modification can occur. After inferring where the (implicit) self-adjusting primitives must be placed, the compiler uses knowledge of the self-adjusting primitive types to both translate and optimize the self-adjusting target program (Chen et al., 2012).

In the systems above, self-adjusting programs have a purely-functional flavor, as modifiables must be written exactly once[4].

High-level, imperative semantics. By giving an imperative higher-order semantics, Acar et al. (2008) lifted the write-once restriction on modifiable state that earlier work im-

3. In Ley-Wild et al. (2008a)'s system, reading from incremental state is a first-order order operation, but it uses a special arrow type to distinguish it from pure, non-self-adjusting code, with which it can interoperate.

4. Through the use of special abstract data types, Ley-Wild (2010)'s complete system incorporates many extensions, including various imperative features. We discuss this further in Section 2.2.

posed on self-adjusting computations. Based on their imperative semantics, they describe a library-based implementation for SML, which resembles an earlier proposal (Acar et al., 2006), except extended with general imperative modifiables, as opposed to modal one-shot modifiables.

As mentioned above in Section 2.2, Ley-Wild (2010) makes self-adjusting programming easier by giving a compilation strategy based on a specially-instrumented continuation-passing-style (CPS) translation. In addition to providing one-shot modifiable references, he extends the special CPS translation to support more general imperative primitives (Acar et al., 2010). Unlike prior work, the programmer writes higher-order code when conceptually necessary; it is not required by the primitives for accessing incremental state[5].

High-level versus low-level semantics. The imperative semantics developed for self-adjusting computation in and for high-level languages are not well-suited for modeling low-level languages. By *low-level* we mean stack-based, machine-oriented languages that lack strong type systems and automatic memory management[6]. A low-level semantics of self-adjusting computation is one that explicitly manages its resources including the heap, stack and execution trace (Section 1.2 introduces these challenges).

A low-level account is missing in the imperative semantics of Acar et al. (2008). Instead, the semantics effectively relies on an oracle to generate reusable traces, and leaves the internal behavior of this oracle unspecified. Consequently, the oracle hides many of the practical issues that arise, such as how memory allocation and collection interact with trace reuse.

Ley-Wild (2010) gives an explicit account of trace editing, but his semantics requires the program be converted into continuation-passing style. By virtue of being based on

5. However, in Ley-Wild (2010)'s library interface, given in section 6.3, one sees that memoization still uses a higher-order programming interface.

6. See Section 1.6.4 for a longer introduction to the notion of *low-level* used in this dissertation.

continuation-passing-style, this approach is very general. For instance, this approach supports patterns of control flow more general than those limited to a stack discipline. For instance, their system supports exceptions; it is conceivable that general control operators such as call/cc could also be incorporated[7].

However, by being so general, the approach demands more than required for the specific case of low-level, stack-based programs, since these programs do not require the full range of incremental control-flow patterns expressible via continuations. Moreover, by maintaining the self-adjusting program's continuation as an incrementally modifiable structure, Ley-Wild (2010)'s approach requires powerful self-adjusting primitives in the target of the CPS translation. Namely, to efficiently incrementalize recursive flows of control, it requires separate memoization and keyed allocation primitives, as well as higher-order modifiable references.

The previously-purposed approaches for self-adjusting computation demand high-level host languages because they require higher-order functions in either the source language or the target language—as exemplified by the library and compilation approaches of Acar et al. (2008) and Ley-Wild (2010), respectively. By requiring high-level host languages, these approaches forgo control of low-level machine-oriented details. These low-level details, which are relevant for an operational, machine-oriented view, include the management of spatial resources such as the stack, execution trace and modifiable heap.

In Chapter 3 of his dissertation, Acar presents a machine model for self-adjusting computation that he calls the *closure machine* (Acar, 2005). Within the context of that work, the closure machine was used as a tool for algorithmic analysis (i.e., complexity analysis for the algorthmics presented in other chapters). From a distance this model is quite similar to the one proposed in this thesis; however, upon closer inspection we note important

7. *Call-with-current-continuation*, often abbreviated as *call/cc*, is a programming primitive that captures the current execution context and binds it to a program variable; this primitive can be used to build very general operations to manipulate a program's control-flow (Friedman et al., 1984).

distinctions: (1) there is no notion of trace memoization, just allocation memoization (so-called *keyed allocation*); (2) there is no characterization of the call stack size with respect to change propagation; (3) there is no characterization of memory management; (4) there is no attempt to connect this model with either a formally-proven semantics, nor a usable surface language. Indeed, this model is the seed for the work presented in this dissertation, but leaves much left to be determined.

2.3 Incremental computation: Contemporary work

Researchers working within and outside the context of self-adjusting computation continue to address incremental computations in a variety of contexts. We survey these below.

Parallelism. More recent work generalizes self-adjusting computation techniques to support parallel computations. We briefly survey current challenges. Hammer et al. (2007) presents a preliminary algorithm for parallel change propagation which relaxes the total order trace structure imposed by sequential self-adjusting computation to a partial order. However, the paper does not incorporate any memoization technique, which due to its interaction with the execution trace, seems to be a difficult technique to generalize to a setting of general-purpose parallel execution. Other work considers parallel self-adjusting computation for individual problems (Acar et al., 2011; Sümer et al., 2011), as well the map-reduce framework (Bhatotia et al., 2011), a more general setting.

Burckhardt et al. (2011) consider a parallel setting where concurrently-executed threads communicate using primitives that resemble that of a revision control system, e.g., CVS (Vesperman, 2003) or subversion (Pilato, 2004). In particular, these primitives tame parallel thread programming by limiting their interaction patterns to graph structures that they refer to as *revision diagrams*. The dependencies arising from these directed graphs consist of a special lattice structure (Burckhardt and Leijen, 2011). It is this lattice structure is that

is cached, reevaluated and incrementally reused. However, due to their limited support for memoization, certain incremental changes to revision diagrams prevent efficient reuse. For instance, compare the arithmetic expression tree evaluation example handled by both Hammer et al. (2009) and Demetrescu et al. (2011): When an internal node in the tree is inserted or removed, these sequential approaches can recover the work associated with the unaffected subtree beneath it (see Section 3.6 for a detailed discussion of this example in the context of this dissertation). By contrast, since this edit of the input tree affects the parent-child relationships of the revision diagram that evaluates it, the comparatively fragile memoization technique proposed by Burckhardt et al. (2011) requires a complete reevaluation of the affected subtree.

DITTO**: incremental invariant checks for Java.** DITTO offers support for incremental invariants-checking in Java (Shankar and Bodik, 2007). It requires no programmer annotations but only supports a purely-functional subset of Java. DITTO also places further restrictions on the programs; while these restrictions are reasonable for expressing invariant checks, they also narrow the scope of the approach.

Reactive imperative programming. Demetrescu et al. (2011) consider a class of dynamic dependencies that arise from one-way data-flow constraints. Their techniques represent a combination of high and low-level techniques: They give high-level abstractions for creating and constraining *reactive memory*, the memory that holds incrementally-changing values; to efficiently implement this memory, they rely on clever low-level techniques at the level of the operating system kernel. They perform empirical comparisons to CEAL that show that their techniques are very competitive, and that their approach can be more efficient (in terms of time) for the problems that they consider.

IncPy**: memoization for scientific workloads in python** Guo and Engler (2011) introduce a variant of python that specifically targets scientists and scientific work flows. They evaluate their tool in case studies with real scientists' scripts and get speedups in these contexts via automatic (file system-level) caching of interemdiate results. Unlike earlier work on memoization (or *function caching*), the work of Guo and Engler (2011) begins with an effectful, IO-oriented scripting language (Python) and specifically targets research scientists; in this context, it gives a practical tool to an audience in need. The tool works by recording dependencies between files and python scripts, where it gives coarse resuse of a function's results whose input (files and data) are not affected. Hence, while useful and demonstrably practical in scientific settings, this work does not offer a general solution for creating fine-grained incremental computations: It lacks the ability to adjust efficiently to fine-grained changes within incrementally-changing datasets.

2.4 Garbage collection

Garbage collection has been shown to be effective in improving programmer productivity often without decreasing performance significantly (Zorn, 1993; Berger et al., 2002). Numerous techniques for garbage collection (McCarthy, 1960; Collins, 1960; Baker, 1978; Diwan et al., 1993; Tofte and Talpin, 1997) have been proposed and studied; we refer the reader to Cohen (1981) and Wilson (1992) for surveys.

It is worth noting that much garbage collection research is done with assumptions that make sense for conventional programming, but which are violated by self-adjusting computation. As one example, it is typically assumed that garbage is much more likely to be young than old. This assumption leads to approaches that are badly misaligned with the setting of incremental or self-adjusting computation, such as ones where reclaimation cost is proportional to total object lifetime (Lieberman et al., 1983).

Traversal-based collection. As outlined in Section 1.6.3, non-traversal collection exists. However, it typically comes as a form of reference counting, which has its own serious shortcomings and limitations (e.g., the difficultly of detecting and collecting cyclic data structures). Hence, most general-purpose techniques for collecting memory consist of some kind of traversal of the pointer-structure of memory. Many garbage collection techniques can be classified based on how they perform this traversal (e.g., it may be done sequentially, incrementally, concurrently, or in parallel, etc.), and what assumptions they make about the memory being traversed (e.g., it may or may not have a uniform, tagged representation).

Of the proposed techniques, sequential non-incremental traversal of memory with a uniform representation is common in high-level languages. As outlined in Section 1.6.3, these high-level langauges make the wrong assumptions about memory use in self-adjusting computation, and their traversal of memory leads to performance that degrades quickly as the heap becomes occupied (e.g., by a large execution trace).

Customized memory traversal. Researchers have explored ways of customizing existing traversal-based collectors. For instance, traversing collectors have been extended to offer domain-specific customization in the form of *simplifiers* (O'Neill and Burton, 2006). Simplifiers can help reduce total live memory by running programmer-supplied code during memory traversal that mutates the heap in a semantics-preserving way. As a second example, researchers can also piggyback other actions on the memory traversal of GC, such as domain-specific invariant checks (Aftandilian and Guyer, 2009; Reichenbach et al., 2010).

In contrast to customized traversal-based approaches, self-adjusting machines perform basic memory management of the trace without fully traversing it, by instead integrating memory management into change propagation (as opposed to, for instance, the other way around with change propagation as a subroutine of garbage collection).

Region-based memory management. Region-based memory management shows that explicit low-level and automatic high-level techniques can be mixed effectively. These techniques apply when several allocations' lifespans coincide and this shared lifespan can be proven to end by a certain execution point (Ruggieri and Murtagh, 1988; Tofte and Talpin, 1997; Grossman et al., 2002; Hallenberg et al., 2002; Tofte et al., 2004). In these cases, the allocations are coalesced into a shared *region* of memory. The detection of regions may be made explicit by the programmer or be inferred by the system; in either case, the decision comes with a check and a proof of soundness. Regions are most notably used to mimic the stack-allocated local variables of C: all stack variables of a stack frame have a lifespan that begins when the frame is pushed, and ends when the frame is popped. However, regions need not follow a stack discipline, and many systems explore extensions that permit other patterns (See Tofte et al. (2004) for a retrospective and a more detailed survey of related systems).

Although self-adjusting machines do not use region-based memory management techniques per se, in the future it might be possible to describe their management technique based on them. However, instead of relying on a particular type system explored by past work, one would require more complex semantic reasoning to determine when a region becomes garbage. In particular, when change propagation revokes an interval from the self-adjusting machine's execution trace, we need to exploit the knowledge the memory regions allocated by this interval will become garbage at the end of change propagation. The soundness of this reasoning holds because change propagation yields a computation that is consistent with a from-scratch execution, a fact that is difficult (or impossible) to establish without domain-specific knowledge (See Section 4.4 for an example of such an argument, based on a formal model of a self-adjusting machine).

CHAPTER 3

SURFACE LANGUAGE

We describe a C-based surface language called CEAL for writing low-level self-adjusting programs. We introduce the programming model used by CEAL (Section 3.1), which is based on the notion of *levels*. We describe the programming abstractions modifiable memory (Section 3.2), and the outer and inner levels of a CEAL program (Sections 3.3 and 3.4). We describe how self-adjusting machine resources interact in the CEAL programming model (Section 3.5). We illustrate the programming model with two simple examples (Sections 3.6 and 3.7). Finally, we describe certain practical concerns (Sections 3.8 and 3.9).

3.1 Programming model

The CEAL language is a C-like surface language for writing low-level programs that contain self-adjusting computations. The CEAL language is closely related to the C language: It restricts some features of C, and it adds several new features for creating and managing self-adjusting computations. A CEAL program consists of several subprograms, which we variously refer to simply as *programs* or *levels*. These levels consist of the non-self-adjusting *outer level*, the self-adjusting *inner level*, and optionally, the *foreign level*. In this section, we introduce these levels conceptually. We give a detailed discussion of the outer, inner and foreign levels in Sections 3.3, 3.4 and 3.9, respectively.

The role of the inner level is to define the behavior of self-adjusting programs. The role of the outer level is to create and manage the self-adjusting programs written in the inner level, including the creation and mutation of the inner level's input data. The role of the foreign level is to write conventional C code that interoperates with both the inner and outer levels, which are both written in CEAL. Such foreign code is is useful, for

instance, if either the inner or outer levels wish to call certain standard C library functions, either when these functions' interfaces do not obey the restrictions imposed on CEAL code (Section 3.9), or when the library interfaces consist of auxilliary functions or state that are included (from their corresponding C header files) into the CEAL program text.

The outer and inner levels are written in the CEAL language, and are compiled with the CEAL compiler. The foreign level is written in conventional C code, and is separately compiled and linked with the output of the CEAL compiler. While foreign C code can be written separately from CEAL code and linked later, it is often more convienent to intermix these levels in a single source file. Moreover, for certain common implementations of standard C header files, this mixing is difficult or impossible to avoid[1]. Our compiler carefully separates each level before it compiles the inner and outer CEAL code to target C code; it recombines this target C code with any foreign C code extracted from the inputed source code.

The C language can be viewed as giving a portal (machine-independent) syntax for writing low-level, resource-aware code. In a similar way, the CEAL language can be viewed as giving a portal syntax for writing low-level, resource-aware code for an extension of the underyling machine model. In this extended model, the underlying (self-adjusting) machine model has an additional resource, namely, that of a self-adjusting execution trace. This machine model is implemented by CEAL by building on the C language, by extending it with a special C-based run-time library that provides implementations of the traced inner-level primitives.

1. In these cases, the library header files include auxilliary functions or state which, through the C preprocessor, are included into the CEAL source file before it reaches the CEAL compiler.

3.2 Modifiable memory

Modifiable memory consists of those memory locations on which the behavior of self-adjusting computations depend, whose values generally change over time. Past language designs for self-adjusting computation distinguish modifiable data from non-modifiable data, and typically require that the programmer explicitly declare and create modifiable data differently from non-modifiable data. In CEAL, however, modifiable data implicitly consists of all *non-local memory*, which we define below. This assumption simplies the initial implementation of self-adjusting programs by CEAL programmers: The language makes the conservative assumption that all non-local data dependencies may be relevant to the modifiable output of the self-adjusting computation. To increase performance, CEAL allows programmers to refine this (safe) conservative assumption with simple annotations, which we introduce in Section 3.8. Using these annotations, programmers explicitly qualify certain fields, indicating if and how they change value.

Local versus non-local memory. In CEAL, *local memory* consists of local function variables that are not explicitly qualified as changing, and which are not aliased by pointers. Conservatively, CEAL defines *non-aliased* local variables as those (unqualified) variables whose addresses are not taken, i.e., they are not used with the address-of operation `&` of C/CEAL. In the current CEAL language, local memory must also be scalar-sized (viz., the size of a pointer or smaller). The CEAL language considers local variables with non-scalar-sized aggregate types (viz., those with a multi-field `struct` type) to also be non-local, even if they are not pointer-aliased[2]. When local variables do not meet the criteria for being treated as local memory (if they are too large in size, or have the potential to be pointer-aliased), the CEAL language implicitly promotes them to non-local memory.

2. This design choice merely simplifies the CEAL compiler's conversion of the program's local memory into static single assignment (SSA) form; this conversion can be generalized to multi-field structures where fields of structures are treated in an SSA-like way, when possible.

Non-local memory consists of all other memory locations that are not local by the definition above. These memory locations consist of those for global variables, heap data, statically-allocated data[3] , and local variables whose addresses are taken, and thus may be aliased by pointers. Since each of these memory locations can potentially be affected by both the inner and outer levels, they naturally fill the role of modifiable references: They offer a shared space for communicating changing inputs and outputs between the inner and outer levels.

Memory allocation and reclamation. Our surface language provides a single `alloc` primitive for allocating memory parameterized by a type, as in the following examples:

```
1 T* x = alloc( T );
2 int y* = alloc ( long );
3 struct myrecord* z = alloc ( struct myrecord );
```

Whether the allocation is collected automatically depends on its allocation context. In the context of the outer level, reclaimation is manual via the `kill` primitive:

```
1 kill( x ); kill( y ); kill( z );
```

In the context of the inner level reclaimation is automatic and implicit (furthermore, the `kill` primitive is ignored).

3.3 The outer level

The outer level of a CEAL program consists of all of its non-self-adjusting code (but still excludes foreign C code). Its role is create and manage the self-adjusting computations of the inner level. To do so, CEAL augments C with the following additional keywords:

- The `inner_self` primitive creates a new self-adjusting computation.

3. In both C and CEAL, this data is indicated by the `static` keyword; it resides within the local scope of a function, but is not stack-allocated.

- The `propagate` primitive change-propagates the current self-adjusting computation.

The `inner_self` primitive has a higher-order flavor: It is parameterized by a CEAL function and a list of arguments to apply. For instance, to create a self-adjusting computation from the application of the function `compute` on the arguments `arg1` and `arg2`, the CEAL programmer writes the following code at the outer level:

```
1 int result = inner_self ( compute ) ( arg1, arg2 ) ;
```

This use of this (outer level) primitive instansiates a new self-adjusting computation at the inner level that consists of an initial (from-scratch) evaluation of the `compute` function's application to the given arguments `arg1` and `arg2`.

Once run initially, the modifiable memory written at the inner level can be read by the outer level as modifiable output. Though shown above returning a value, the CEAL language currently restricts functions used with `inner_self` to return `void`. This language restriction can be relaxed using a compiler transformation similar to one already implemented by the compiler (viz., destination-passing-style conversion). Alternatively, programmers can always apply this conversion manually, by adding an explicit destination argument to the functions that they use with `inner_self`:

```
1 int result ;
2 inner_self ( compute ) ( arg1, arg2, & result ) ;
```

By taking its address to pass by reference, this code conversion implicitly moves the local variable `result` into local-local, modifiable memory, where it holds the changing return value of the inner computation.

After receiving an initial output of the inner level, the outer level can mutate the input memory of the inner level; aferwards, it can automatically update the output of the inner level using the `propagate` primitive. No other parameters are required for this primitive; it implicitly change-propagates the current self-adjusting computation's execution trace:

```
propagate ;
```

While in general the outer level program can simultaneously manage multiple self-adjusting inner programs, for simplicitly we assume throughout this chapter that each outer program manages at most one inner program. We give a more general interface to creating and change-propagating inner programs in Chapter 6, in the context of CEAL's run-time system implementation.

Determining inner versus outer levels. The CEAL language does not require that programmers declare the level of functions a priori. That is to say, in CEAL, functions are not tied to either the inner or outer level at the exclusion of the other; rather, the level of a function depends on the context of its use in the CEAL program. In the listing above, the function `compute` is neither intrinsically inner nor outer level; it can be used at both levels, and its status as "inner level" above is deterined by its use with `inner_self`. When used in a outer level context, appearances of inner level primitives (described below) are ignored; similarly, when used at the inner level, uses of the outer level `kill` primitive are ignored. Currently, CEAL forbids using either `inner_self` or `propagate` at the inner level, and our compiler issues a compile-time error in both cases.

3.4 The inner level

The role of the inner level is to define both the initial (from-scratch) behavior of a self-adjusting computation, as well its incremental-update behavior during change propagation. This behavior is determined by the way that the inner level (implicitly) controls the execution trace resource exposed by the CEAL language abstractions. In this sense, the language name *CEAL* can aptly be expanded to *C-based, Execution-Adjusting Language*.

The CEAL surface language gives a compositional, portal way of describing this trace and controling it as a resource. Though it masks tedious details, CEAL simultaneously

provides a minimal layer of abstraction over the run-time interface, which consists of all the low-level details of trace construction and maintenance. Specifically, CEAL separates the programmer from the low-level details of saving and restoring local state from the trace, as well as from the details of determining exactly which execution points should be traced and recorded, versus executed but untraced. Though not always made syntactically explicit in concrete syntax, the CEAL programmer remains in control over these decisions, as we explain below.

The traces created by CEAL inner programs have a structure that is determined in three complementrary, but ultimately orthogonal ways. These three dimensions of control correspond to three resources of the underlying self-adjusting machine model:

- **Modifiable memory** records non-local state changes.
- The **control stack** manages nested frames of local state.
- The **execution trace** correlates local state with past execution on non-local state.

We explain the concepts behind each machine resource above. We focus on how these concepts appear to the programmer, and how they help the programmer to control the structure of inner program's execution trace.

Modifiable memory operations consist of allocating, dereferencing and updating non-local state (i.e., modifiable memory). These inner-level effects are traced implicitly, in the sense that no special syntax is used to distinguish pointer dereferences that are traced (as a modifiable memory dependency), versus those that are not. Rather, the programmer supplies type qualifiers to distinguish the later case (Section 3.8).

Stack operations consist of push and pop operations, which are implicit in C syntax; they correspond to (most) function calls[4]. In addition to being implicit around every (non-tail-recursive) function call, the CEAL language primitives include syntax for inserting explicit pairs of balanced push and pop operations. For instance, the following code isolates statement S_2 from statements that preceede and follow it by employing a *cut block*.

```
1    S1 ; cut { S2 } ; S3
```

The extensional meaning of this code is equivalent to the statement sequence $S_1; S_2; S_3$.

Intensionally, the cut block alters the stack behavior of the code via *function outlining*, which consists of the following (non-local, tedious) code transformation: create a new function abstraction f_{S_2} whose body is S_2; the parameters of function f_{S_2} consist of the live local variables upon statement S_2's entry; the return values of function f_{S_2} consist of the live local variables upon statement S_2's exit; replace the cut block with a call to function f_{S_2}, passing and returning the live local state to and from f_{S_2}. Clearly, this is a meaning-preserving transformation, in terms of extensional semantics (i.e., ignoring resource usage such as the call stack). This primitive is significant only in an intensional sense, which we discuss further below.

Trace checkpoints consist of the points during execution where a snapshot of the current local state is saved in the trace. These points come in two varieties: *update points* and *memoization points* (or *memo points* for short). Semantically, memo points and update points both preserve the extensional meaning of the program; they both alter the intensional meaning during change propagation, in terms of information saved in the execution

4. Some C compilers may transform certain function calls into non-returning jumps, as an optimization. In these cases, the stack does not grow in size. This is commonly refered to as *tail-call* optimization, since it optimizes those function calls in *tail position*, i.e., immediately before the calling function returns control to its caller.

trace. Syntactically, memo points are explicit (and "opt-in" where desired), and update points are implicit[5] . We explain their various uses below.

In the case of both checkpoint varieties, we say that these points *guard* the execution of a subtrace. This subtrace is a subtree of the execution trace; it corresponds to the interval of execution consisting of the checkpoint's local continuation, up to and including the end of its immediate stack frame. Using cut blocks, introduced above, the CEAL programmer can further refine this tree structure easily; these blocks are syntactic sugar that insert additional pairs of balanced stack operations (a stack push followed by a corresponding stack pop).

Memo points. Programmers can explicitly add memo points using a simple memo statement primitive, as in the following code:

```
1  S1 ; memo ; S2
```

Extensionally, the meaning of this code is equivalent to the statement sequence $S_1; S_2$. Intensionally, the memo primitive *guards* statement S_2 with a memo point that is identified by this (static) program point, along with the current *live local state* upon entry to S_2 (or equivalently, the live local state upon exit of S_1).

Update points. By default, the CEAL language associates every read of non-local memory with an implicit update point that dominates it immediately. Compilers for CEAL can implicitly coalesce these update points during compilation when they can be safely shared by larger blocks of code (our current compiler does this optimization to a limited extent).

5. In Section 8.1, we describe future work that makes update points explicit, and provides a way to "opt-out" of their implicit positioning by the CEAL compiler.

3.5 Resource interaction

The CEAL language allows programmers to control the underlying stack and trace resources using orthogonal primitives that permit local reasoning, and localized evolution of program logic (i.e., the primitives can be inserted and removed using local code changes). Since they permit local changes, they lead to straight-forward evolution of code from having coarse-grained to fine-grained incremental behavior. Yet, the intensional meaninings of these primitives, while highly orthogonal in nature, still witness interactions that are inherent to resource interactions in the underlying machine model (viz., self-adjusting machines).

In the CEAL programming model, (and more generally, in the *self-adjusting machine* programming model), stack operations affect the trace's shape, and consequently, these operations help to define the possible dynamic behaviors when traces adjust to changes. Recall that, ignoring the trace resource, our stack operations are conventional in terms of their intensional semantics: Push operations extend the control stack of the underlying machine with a new stack frame that saves a return control context; pop operations (alternatively, *returns*) remove the topmost evaluation context, and return control to it, either passing no extra information (in the case of a `void` return type), or perhaps one or more values (viz., the return values).

Stack operations implicitly control the trace structure. In addition to managing local state and implicitly controlling the set of live local variables (viz., the local state saved and restored from the trace at checkpoints), stack operations also define the nesting structure of the trace, which can be thought of both as a temporal sequence with nested subsequences, and as a tree with nested subtrees. In particular, these stack operations determine the branching structure of the corresponding execution trace; as a first approximation, one

can think of the execution trace as an incrementally-constructed call graph (though the information stored therein is generally more fine-grained than that of a call-graph).

During execution of the inner-level program, the trace grows in a depth-first order: Statements extend the current execution interval, push operations begin a nested interval (a nested subtree); pop operations end the current nested interval (nested subtree), and resume extending the top-most interval. To better see how stack operations determine the trace structure, consider a balanced pair of push and pop operations defined by the cut block of the following code:

```
1 S1 ; cut{ S2 } ; S3
```

The cut block isolates the interval of execution for S_2 (and its trace T_2) from that of the surrounding execution context (and its trace $T_1; [\,]; T_3$). In this sense, balanced pairs of push/pop operations define the nesting structure of the trace (viz., the nested intervals of the sequence, under the sequence-oriented view).

Many update or control points may be executed before a particular execution point in question. However, each control point in the code is *guarded immediately* by at most one unique memo point, and at most one unique update point. These immediate guardians are defined by the CEAL language to be those memo/update points that *immediately dominate* the point in question[6]. When reasoning about the incremental behavior defined by the inner level of a CEAL program, it are these immediate guards that determine the finest granularity at which execution intervals (trace subtrees) can be reevaluated (via an update point) or reused (via a memo point). Hence, by having control over their placement, the CEAL programmer can implicitly control (by implicitly extending or limiting) the possible behaviors of the trace during change propagation.

6. Intuitively, the immediate dominator relation is a tree where parent-child relationships are determined by domination; in turn, a control point dominates another if all possible control paths that reach the point in question must first travel through the candidate domination point.

Stack operations implicitly control the trace dynamics. As described above, trace checkpoints determine *where* local state is saved in the trace; stack operations define *what subset* of local state is saved in the trace (viz., those variables that are live upon entry to and exit from the nested execution interval). Given this structure, the two checkpoint varieties play dual roles within the dynamics of the trace during change propagation. Memo points save the local state for the purposes of caching and resuing the work performed by the guarded code, including its resulting return value and its trace. Update points save the local state for the purpose of incrementally reevaluating the work performed by the guarded code; this reevaluation occurs if and when the modifiable values accessed by the corresponding subtrace change value (in which case these inconsistencies in the prior trace will be replaced).

Syntactic sugar. Since memo points and cut blocks often go together in practice, CEAL introduces syntanctic sugar that defines a *memo block* with the following syntax:

```
1  memo { S }
```

The memo block above is equivalent to a cut block whose body is immediately guarded by a memo point, as in the following:

```
1  cut { memo ; S }
```

Further, CEAL generalizes memo and cut forms beyond C blocks and statements, and allows them both to be used in the context of C expressions, as below:

```
1  T1 x1 = memo(e1);
2  T2 x2 = cut (e2);
```

These have an analogous meaning to their block-form counterparts:

```
1  T1 x1; memo { x1 = e1 };
2  T2 x2; cut { x2 = e2 };
```

```
typedef struct node_s* node_t;
struct node_s {
   enum { LEAF, BINOP } tag;
   union { int leaf_val;
           struct { enum { PLUS, MINUS } op;
                    node_t left, right; } binop;
         } u; };
```

Figure 3.1: Type declarations for expression trees in C.

```
int eval (node_t root) {
   if (root->tag == LEAF)
      return root->u.leaf_val;
   else {
      int l = eval (root->u.binop.left);
      int r = eval (root->u.binop.right);
      if (root->u.binop.op == PLUS) return (l + r);
      else return (l - r);
} }
```

Figure 3.2: The `eval` function in C.

3.6 Example: Reducing Trees

For our first example, we consider a simple evaluator for expression trees, as expressed with user-defined C data structures. These expression trees consist of integer-valued leaves and internal nodes that represent the binary operations of addition and subtraction. Figure 3.1 shows their representation in C. The `tag` field (either `LEAF` or `BINOP`) distinguishes between the `leaf_val` and `binop` fields of the **union** `u`. Figure 3.2 gives a simple C function that evaluates these trees.

Suppose we first run `eval` with an expression tree as shown on the left in Figure 3.3; evaluating $((3+4)-0)+(5-6)$, the execution will return the value 6. Suppose we then change the expression tree to $((3+4)-0)+((5-6)+\mathbf{5})$ as shown in Figure 3.3 on the

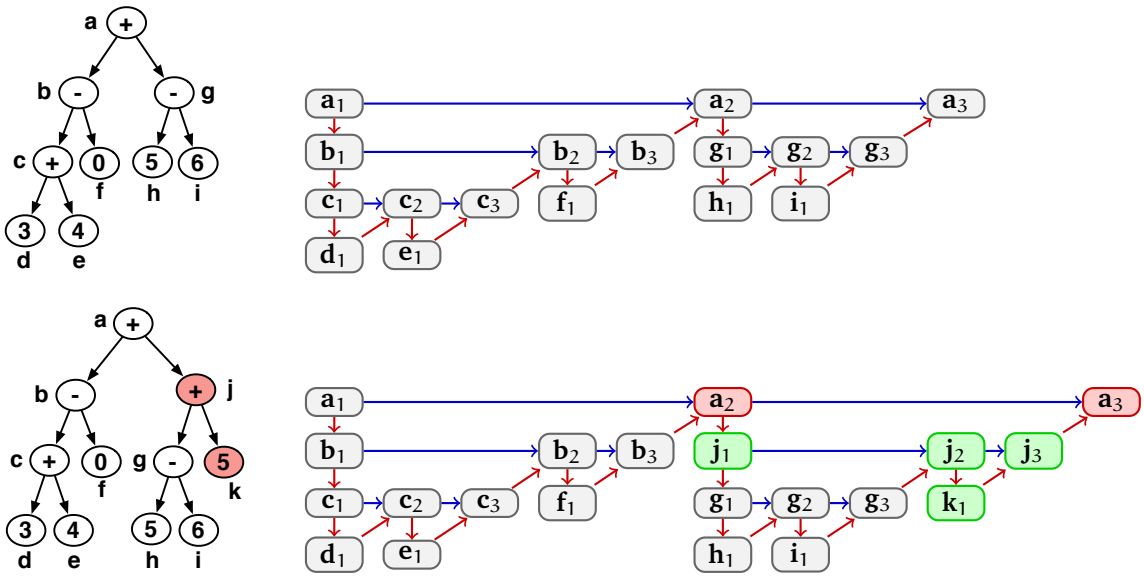

Figure 3.3: Example input trees (left) and corresponding execution traces (right).

right. How shall change propagation efficiently update the output?

Strategy for change propagation. We first consider the computation's structure, of which Figure 3.3 gives a summary: the upper and lower versions summarize the computation before and after the change, respectively. Their structure reflects the stack behavior of `eval`, which divides each invocation into (up to) three fragments: Fragment one checks the tag of the node, returning the leaf value, if present, or else recurring on the left subtree (lines 2–5); fragment two recurs on the right subtree (line 6); and fragment three combines and returns the results (lines 7–8).

In Figure 3.3, each fragment is labeled with a tree node, e.g., $\mathbf{b}_2$ represents fragment two's execution on node **b**. The dotted horizontal arrows indicate pushing a code fragment on the stack for later. Solid arrows represent the flow of control from one fragment to the next; when diagonal, they indicate popping the stack to continue evaluation.

Based on these two computations' structure, we informally sketch a strategy for change propagation. First, since the left half of the tree is unaffected, the left half of the compu-

tation ($\mathbf{a}_1$–$\mathbf{b}_3$) is also unaffected, and as such, change propagation should reuse it. Next, since the right child for **a** has changed, the computation that reads this value, fragment $\mathbf{a}_2$, should be reevaluated. This reevaluation recurs to node **g**, whose subtree has not changed. Hence, change propagation should reuse the corresponding computation ($\mathbf{g}_1$–$\mathbf{g}_3$), including its return value, -1. Comparing $\mathbf{j}_1$–$\mathbf{j}_3$ against $\mathbf{g}_1$–$\mathbf{g}_3$, we see that **a**'s right subtree evaluates to 4 rather than -1. Hence, change propagation should reevaluate $\mathbf{a}_3$, to yield the new output of the program, 11.

Complexity of change propagation. To consider the complexity of change propagation for this expression tree evaluation program, let us first generalize the change considered above to a class of changes of unit size. Define this class of (unit sized) changes as all of those that insert, remove or replace a constant-sized, continuous group of nodes from the tree (e.g., either inserting a binary operation, removing a binary operation, or replacing a constant-sized subtree with a different constant-sized subtree). Across this class of changes, we expect that the change propagation strategy above generalizes: a constant-sized group of work is inconsistent and must be replaced (viz., the work performed directly on the changed input), some group of work can be reused (viz., the work associated with the subtrees that are not changed). Some subset of return values are also affected (viz., those return values that correspond to the path from the highest affected input position back up to the root of the tree). Hence, the expected complexity for handing this class of changes is dependent on the length of the longest pathes in the tree; the complexity of change propagation for such changes is proportional to its expected length.

Challenges for change propagation. For change propagation to use the strategy sketched above, it must identify dependencies among data and the three-part structure of this code,

```
int MAX;

void array_max(int* arr, int len) {
   while(len > 1) {
      for(int i = 0; i < len - 1; i += 2) {
         int m;
         max(arr[i], arr[i + 1], &m);
         arr[i / 2] = m;
      }
      len = len / 2;
   }
   MAX = arr[0];
}
```

Figure 3.4: Iteratively compute the maximum of an array.

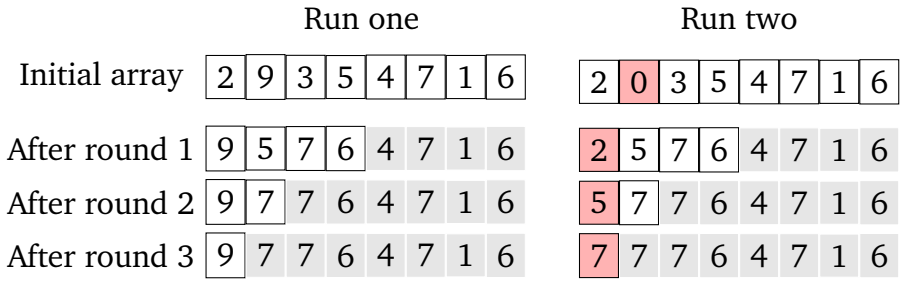

Figure 3.5: Snapshots of the array from Figure 3.4.

including its call/return dependencies. In particular, it must identify where previous computations should be reused, reevaluated or discarded[7].

3.7 Example: Reducing Arrays

As a second example, Figure 3.4 gives C code for (destructively) computing the maximum element of an array. Rather than perform a single linear scan, it finds this maximum iteratively by performing a logarithmic number of *rounds*, in the style of a (sequentialized)

7. To see an example where computation is discarded, imagine the change in reverse; that is, changing the lower computation into the upper one.

data-parallel algorithm. For simplicity, we assume that the length of arrays is always a power of two. Each round combines pairs of adjacent elements in the array, producing a sub-sequence with half the length of the original. The remaining half of the array contains inactive elements no longer accessed by the function.

Rather than return values directly, we illustrate commonly used imperative features of C by returning them indirectly: function `max` returns its result by writing to a provided pointer, and `array_max` returns its result by assigning it to a special global variable `MAX`.

Figure 3.5 illustrates the computation for two (closely-related) example inputs. Below each input, each computation consists of three snapshots of the array, one per round. For readability, the inactive elements of the array are still shown but are greyed, and the differences between the right and left computation are highlighted on the right.

Strategy for change propagation. We use Figure 3.5 to develop a strategy for change propagation. Recall that each array snapshot summarizes one round of the outer `while` loop. Within each snapshot, each (active) cell summarizes one iteration of the inner `for` loop. That `array_max` uses an iterative style affects the structure of the computation, which consequently admits an efficient strategy for change propagation: reevaluate each affected iteration of the inner `for` loop, that is, those summarized by the highlighted cells in Figure 3.5.

Complexity of change propagation. It is simple to (manually) check that each active cell depends on precisely two cells in the previous round, affects at most one cell in the next round, and is computed independently of other cells in the same round. Hence, for a single input change, at most one such iteration is affected per round. Since the number of rounds is logarithmic in the length of the input array, this change propagation strategy is efficient.

Challenges of change propagation. To efficiently update the computation, change propagation should reevaluate each affected iteration, being careful not to reevaluate any of the unaffected iterations.

3.8 Type qualifiers

The CEAL programmer uses type qualifiers to explicitly indicate how memory changes over time, if at all. The set of qualifiers is not fixed, and is existensible by library programmers. The basic set of built-in qualifiers includes a *stable* qualifier, for qualifying fixed-value memory that need not be traced at all. It also includes a "one-shot" qualifier, for marking memory as changing, but only in constrained way. A common example of such memory, *one-shot* memory, consists of memory that is only ever written once in within a single execution, but whose "one-write" value can change across successive executions (e.g., destinations that hold a function's return values are examples of one-shot memory).

3.9 Foreign level

The CEAL compiler accepts source files that consist of a mixture of inner and outer code with foreign C code. Foreign C code is neither inner nor outer code; it does not directly interact with self-adjusting machines or their modifiable memory. Rather, it can call outer called, and be called by both inner and outer code.

The presence of foreign C code has several purposes. First, CEAL allows programmers to mix foreign (non-CEAL) C code into their CEAL source files, to act as utility functions or wrappers around standard library code. Wrapping calls to foreign C libraries is required whenever the interfaces of these libraries require clients to operate in destination-passing style, where they provide the library with a pointer to store results. It is most convenient to allocate this space temporarily, directly on the C stack; to gain access to this stack as

a mutable structure that is non-modifiable (not traced as a modifiable), we isolate the mutated stack frame by defining it with foreign C code.

CHAPTER 4

ABSTRACT MACHINES

In this chapter, we describe techniques for sound self-adjusting computation which are suitable for low-level languages. We present an abstract self-adjusting machine model that runs programs written in an intermediate language for self-adjusting computation that we simply call IL. We give an informal introduction to the design of IL (Section 4.1), and using with examples from Chapter 3, we illustrate example IL programs (Section 4.2). We give the formal syntax of IL, to which we give two semantics, by defining two abstract machines: The *reference machine* models conventional evaluation semantics, while the *tracing machine* models the trace structure and self-adjusting semantics of a self-adjusting machine (Section 4.3).

Our low-level setting is reflected by the abstract machines' configurations: each consists of a store, a stack, an environment and a program. Additionally, the tracing machine's configurations include a trace component. We show that automatic memory management is a natural aspect of automatic change propagation by defining a notion of garbage collection. Under certain conditions, we show that this self-adjusting semantics is (extensionally) consistent (Section 4.4). We give a simple transformation that achieves these necessary conditions (Section 4.5). Finally, we give a cost model framework to relate the intensional semantics of self-adjusting machines to that of the reference semantics (Section 4.6).

4.1 Introduction to IL

The primary role of IL is to make precise the computational dependencies and possible change propagation behaviors of a low-level self-adjusting program. In particular, it is easy to answer the following questions for a program when expressed in IL:

- Which data dependencies are *local* versus *non-local*?

```
let TAG = 0 in
let LEAF_VAL = 1 in
let OP = 1 in
let LEFT = 2 in
let RIGHT = 3 in

let eval (node) =
  memo
  let eval_right (l) =
    let eval_op (r) =
      update
      let op = read (node[OP]) in
      if (op == PLUS) then pop (l+r)
      else pop (l-r)
    in
    push eval_op do
        update
      let right = read (node[RIGHT]) in
      eval (right)
  in
  update
    let tag = read (node[TAG]) in
    if (tag == LEAF)
      let leaf_val = read (node[LEAF_VAL]) in
      pop (leaf_val)
    else
      push eval_right do
      update
        let left = read (node[LEFT]) in
        eval (left)
```

Figure 4.1: The `eval` CEAL function of Figure 3.2, translated into in IL.

```
let for_loop (i) =
    let m_ptr = alloc(1) in
    let after_max() = update
        let m_val = read(m_ptr[0]) in
        let _ = write(arr[i/2], m_val) in
        if (i < len - 1)
            then for_loop(i + 2)
            else ...
    in
    push after_max do update
        let a = read(arr[i]) in
        let b = read(arr[i + 1]) in
        max(a, b, m_ptr)
in for_loop(0)
```

(a)

```
let for_loop (i) =
    let m_ptr = alloc(1) in
    let after_max() = update
        let m_val = read(m_ptr[0]) in
        let _ = write(arr[i/2], m_val) in
        memo
            if (i < len - 1)
                then for_loop(i + 2)
                else ...
    in
    push after_max do update
        let a = read(arr[i]) in
        let b = read(arr[i + 1]) in
        max(a, b, m_ptr)
in for_loop (0)
```

(b)

```
let for_loop (i) =
    let for_next () =
        if (i < len - 1) then for_loop(i + 2)
            else ...
    in
    push for_next do
        let m_ptr = alloc(1) in
        let after_max() = update
            let m_val = read(m_ptr[0]) in
            let _ = write(arr[i/2], m_val) in
            pop ()
        in
        push after_max do update
            let a = read(arr[i]) in
            let b = read(arr[i + 1]) in
            max(a, b, m_ptr)
in for_loop(0)
```

(c)

Figure 4.2: Three versions of IL code for the `for` loop in Figure 3.4; highlighting indicates their slight differences.

- Which code fragments are saved on the *control stack*?
- Which computation fragments are saved in the computation's *trace*, for later reevaluation or reuse?

We informally introduce the syntax and semantics of IL by addressing each of these questions for the examples in Sections 3.6 and 3.7. In Section 4.3, we make the syntax and semantics precise.

Static Single Assignment. To clearly separate local and non-local dependencies, IL employs a (functional variant of) static single assignment form (SSA) (Appel, 1998b). Within this representation, the control-flow constructs of C are represented by locally-defined functions, *local state* is captured by let-bound variables and function parameters, and all *non-local state* (memory content) is explicitly allocated within the store and accessed via **read**s and **write**s.

For example, we express the `for` loop from Figure 3.4 as the recursive function `for_loop` in Figure 4.2(a). This function takes an argument for each variable whose definition is dependent on the `for` loop's control flow[1], in this case, just the iteration variable `i`. Within the body of the loop, the local variable `m` is encoded by an explicit store allocation bound to a temporary variable `m_ptr`. Although not shown, global variable `MAX` is handled analogously. This kind of indirection is necessary whenever assignments can occur non-locally (as with global variables like `MAX`) or via pointer indirection (as with local variable `m`). By contrast, local variables `arr`, `i` and `len` are only assigned directly and locally, and consequently, each is a proper SSA variable in Figure 4.2(a). Similarly, in Figure 3.2 the assignments to `l` and `r` are direct, and hence, we express each as a proper SSA variable in

1. Where traditional SSA employs ϕ-operators to express control-dependent variable definitions, functional SSA uses ordinary function abstraction.

Figure 4.1. We explain the other IL syntax from Figures 4.1 and 4.2(a) below (**push**, **pop**, **update**, **memo**).

Stack operations. As our first example illustrates (Section 3.6), the control stack necessarily breaks a computation into multiple fragments. In particular, before control flow follows a function call, it first pushes on the stack a code fragment (a local continuation) which later takes control when the call completes.

The stack operations of IL make this code fragmentation explicit: the expression **push** `f` **do** *e* saves function `f` (a code fragment expecting zero or more arguments) on the stack and continues by evaluating *e*; when this subcomputation **pop**s the stack, the saved function `f` is applied to the (zero or more) arguments of the **pop**.

In Figure 4.1, the two recursive calls to `eval` are preceded by **push**es that save functions `eval_right` and `eval_op`, corresponding to code fragments for evaluating the right subtree (fragment two) and applying the binary operator (fragment three), respectively. Similarly, in Figure 4.2(a), the call to `max` is preceded by a **push** that saves function `after_max`, corresponding to the code fragment following the call. We note that since `max` returns no values, `after_max` takes no arguments.

Reevaluation and reuse. To clearly mark which computations are saved in the trace—which in turn defines which computations can be reevaluated and reused—IL uses the special forms **update** and **memo**, respectively.

The IL expression **update** *e*, which we call an *update point*, has the same meaning as *e*, except that during change propagation, the computation of *e* can be recovered from the program's original computation and reevaluated. This reevaluation is necessary exactly when the original computation of *e* contains **read**s from the store that are no longer consistent within the context of new computation.

Dually, the IL expression **memo** *e*, which we call a *memo point*, has the same meaning as *e*, except that during reevaluation, a previous computation of *e* can be reused in place the present one, provided that they *match*. Two computations of the same expression *e* match if they begin in locally-equivalent states (same local state, but possibly different non-local state). This notion of memoization is similar to function caching (Pugh and Teitelbaum, 1989) in that it reuses past computation to avoid reevaluation, but it is also significantly different in that impure code is supported, and non-local state need not match (a matching computation may contain inconsistent **read**s). We correct inconsistencies by reevaluating each inconsistent **read** within the reused computation.

We can insert **update** and **memo** points freely within an existing IL program without changing its meaning (up to reevaluation and reuse behavior). Since they allow more fine-grained reevaluation and reuse, one might want to insert them before and after every instruction in the program. Unfortunately, each such insertion incurs some tracing overhead, as **memo** and **update** points each necessitate saving a snapshot of local state.

Fortunately, we can automatically insert a smaller yet equally effective set of **update** points by focusing only on **read**s. Figures 4.1 and 4.2(a) show examples of this: since each **read** appears within the body of an **update** point, we can reevaluate these **read**s, including the code that depends on them, should they become inconsistent with memory. We say that each such **read** is *guarded* by an **update** point.

For memo points, however, it is less clear how to automatically strike the right balance between too many (too much overhead) and not enough (not enough reuse). Instead, we expose surface syntax to the C programmer, who can insert them as statements (`memo;`) as well as expressions (e.g., `memo(f(x))`). In Section 4.2, we discuss where to place memo points within our running examples.

4.2 Example programs

In Sections 3.6 and 3.7, we sketched strategies for updating computations using change propagation. Based on the IL representations described in Section 4.1, we informally describe our semantics for change propagation in greater detail. The remainder of the chapter makes this semantics precise and describes our current implementation.

Computations as traces. We represent computations using an execution trace, which records the **memo** and **update** points, store operations (**alloc**s, **read**s and **write**s), and stack operations (**push** and **pop**).

To a first approximation, change propagation of these traces has two aspects: reevaluating inconsistent subtraces, and reusing consistent ones. Operationally, these aspects mean that we need to decide not only which computations in the trace to reevaluate, but also where this reevaluation should cease.

Beginning a reevaluation. In order to repair inconsistencies in the trace, we begin reevaluations at **update** points that guard inconsistent **read**s. We identify **read**s as inconsistent when the memory location they depend on is affected by **write**s being inserted into or removed from the trace. That is, a **read** is identified as *affected* in one of two ways: when inserting a newly traced **write** (of a different value) that becomes the newly read value, or when removing a previously traced **write** that had been the previously read value. In either case, the **read** in question becomes inconsistent and cannot be reused in the trace without first being reevaluated. To begin such a reevalaution, we restore the local state from the trace and reevaluate within the context of the current memory and control stack, which generally both differ from those of the original computation.

Ending a reevaluation. We end a reevaluation in one of two ways. First, recall that we begin reevaluation with a different control stack than that used by the original computa-

tion. Hence, we will eventually encounter a **pop** that we cannot correctly reevaluate, as doing so requires knowing the contents of the original computation's stack. Instead, we cease reevaluation at such **pop**s. We justify this behavior below and describe how it still leads to a sound approach.

Second, as described in Section 4.1, when we encounter a **memo** point, we may find a matching computation to reuse. If so, we cease the current reevaluation and begin reevaluations that repair inconsistencies within the reused computation, if any.

Example 1 revisited. The strategy from Section 3.6 requires that the previous computation be reevaluated in some places, and reused in others. First, as Figure 4.1 shows, we note that however an input tree is modified, **update** points guard the computation's affected **read**s. We reevaluate these update points. For instance, in the given change (of the right subtree of **a**), line 9 has the first affected **read**, which is guarded by an **update** point on line 8; this point corresponds to $\mathbf{a}_2$, which we reevaluate first. Second, our strategy reuses computation $\mathbf{g}_1$–$\mathbf{g}_3$. To this end, we can insert a `memo` statement at the beginning of function `eval` in Figure 3.2 (not shown), resulting in the **memo** point shown on line 1 in Figure 4.1. Since it precedes each invocation, this **memo** point allows for the desired reuse of unaffected subcomputations.

Example 2 revisited. Recall that our strategy for Section 3.7 consists of reevaluating iterations of the inner `for` loop that are affected, and reusing those that are not. To begin each reevaluation within this loop (Figure 4.2(a)), we reevaluate their **update** points.

Now we consider where to cease reevaluation. Note that the **update** point in `after_max` guards a **read**, as well as the recursive use of `for_loop`, which evaluates the remaining (possibly unaffected) iterations of the loop. However, recall that we do not want reevaluation to continue with the remaining iterations—we want to reuse them.

We describe two ways to cease reevaluation and enable reuse. First, we can insert a `memo` statement at the end of the inner `for` loop in Figure 3.4, resulting in the **memo** point shown in Figure 4.2(b). Second, we can wrap the `for` loop's body with a *cut block*, written `cut{...}`, resulting in the additional **push**-**pop** pair in Figure 4.2(c). Cut blocks are optional but convenient syntactic sugar: their use is equivalent to moving a code block into a separate function (hence the **push**-**pop** pair in Figure 4.2(c)). Regardless of which we choose, the new **memo** and **pop** both allow us to cease reevaluation immediately after an iteration is reevaluated within Figures 4.2(b) and 4.2(c), respectively.

Call/return dependencies. Recall from Section 3.6 that we must be mindful of call/return dependencies among the recursive invocations. In particular, after reevaluating a subcomputation whose return value changes, the consumer of this return value (another subcomputation) is affected and should be reevaluated ($\mathbf{a}_3$ in the example).

Our general approach for call/return dependencies has three parts. First, when proving consistency (Section 4.4), we restrict our attention to programs whose subcomputations' return values do not change, a crucial property of programs that we make precise in Section 4.4. Second, in Section 4.5, we provide an automatic transformation of arbitrary programs into ones that have this property. Third, in Section 5.8, we introduce one simple way to refine this transformation to reduce the overhead that it adds to the transformed programs. With more aggressive analysis, we expect that further efficiency improvements are possible.

Contrasted with proving consistency for a semantics where a fixed approach for call/return dependencies is "baked in", our consistency proof is more general. It stipulates a property that can be guarenteed by either of the two transformations that we describe (Sections 4.5 and 5.8). Furthermore, it leaves the possibility open for future work to improve the currently proposed transformations, e.g., by employing more sophosticated static analysis to further reduce the overhead that they introduce.

e	$::=$	e^u	Untraced expression
	$\mid$	e^t	Traced expression
e^u	$::=$	**let fun** $f(\overline{x}).e_1$ **in** e_2	Function definition
	$\mid$	**let** x **=** $\oplus(\overline{v})$ **in** e	Primitive operation
	$\mid$	**if** x **then** e_1 **else** e_2	Conditional
	$\mid$	$f\ (\overline{x})$	Function application
e^t	$::=$	**let** x **=** ι **in** e	Store instruction
	$\mid$	**memo** e	Memo point
	$\mid$	**update** e	Update point
	$\mid$	**push** f **do** e	Stack push
	$\mid$	**pop** $\overline{x}$	Stack pop
ι	$::=$	**alloc**(x)	Allocate an array of size x
	$\mid$	**read**$(x[y])$	Read yth entry at x
	$\mid$	**write**$(x[y],z)$	Write z as yth entry at x
v	$::=$	$n \mid x$	Natural numbers, variables

Figure 4.3: IL syntax.

4.3 IL: a self-adjusting intermediate language

We present IL, a self-adjusting intermediate language, as well as two abstract machines that evaluate IL syntax. We call these the *reference machine* and the *tracing machine*, respectively. As its name suggests, we use the first machine as a reference when defining and reasoning about the tracing machine. Each machine is defined by its own transition relation over similar machine components. The tracing machine mirrors the reference machine, but includes additional machine state components and transition rules that work together to generate and edit execution traces. This tracing behavior formalizes the notion of IL as a self-adjusting language.

Abstract syntax of IL. Figure 4.3 shows the abstract syntax for IL. Programs in IL are expressions, which we partition into traced e^t and untraced e^u. This distinction does not constrain the language; it merely streamlines the technical presentation. Expressions in IL follow an administrative normal form (ANF) Flanagan et al. (1993) where (nearly) all

values are variables.

Expressions consist of function definitions, primitive operations, conditionals, function calls, store instructions (ι), **memo** points, **update** points, and operations for pushing (**push**) and popping (**pop**) the stack. Store instructions (ι) consist of operations for allocating (**alloc**), reading (**read**) and writing (**write**) memory. Values v include natural numbers and variables (but not function names). Each expression ends syntactically with either a function call or a stack **pop** operation. Since the form for function calls is syntactically in tail position, the IL program must explicitly push the stack to perform non-tail calls. Expressions terminate when they **pop** on an empty stack—they yield the values of this final pop.

Notice that IL programs are first-order: although functions can nest syntactically, they are not values; moreover, function names f, g, h are syntactically distinct from variables x, y, z. Supporting either first-class functions (functions as values) or function pointers is beyond the scope of the current work, though we believe our semantics could be adapted for these settings[2].

In the remainder, we restrict our attention to programs (environments ρ and expressions e) that are well-formed in the following sense:

1. They have a unique arity (the length of the value sequence they potentially return) that can be determined syntactically.

2. All variable and function names therein are distinct. (This can easily be implemented in a compiler targeting IL.) Consequently we don't have to worry about the fact that IL is actually dynamically scoped.

2. For example, to model function pointers, one could adapt this semantics to allow a function f to be treated as a value if f is closed by its arguments; this restriction models the way that functions in C admit function pointers, a kind of "function as a value", even though C does not include features typically associated with first-class functions (e.g. implicitly-created closures, partial application).

Machine configurations and transitions. In addition to sharing a common expression language (viz. IL, Section 4.3), the reference and tracing machines share common *machine components*; they also have related *transition relations*, which specify how these machines change their components as they run IL programs.

Machine configurations. Each machine configuration consists of a handful of components. Figure 4.4 defines the common components of two machines: a store (σ), a stack (κ), an environment (ρ) and a command (α_r for the reference machine, and α_t for the tracing machine). The tracing machine has an additional component—its trace—which we describe in Sections 4.3 and 4.3.

A store σ maps each store *entry* ($\ell[n]$) to either *uninitialized* contents (written $\perp$) or a machine value v. Each entry $\ell[n]$ consists of a store location ℓ and a (natural number) offset n. In addition, a store may mark a location as garbage, denoted as $\ell \mapsto \diamond$, in which case all store entries for ℓ are undefined. These garbage locations are not used in the reference semantics; in the tracing machine, they help to define a notion of garbage collection. A stack κ is a (possibly empty) sequence of *frames*, where each frame $\lfloor \rho, f \rfloor$ saves an evaluation context that consists of an environment ρ and a function f (defined in ρ). An *environment* ρ maps variables to machine values and function names to their definitions.

In the case of the reference machine, a *(reference) command* α_r is either an IL expression e or a sequence of machine values $\overline{v}$; for the tracing machine, a *(tracing) command* α_t is either e, $\overline{v}$, or an additional command **prop**, which indicates that the machine is performing *change propagation* (i.e., replay of an existing trace).

Each *machine value* v consists of a natural number n or a store location ℓ. Intuitively, we think of machine values as corresponding to machine words, and we think of the store as mapping location-offset pairs (each of which is itself a machine word) to other machine words.

For convenience, when we do not care about individual components of a machine configuration (or some other syntactic object), we often use underscores (_) to avoid giving them names. The quantification should always be clear from context.

Transition relations. In the reference machine, each machine configuration, written $\sigma, \kappa, \rho, \alpha_r$, consists of four components: a store, a stack, an environment and a command, as described above. In Section 4.3, we formalize the following stepping relation for the reference machine:

$$\sigma, \kappa, \rho, \alpha_r \xrightarrow{r} \sigma', \kappa', \rho', \alpha_r{}'$$

Intuitively, the command α_r tells the reference machine what to do next. In the case of an expression e, the machine proceeds by evaluating e, and in the case of machine values $\overline{v}$, the machine proceeds by popping a stack frame $\lfloor \rho, f \rfloor$ and using it as the new evaluation context. If the stack is empty, the machine terminates and the command $\overline{v}$ can be viewed as giving the machine's results. Since these results may consist of store locations, the complete extensional result of the machine must include the store (or at least, the portion reachable from $\overline{v}$).

The tracing machine has similar machine configurations, though it also includes a pair $\langle \Pi, T \rangle$ that represents the current trace, which may be in the midst of adjustment; we describe this component separately in Sections 4.3 and 4.3. In Section 4.3, we formalize the following stepping relation for the tracing machine:

$$\langle \Pi, T \rangle, \sigma, \kappa, \rho, \alpha_t \xrightarrow{t} \langle \Pi', T' \rangle, \sigma', \kappa', \rho', \alpha_t{}'$$

At a high level, this transition relation accomplishes several things: (1) it "mirrors" the semantics of the reference machine when evaluating IL expressions; (2) it traces this evaluation, storing the generated trace within its trace component; and (3) it allows previously-generated traces to be either reused (during change propagation), or discarded (when

$$
\begin{array}{rrcl}
\textit{Store} & \sigma & ::= & \varepsilon \mid \sigma[\ell[n] \mapsto \bot] \mid \sigma[\ell[n] \mapsto v] \mid \sigma[\ell \mapsto \diamond] \\
\textit{Stack} & \kappa & ::= & \varepsilon \mid \kappa \cdot \lfloor \rho, f \rfloor \\
\textit{Environment} & \rho & ::= & \varepsilon \mid \rho[x \mapsto v] \mid \rho[f \mapsto \textbf{fun } f(\overline{x}).e] \\
\textit{Reference command} & \alpha_r & ::= & e \mid \overline{v} \\
\textit{Tracing command} & \alpha_t & ::= & \alpha_r \mid \textbf{prop} \\
\textit{Machine value} & v & ::= & n \mid \ell
\end{array}
$$

Figure 4.4: Common machine components.

$$
\frac{\rho' = \rho[f \mapsto \textbf{fun } f(\overline{x}).e_1]}{\sigma, \kappa, \rho, \textbf{let fun } f(\overline{x}).e_1 \textbf{ in } e_2 \xrightarrow{r} \sigma, \kappa, \rho', e_2} \textbf{ R.1}
$$

$$
\frac{\rho(v_i)_{i=1}^{|\overline{v}|} = \overline{v} \qquad \rho' = \rho[x \mapsto \textbf{primapp}(\oplus, \overline{v})]}{\sigma, \kappa, \rho, \textbf{let } x = \oplus(\overline{v}) \textbf{ in } e \xrightarrow{r} \sigma, \kappa, \rho', e} \textbf{ R.2}
$$

$$
\frac{\rho(x) \neq 0}{\sigma, \kappa, \rho, \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2 \xrightarrow{r} \sigma, \kappa, \rho, e_1} \textbf{ R.3}
$$

$$
\frac{\rho(x) = 0}{\sigma, \kappa, \rho, \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2 \xrightarrow{r} \sigma, \kappa, \rho, e_2} \textbf{ R.4}
$$

$$
\frac{\rho(f) = \textbf{fun } f(\overline{x}).e \qquad \rho' = \rho[x_i \mapsto \rho(x_i)]_{i=1}^{|\overline{x}|}}{\sigma, \kappa, \rho, f\ (\overline{x}) \xrightarrow{r} \sigma, \kappa, \rho', e} \textbf{ R.5}
$$

$$
\frac{\sigma, \rho, \iota \xrightarrow{s} \sigma', v}{\sigma, \kappa, \rho, \textbf{let } x = \iota \textbf{ in } e \xrightarrow{r} \sigma', \kappa, \rho[x \mapsto v], e} \textbf{ R.6} \qquad \frac{}{\sigma, \kappa, \rho, \textbf{memo } e \xrightarrow{r} \sigma, \kappa, \rho, e} \textbf{ R.7}
$$

$$
\frac{}{\sigma, \kappa, \rho, \textbf{update } e \xrightarrow{r} \sigma, \kappa, \rho, e} \textbf{ R.8} \qquad \frac{}{\sigma, \kappa, \rho, \textbf{push } f \textbf{ do } e \xrightarrow{r} \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \rho, e} \textbf{ R.9}
$$

$$
\frac{\overline{v} = \rho(x_i)_{i=1}^{|\overline{x}|}}{\sigma, \kappa, \rho, \textbf{pop } \overline{x} \xrightarrow{r} \sigma, \kappa, \varepsilon, \overline{v}} \textbf{ R.10} \qquad \frac{\rho(f) = \textbf{fun } f(\overline{x}).e \qquad \rho' = \rho[x_i \mapsto v_i]_{i=1}^{|\overline{x}|}}{\sigma, \kappa \cdot \lfloor \rho, f \rfloor, \varepsilon, \overline{v} \xrightarrow{r} \sigma, \kappa, \rho', e} \textbf{ R.11}
$$

Figure 4.5: Stepping relation for reference machine ($\xrightarrow{r}$).

they cannot be reused). To accomplish these goals, the tracing machine distinguishes machine transitions for change propagation from those of normal execution by giving change propagation the distinguished command **prop**.

$$\frac{\ell \notin \mathsf{dom}(\sigma) \qquad \sigma' = \sigma[\ell[i] \mapsto \bot]_{i=1}^{\rho(x)}}{\sigma, \rho, \textbf{alloc}(x) \xrightarrow{s} \sigma', \ell} \textbf{S.1} \qquad \frac{\sigma(\rho(x)[\rho(y)]) = v}{\sigma, \rho, \textbf{read}(x[y]) \xrightarrow{s} \sigma, v} \textbf{S.2}$$

$$\frac{\sigma' = \sigma[\rho(x)[\rho(y)] \mapsto \rho(z)]}{\sigma, \rho, \textbf{write}(x[y],z) \xrightarrow{s} \sigma', 0} \textbf{S.3}$$

Figure 4.6: Stepping relation for store instructions ($\xrightarrow{s}$).

Reference machine transitions. Figure 4.5 specifies the transition relation for the reference machine, as introduced in Section 4.3. A function definition updates the environment, binding the function name to its definition. A primitive operation first converts each value argument v_i into a machine value v_i using the environment. Here we abuse notation and write $\rho(v)$ to mean $\rho(x)$ when $v = x$ and n when $v = n$. The machine binds the result of the primitive operation (as defined by the abstract **primapp** function) to the given variable in the current environment. A conditional steps to the branch specified by the scrutinee. A function application steps to the body of the specified function after updating the environment with the given arguments. A store instruction ι steps using an auxiliary judgement (Figure 4.6) that allocates in, reads from and writes to the current store. An **alloc** instruction allocates a fresh location ℓ for which each offset (from 1 to the specified size) is marked as uninitialized. A **read** (resp. **write**) instruction reads (resp. writes) the store at a particular location and offset. A **push** expression saves a return context in the form of a stack frame $\lfloor \rho, f \rfloor$ and steps to the body of the **push**. A **pop** expression steps to a machine value sequence $\overline{v}$, as specified by a sequence of variables. If the stack is non-empty, the machine passes control to function f, as specified by the topmost stack frame $\lfloor \rho, f \rfloor$, by applying f to $\overline{v}$; it recovers the environment ρ before discarding this frame. Otherwise, if the stack is empty, the value sequence $\overline{v}$ signals the termination of the machine with results $\overline{v}$.

$$
\begin{array}{rrcl}
\textit{Trace} & \mathsf{T} & ::= & \mathsf{t}\cdot\mathsf{T} \mid \varepsilon \\
\textit{Trace action} & \mathsf{t} & ::= & \mathsf{A}_{\ell,n} \mid \mathsf{R}^{v}_{\ell[n]} \mid \mathsf{W}^{v}_{\ell[n]} \mid \mathsf{M}_{\rho,e} \mid \mathsf{U}_{\rho,e} \mid (\mathsf{T}) \mid \overline{v} \\
\textit{Trace context} & \Pi & ::= & \varepsilon \mid \Pi\cdot\mathsf{t} \mid \Pi\cdot\square \mid \Pi\cdot\boxplus_{\mathsf{T}} \mid \Pi\cdot\boxminus_{\mathsf{T}}
\end{array}
$$

Figure 4.7: Traces, trace actions and trace contexts.

The structure of the trace. The structure of traces used by the tracing machine is specified by Figure 4.7. They each consist of a (possibly empty) sequence of zero or more *trace actions* t. Each action records a transition for a corresponding traced expression e^{t}.

In the case of store instructions, the corresponding action indicates both the instruction and each machine value involved in its evaluation. For **alloc**s, the action $\mathsf{A}_{\ell,n}$ records the allocated location as well as its size (i.e., the range of offsets it defines). For **read**s ($\mathsf{R}^{v}_{\ell[n]}$) and **write**s ($\mathsf{W}^{v}_{\ell[n]}$) the action stores the location and offset being accessed, as well as the machine value being read or written, respectively. For **memo** expressions, the trace action $\mathsf{M}_{\rho,e}$ records the body of the memo point, as well as the current environment at this point; **update** expressions are traced analogously. For **push** expressions, the action (T) records the trace of evaluating the **push** body; it is significant that in this case, the trace action is not atomic: it consists of the arbitrarily large subtrace T. For **pop** expressions, the action $\overline{v}$ records the machine values being returned via the stack.

There is a close relationship between the syntax of traced expressions in IL and the structure of their traces. For instance, in nearly all traced expressions, there is exactly one subexpression, and hence their traces $\mathsf{t}\cdot\mathsf{T}$ contain exactly one subtrace, T. The exception to this is **push**, which can be thought of as specifying two subexpressions: the first subexpression is given by the body of the **push**, and recorded within the push action as (T); the second subexpression is the body of the function being pushed, which is evaluated when the function is later popped. Hence, push expressions generate traces of the form $(\mathsf{T})\cdot\mathsf{T}'$, where T' is the trace generated by evaluating the pushed/popped function.

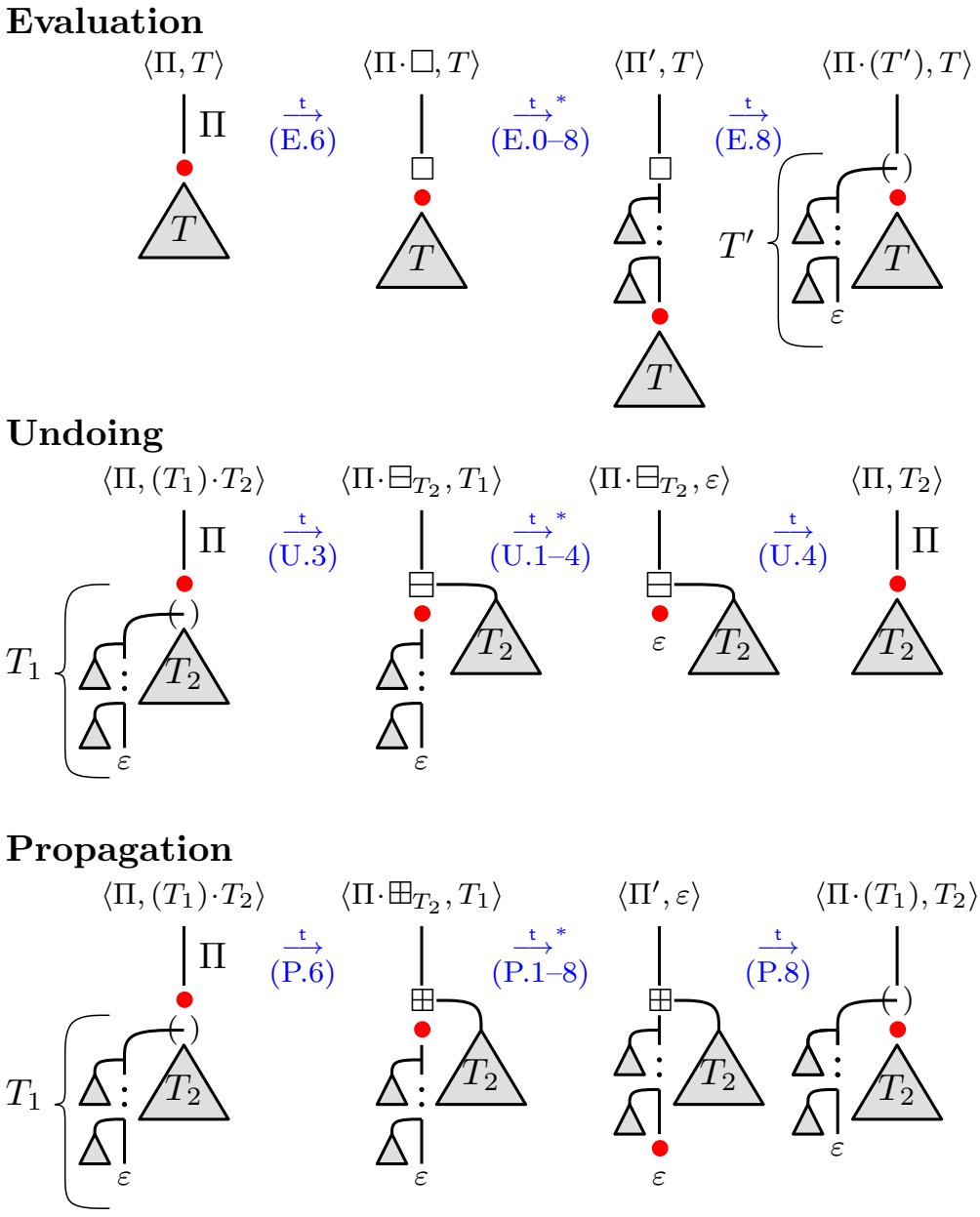

Figure 4.8: Tracing transition modes, across push actions.

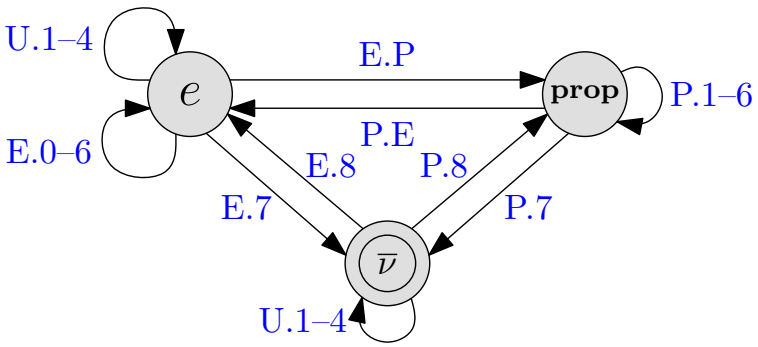

Figure 4.9: Tracing machine: commands and transitions.

Trace contexts and the trace zipper. As described above, our traces are not strictly sequential structures: they also consist of nested subtraces created by **push**. This fact poses a technical challenge for transition semantics (and by extension, an implementation). For instance, while generating such a subtrace, how should we maintain the context of the trace that will eventually enclose it?

To address this, the machine augments the trace with a *context* (Figure 4.7), maintaining in each configuration both a *reuse trace* T, which we say is *in focus*, as well as an unfocused trace context Π. The trace context effectively records a path from the focus back to the start of the trace. To move the focus in a consistent manner, the machine places additional markings $\square$, $\boxminus$, $\boxplus$ into the context; two of these markings (viz. $\boxminus$, $\boxplus$) also carry a subtrace. We describe these markings and their subtraces in more detail below.

This pair of components $\langle \Pi, \mathsf{T} \rangle$ forms a kind of *trace zipper*. More generally, a zipper augments a data structure with a *focus* (for zipper $\langle \Pi, \mathsf{T} \rangle$, we say that T is *in focus*), the ability to perform local edits at the focus and the ability to move this focus throughout the structure (Huet, 1997; Abbott et al., 2004). A particularly attractive feature of zippers is that the "edits" can be performed in a non-destructive, incremental fashion.

To characterize focus movement using trace zippers, we define the *transition modes* of the tracing machine:

- **Evaluation** mirrors the transitions of the reference machine and generates new trace actions, placing them behind the focus, i.e., $\langle \Pi, T \rangle$ becomes $\langle \Pi \cdot t, T \rangle$.
- **Propagation** replays the actions of the reuse trace; it moves the focus through it action by action, i.e., $\langle \Pi, t \cdot T \rangle$ becomes $\langle \Pi \cdot t, T \rangle$.
- **Undoing** removes actions from the reuse trace, just ahead of the focus, i.e., $\langle \Pi, t \cdot T \rangle$ becomes $\langle \Pi, T \rangle$.

If we ignore **push** actions and their nested subtraces (T), the tracing machine moves the focus in the manner just described, either generating, undoing or propagating at most one trace action for each machine transition. However, since **push** actions consist of an entire subtrace T, the machine cannot generate, undo or propagate them in a single step. Rather, the machine must make a series of transitions, possibly interleaving transition modes. When this process completes and the machine moves its focus out of the subtrace, it is crucial that it does so in a manner consistent with its mode upon entering the subtrace. To this end, the machine may extend the context Π with one of three possible markings, each corresponding to a mode.

For each transition mode, Figure 4.8 gives both syntactic and pictorial representations of the focused traces and illustrates how the machine moves its focus. The transitions are labeled with corresponding (blue) transition rules from the tracing machine, but at this time the reader can ignore them. For each configuration, the (initial) trace context is illustrated with a vertical line, the focus is represented by a (red) filled circle and the (initial) reuse trace is represented by a tree-shaped structure that hangs below the focus.

Evaluation (Figure 4.10). To generate a new subtrace in evaluation mode (via a **push**), the machine extends the context Π to $\Pi \cdot \Box$; this effectively marks the beginning of the new subtrace. The machine then performs evaluation transitions that extend the context, perhaps recursively generating nested subtraces in the process (drawn as smaller, unlabeled

triangles hanging to the left). After evaluating the **pop** matching the initial **push**, the machine *rewinds* the current context Π', moving the focus back to the mark $\square$, gathering actions and building a completed subtrace T'; it replaces the mark with a push action (T') (consisting of the completed subtrace), and it keeps reuse trace T in focus. We specify how this rewinding works in Section 4.3; intuitively, it simply moves the focus backwards, towards the start of the trace.

Undoing (Figure 4.12). To undo a subtrace T_1 of the reuse trace $(T_1) \cdot T_2$, the machine extends the context Π to $\Pi \cdot \boxminus_{T_2}$; this effectively saves the remaining reuse trace T_2 for either further undo transitions or for eventual reuse. Assuming that the machine undoes all of T_1, it will eventually focus on an empty trace ε. In this case, the machine can move the saved subtrace T_2 into focus (again, for either further undo transitions or for reuse).

Propagation (Figure 4.11). Finally, to propagate a subtrace T_1, the machine uses an approach similar to undoing: it saves the remaining trace T_2 in the context using a distinguished mark $\boxplus_{T_2}$, moves the focus to the end of T_1 and eventually places T_2 into focus. In contrast to the undo transitions, however, propagation transitions do not discard the reuse trace, but only move the focus by moving trace actions from the reuse trace into the trace context. Just as in evaluation mode, in propagation mode we rewind these actions from the context and move the focus back to the propagation mark ($\boxplus$).

We note that while our semantics characterizes change propagation using a step-by-step replay of the trace, this does not yield an efficient algorithm. In Chapter 6, we give an efficient implementation that is faithful to this replay semantics, but in which the change propagation transitions have zero cost.

Tracing machine transitions. We use the components and transitions of the reference machine (Sections 4.3 and 4.3, respectively) as a basis for defining the transitions of the

Evaluation

E.0 $\langle \Pi, T\rangle, \sigma, \kappa, \rho, e^{u} \xrightarrow{t} \langle \Pi, T\rangle, \sigma, \kappa, \rho', e$
when $\sigma, \kappa, \rho, e^{u} \xrightarrow{r} \sigma, \kappa, \rho', e$

E.1 $\langle \Pi, T\rangle, \sigma, \kappa, \rho, \textbf{let } x = \textbf{alloc}(y) \textbf{ in } e \xrightarrow{t} \langle \Pi \cdot A_{\ell,\rho(y)}, T\rangle, \sigma', \kappa, \rho[x \mapsto \ell], e$
when $\sigma, \rho, \textbf{alloc}(y) \xrightarrow{s} \sigma', \ell$

E.2 $\langle \Pi, T\rangle, \sigma, \kappa, \rho, \textbf{let } x = \textbf{read}(y[z]) \textbf{ in } e \xrightarrow{t} \langle \Pi \cdot R^{v}_{\rho(y)[\rho(z)]}, T\rangle, \sigma, \kappa, \rho[x \mapsto v], e$
when $\sigma, \rho, \textbf{read}(y[z]) \xrightarrow{s} \sigma, v$

E.3 $\langle \Pi, T\rangle, \sigma, \kappa, \rho, \textbf{let } \_ = \textbf{write}(x[y],z) \textbf{ in } e \xrightarrow{t} \langle \Pi \cdot W^{\rho(z)}_{\rho(x)[\rho(y)]}, T\rangle, \sigma', \kappa, \rho, e$
when $\sigma, \rho, \textbf{write}(x[y],z) \xrightarrow{s} \sigma', 0$

E.4 $\langle \Pi, T\rangle, \sigma, \kappa, \rho, \textbf{memo } e \xrightarrow{t} \langle \Pi \cdot M_{\rho,e}, T\rangle, \sigma, \kappa, \rho, e$

E.5 $\langle \Pi, T\rangle, \sigma, \kappa, \rho, \textbf{update } e \xrightarrow{t} \langle \Pi \cdot U_{\rho,e}, T\rangle, \sigma, \kappa, \rho, e$

E.6 $\langle \Pi, T\rangle, \sigma, \kappa, \rho, \textbf{push } f \textbf{ do } e \xrightarrow{t} \langle \Pi \cdot \Box, T\rangle, \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \rho, e$

E.7 $\langle \Pi, T\rangle, \sigma, \kappa, \rho, \textbf{pop } \overline{x} \xrightarrow{t} \langle \Pi \cdot \overline{v}, T\rangle, \sigma, \kappa, \varepsilon, \overline{v}$
when $\overline{v} = \rho(x_i)_{i=1}^{|\overline{x}|}$

E.8 $\langle \Pi, T_2\rangle, \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \varepsilon, \overline{v} \xrightarrow{t} \langle \Pi' \cdot (T_1), T_2'\rangle, \sigma, \kappa, \rho', e$
when $\langle \Pi, T_2\rangle ; \varepsilon \circlearrowleft^{*} \langle \Pi' \cdot \Box, T_2'\rangle ; T_1$
and $\rho(f) = \textbf{fun } f\textbf{(}\overline{x}\textbf{)}.e$
and $\rho' = \rho[x_i \mapsto v_i]_{i=1}^{|\overline{x}|}$

Figure 4.10: Stepping relation for tracing machine ($\xrightarrow{t}$): Evaluation.

tracing machine. You may recall from Section 4.3 that the tracing machine extends the reference machine in two important ways.

First, the machine configurations of the tracing machine extend the reference configurations with an extra component $\langle \Pi, T\rangle$, the trace zipper (Section 4.3), which augments the trace structure T (Section 4.3) with a trace context and a movable focus.

Second, a tracing command α_t consists of either a reference command α_r or the additional propagation command **prop**, which indicates that the machine is doing change propagation. Using these two extensions of the reference machine, the tracing machine generates traces of execution (during evaluation transitions), discards parts of previously-generated traces (during undoing transitions), and reuses previously-generated traces

Reevaluation and reuse

P.E $\langle \Pi, \mathsf{U}_{\rho,e} \cdot \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop} \xrightarrow{t} \langle \Pi \cdot \mathsf{U}_{\rho,e}, \mathsf{T} \rangle, \sigma, \kappa, \rho, e$

E.P $\langle \Pi, \mathsf{M}_{\rho,e} \cdot \mathsf{T} \rangle, \sigma, \kappa, \rho, \textbf{memo}\ e \xrightarrow{t} \langle \Pi \cdot \mathsf{M}_{\rho,e}, \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop}$

Propagation

P.1 $\langle \Pi, \mathsf{A}_{\ell,n} \cdot \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop} \xrightarrow{t} \langle \Pi \cdot \mathsf{A}_{\ell,n}, \mathsf{T} \rangle, \sigma', \kappa, \varepsilon, \textbf{prop}$
when $\sigma, \varepsilon, \textbf{alloc}(n) \xrightarrow{s} \sigma', \ell$

P.2 $\langle \Pi, \mathsf{R}^{\nu}_{\ell[n]} \cdot \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop} \xrightarrow{t} \langle \Pi \cdot \mathsf{R}^{\nu}_{\ell[n]}, \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop}$
when $\sigma, \varepsilon, \textbf{read}(\ell[n]) \xrightarrow{s} \sigma, \nu$

P.3 $\langle \Pi, \mathsf{W}^{\nu}_{\ell[n]} \cdot \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop} \xrightarrow{t} \langle \Pi \cdot \mathsf{W}^{\nu}_{\ell[n]}, \mathsf{T} \rangle, \sigma', \kappa, \varepsilon, \textbf{prop}$
when $\sigma, \varepsilon, \textbf{write}(\ell[n], \nu) \xrightarrow{s} \sigma', 0$

P.4 $\langle \Pi, \mathsf{M}_{\rho,e} \cdot \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop} \xrightarrow{t} \langle \Pi \cdot \mathsf{M}_{\rho,e}, \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop}$

P.5 $\langle \Pi, \mathsf{U}_{\rho,e} \cdot \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop} \xrightarrow{t} \langle \Pi \cdot \mathsf{U}_{\rho,e}, \mathsf{T} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop}$

P.6 $\langle \Pi, (\mathsf{T}_1) \cdot \mathsf{T}_2 \rangle, \sigma, \kappa, \varepsilon, \textbf{prop} \xrightarrow{t} \langle \Pi \cdot \boxplus_{\mathsf{T}_2}, \mathsf{T}_1 \rangle, \sigma, \kappa, \varepsilon, \textbf{prop}$

P.7 $\langle \Pi, \overline{\nu} \rangle, \sigma, \kappa, \varepsilon, \textbf{prop} \xrightarrow{t} \langle \Pi \cdot \overline{\nu}, \varepsilon \rangle, \sigma, \kappa, \varepsilon, \overline{\nu}$

P.8 $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \varepsilon, \overline{\nu} \xrightarrow{t} \langle \Pi' \cdot (\mathsf{T}_1), \mathsf{T}_2 \rangle, \sigma, \kappa, \varepsilon, \textbf{prop}$
when $\langle \Pi, \varepsilon \rangle ; \varepsilon \circlearrowleft^* \langle \Pi' \cdot \boxplus_{\mathsf{T}_2}, \varepsilon \rangle ; \mathsf{T}_1$

Figure 4.11: Stepping relation for tracing machine ($\xrightarrow{t}$): Revaluation and propagation.

Undoing

U.1 $\langle \Pi, \mathsf{A}_{\ell,n} \cdot \mathsf{T} \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t} \langle \Pi, \mathsf{T} \rangle, \sigma[\ell \mapsto \diamond], \kappa, \rho, \alpha_r$

U.2 $\langle \Pi, t \cdot \mathsf{T} \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t} \langle \Pi, \mathsf{T} \rangle, \sigma, \kappa, \rho, \alpha_r$
when $(t = \mathsf{R}^{-}_{-[-]} \mid \mathsf{W}^{-}_{-[-]} \mid \mathsf{M}_{-,-} \mid \mathsf{U}_{-,-} \mid \overline{\nu})$

U.3 $\langle \Pi, (\mathsf{T}_1) \cdot \mathsf{T}_2 \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t} \langle \Pi \cdot \boxminus_{\mathsf{T}_2}, \mathsf{T}_1 \rangle, \sigma, \kappa, \rho, \alpha_r$

U.4 $\langle \Pi \cdot \boxminus_{\mathsf{T}}, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t} \langle \Pi, \mathsf{T} \rangle, \sigma, \kappa, \rho, \alpha_r$

Figure 4.12: Stepping relation for tracing machine ($\xrightarrow{t}$): Undoing the trace.

(during propagation transitions).

These three transition modes (evaluation, undoing and propagation) can interact in ways that are not straightforward. Figure 4.9 helps illustrate their interrelationships, giving us a guide for the transition rules of the tracing machine. The arcs indicate the machine command before and after the machine applies the indicated transition rule (written in blue). Figures 4.10, 4.11 and 4.12 give the complete transition relation for the tracing machine in three parts: Evaluation rules (Figure 4.10) and reevaluation and change propagation rules (Figure 4.11) and undoing rules (Figure 4.12). Recall that each transition is of the form:

$$\langle \Pi, T \rangle, \sigma, \kappa, \rho, \alpha_t \xrightarrow{t} \langle \Pi', T' \rangle, \sigma', \kappa', \rho', \alpha_t{}'$$

We explain Figures 4.10, 4.11 and 4.12 using Figure 4.9 as a guide. Under an expression command e, the machine can take both evaluation (**E.0–6**) and undo (**U.1–4**) transitions while remaining in evaluation mode, as well as transitions **E.P** and **E.7**, which each change to another command. Under the propagation command **prop**, the machine can take propagation transitions (**P.1–6**) while remaining in propagation mode, as well as transitions **P.E** and **P.7**, which each change to another command.

Propagation can transition into evaluation (**P.E**) when it's focused on an **update** action that it (non-deterministically) chooses to activate; it may also (non-deterministically) choose to ignore this opportunity and continue propagation. Dually, evaluation can transition directly into propagation (**E.P**) when its command is a **memo** point that matches a **memo** point currently focused in the reuse trace (and in particular, the environment ρ must also match); it may also (non-deterministically) choose to ignore this opportunity and continue evaluation. We describe a deterministic algorithms for change propagation and memoization in Chapter 6.

Evaluation (respectively, propagation) transitions into a value sequence $\overline{v}$ after evaluating (respectively, propagating) a **pop** operation under **E.7** (respectively, **P.7**). Under the

value sequence command, the machine can continue to undo the reuse trace (**U.1–4**). To change commands, it rewinds its trace context and either resumes evaluation (**E.8**) upon finding the mark $\square$, or resumes propagation (**P.8**) upon finding the mark $\boxplus$. The machine rewinds the trace using the following *trace rewinding* relation:

$$
\begin{array}{lcl}
\langle \Pi \cdot t, T \rangle; T' & \circlearrowleft & \langle \Pi, T \rangle; t \cdot T' \\
\langle \Pi \cdot \boxminus_{T_2}, \varepsilon \rangle; T' & \circlearrowleft & \langle \Pi, T_2 \rangle; T' \\
\langle \Pi \cdot \boxminus_{T_2}, t \cdot T_1 \rangle; T' & \circlearrowleft & \langle \Pi, (t \cdot T_1) \cdot T_2 \rangle; T'
\end{array}
$$

This relation simultaneously performs two functions. First, it moves the focus backwards across actions (towards the start of the trace) while moving these actions into a new subtrace T'; the first case captures this behavior. Second, when moving past a leftover undo mark $\boxminus_{T_2}$, it moves the subtrace T_2 back into the reuse trace; the second and third cases capture this behavior. Note that unlike $\boxminus$, there is no way to rewind beyond either $\square$ or $\boxplus$ marks. This is intentional: rewinding is meant to stop when it encounters either of these marks.

4.4 Consistency

In this section we formalize a notion of consistency between the reference machine and tracing machine. As a first step, we show that when run from scratch (without a reuse trace), the results of the tracing machine are consistent with the reference machine, i.e., the final machine values and stores coincide. To extend this property beyond from-scratch runs, it is necessary to make an additional assumption: we require each IL program run in the tracing machine to be *compositionally store agnostic* (CSA, see below). We then show that, for CSA IL programs, the tracing machine reuses computations in a consistent way: its final trace, store, and machine values are consistent with a from-scratch run of the tracing machine, and hence, they are consistent with a run of the reference machine.

Finally, we discuss some interesting invariants of the tracing machine (Section 4.4) that play a crucial role in the consistency proof.

Compositional store agnosticism (CSA). The property of compositional store agnosticism characterizes the programs for which our tracing machine runs consistently. We build this property from a less general property that we call *store agnosticism*. Intuitively, an IL program is store agnostic iff, whenever an **update** instruction is performed during its execution, then the value sequence that will eventually be popped is already determined at this point and, moreover, independent of the current store.

Definition 4.4.1. Formally, we define $\mathsf{SA}(\sigma, \rho, e)$ to mean:
If $\sigma, \epsilon, \rho, e \xrightarrow{r}{}^{*} \_, \_, \rho', \textbf{update}\ e'$, then there exists $\overline{v}$ such that $\overline{w} = \overline{v}$ whenever $\_, \epsilon, \rho', e' \xrightarrow{r}{}^{*} \_, \epsilon, \epsilon, \overline{w}$.

To see why this property is significant, recall how the tracing machine deals with intermediate results. In stepping rule **E.8**, the tracing machine mirrors the reference machine: it passes the results to the function on the top of the control stack. However, in stepping rule **P.8**, the tracing machine does *not* mirror the reference machine: it essentially discards the intermediate results and continues to process the remaining reuse trace. This behavior is not generally consistent with the reference machine: If **P.8** is executed after switching to evaluation mode (**P.E**) and performing some computation in order to adjust to a modified store, then the corresponding intermediate result may be different. However, if the subprogram that generated the reuse trace was store agnostic, then this new result will be the same as the original one; consequently, it is then safe to continue processing the remaining reuse trace.

Compositional store agnosticism is a generalization of store agnosticism that is preserved by execution.

Definition 4.4.2. We define $\mathsf{CSA}(\sigma, \rho, e)$ to mean:
If $\sigma, \varepsilon, \rho, e \xrightarrow{\mathsf{r}}{}^{*} \sigma', \kappa, \rho', e'$, then $\mathsf{SA}(\sigma', \rho', e')$.

Lemma 4.4.1. *If* $\sigma, \varepsilon, \rho, e \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', e'$ *and* $\mathsf{CSA}(\sigma, \rho, e)$*, then* $\mathsf{CSA}(\sigma', \rho', e')$.

Consistency of the tracing machine. The first correctness property says that, when run from scratch (i.e. without a reuse trace), the tracing machine mirrors the reference machine.

Theorem 4.4.2 (Consistency of from-scratch runs)**.**
If $\langle \varepsilon, \varepsilon \rangle, \sigma, \varepsilon, \rho, \alpha_{\mathsf{r}} \xrightarrow{t}{}^{*} \langle \_, \_ \rangle, \sigma', \varepsilon, \varepsilon, \overline{\nu}$
then $\sigma, \varepsilon, \rho, \alpha_{\mathsf{r}} \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \overline{\nu}$.

In the general case, the tracing machine does not run from scratch, but with a reuse trace generated by a from-scratch run. To aid readability for such executions we introduce some notation. We call a machine reduction *balanced* if the initial and final stacks are each empty, and the initial and final trace contexts are related by the trace rewinding relation. If know that the stack and trace context components of a machine reduction meet this criteria, we can specify this (balanced) reduction more concisely.

Definition 4.4.3 (Balanced reductions)**.**

$$\frac{\langle \varepsilon, \mathsf{T} \rangle, \sigma, \epsilon, \rho, \alpha_{\mathsf{r}} \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma', \epsilon, \epsilon, \overline{\nu} \qquad \langle \Pi, \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \varepsilon, \varepsilon \rangle; \mathsf{T}'}{\mathsf{T}, \sigma, \rho, \alpha_{\mathsf{r}} \Downarrow \mathsf{T}', \sigma', \overline{\nu}}$$

$$\frac{\langle \varepsilon, \mathsf{T} \rangle, \sigma, \epsilon, \epsilon, \mathbf{prop} \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma', \epsilon, \epsilon, \overline{\nu} \qquad \langle \Pi, \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \varepsilon, \varepsilon \rangle; \mathsf{T}'}{\mathsf{T}, \sigma \curvearrowright \mathsf{T}', \sigma', \overline{\nu}}$$

We now state our second correctness result. It uses an auxiliary function that collects garbage: $\sigma|_{\mathsf{gc}}(\ell) = \sigma(\ell)$ for $\ell \in \mathsf{dom}(\sigma|_{\mathsf{gc}}) = \{\ell \mid \ell \in \mathsf{dom}(\sigma) \text{ and } \sigma(\ell) \neq \diamond\}$.

Theorem 4.4.3 (Consistency)**.**
Suppose $\varepsilon, \sigma_1, \rho_1, \alpha_{\mathsf{r}1} \Downarrow \mathsf{T}_1, \sigma_1', \overline{\nu}_1$ *and* $\mathsf{CSA}(\sigma_1, \rho_1, \alpha_{\mathsf{r}1})$.

1. *If* $T_1, \sigma_2, \rho_2, \alpha_{r2} \Downarrow T_1', \sigma_2', \overline{v}_2$
   then $\varepsilon, \sigma_2|_{gc}, \rho_2, \alpha_{r2} \Downarrow T_1', \sigma_2'|_{gc}, \overline{v}_2$

2. *If* $T_1, \sigma_2 \curvearrowright T_1', \sigma_2', \overline{v}_2$
   then $\varepsilon, \sigma_2|_{gc}, \rho_1, \alpha_{r1} \Downarrow T_1', \sigma_2'|_{gc}, \overline{v}_2$

The first statement says that, when run with an arbitrary from-scratch generated trace T_1, the tracing machine produces a final trace, store and return value sequence that are consistent with a from-scratch run of the same program. The second statement is analogous, except that it concerns change propagation: when run over an arbitrary from-scratch generated trace T_1, the machine produces a result consistent with a from-scratch run of the program that generated T_1. Note that in each case the initial store may be totally different from the one used to generate T_1.

Finally, observe how each part of Theorem 4.4.3 can be composed with Theorem 4.4.2 to obtain a corresponding run of the reference machine.

Garbage collection. The tracing machine may undo portions of the reuse trace in order to adjust it to a new store. Whenever it undoes an allocation (rule **U.1**), it marks the corresponding location as garbage ($\ell \mapsto \diamond$).

In order for this to make sense we better be sure that these locations are not live in the final result, i.e., they neither appear in T_1' nor $\overline{v}_2$ nor are referenced from the live portion of σ_2'. In fact, this is a consequence of the consistency theorem: the from-scratch run in the conclusion produces the same T_1' and $\overline{v}_2$. Moreover, since its final store is $\sigma_2'|_{gc}$, it is clear that these components and $\sigma_2'|_{gc}$ itself cannot refer to garbage.

Invariants. The proof of Theorem 4.4.3 is by induction on the length of the given from-scratch run producing T_1. It requires numerous lemmas and, moreover, the theorem statement needs to be strengthened in several ways. In the remainder of this section, we explain

the main generalizations as they expose invariants of the tracing machine that are crucial for its correct functioning[3]. Full details of this and all other proofs mentioned later on can be found in the accompanying technical appendix.

Non-empty trace context and stack. Neither the trace context nor the stack will stay empty during execution, so we need to account for that. In part 2 of the generalized version of the theorem we therefore assume the following about the given from-scratch run (see below for part 1):

a) $\mathsf{CSA}(\sigma_1, \rho_1, \alpha_{r1})$

b) $\langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{t}{}^{*} \langle \Pi_1', \varepsilon \rangle, \sigma_1', \kappa_1, \varepsilon, \overline{v_1}$

c) $\langle \Pi_1', \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \Pi_1, \varepsilon \rangle; T_1$

d) Π_1 contains neither undo (⊟) nor propagation (⊞) marks

When these conditions are all met, we say T_1 fsc ("from-scratch consistent"). Condition (a) is the same as in the theorem statement. Conditions (b) and (c) are similar to the assumptions stated in the theorem, except more general: they allow a non-empty trace context and a non-empty stack. The new condition (d), ensures that the trace context mentioned in (b) and (c) only describes past evaluation steps, and not past or pending undoing or propagating steps. Apart from the assumption, we also must generalize the rest of part 2 accordingly but we omit the details here.

Reuse trace invariants. While it is intuitively clear that propagation (part 2) must run with a from-scratch generated trace in order to generate one, this is not strictly necessary for evaluation (part 1). In fact, here the property T_1 fsc is not always preserved: Recall

3. To our knowledge, this is the first work that characterizes the entire trace (both in and out of focus), in the midst of adjustment. Such characterizations may be useful, for example, to verify efficient implementations of the tracing machine.

that in evaluation mode the machine may undo steps in T_1. Doing so may lead to a reuse trace that is no longer from-scratch generated! In particular, if $T_1 = (t \cdot T_2) \cdot T_3$, then, using steps **U.3**, **U.2** and eventually **E.8**, the machine may essentially transform this into $(T_2) \cdot T_3$, which in general may not have a corresponding from-scratch run.

In order for the induction to go through, we therefore introduce a weaker property, T_1 ok, for part 1. It is defined as follows:

$$\frac{}{\varepsilon \text{ ok}} \qquad \frac{T \text{ fsc}}{T \text{ ok}} \qquad \frac{T \text{ ok} \qquad T' \text{ ok}}{(T) \cdot T' \text{ ok}}$$

Note that if $T_1 = M_{\rho,e} \cdot T_2$ and T_1 ok, then T_1 fsc (and thus T_2 fsc) follows by inversion. This comes up in the proof precisely when in part 1 evaluation switches to propagation (step **E.P**) and we therefore want to apply the inductive hypothesis of part 2, where we need to know that the new reuse trace is fsc (not "just" ok).

Trace context invariant. In order for T_1 ok and T_1 fsc to be preserved by steps **U.4** and **P.8**, respectively, we also require Π_1 ok, defined as follows:

$$\frac{}{\varepsilon \text{ ok}} \qquad \frac{\Pi \text{ ok}}{\Pi \cdot t \text{ ok}} \qquad \frac{\Pi \text{ ok}}{\Pi \cdot \square \text{ ok}}$$

$$\frac{\Pi \text{ ok} \qquad T \text{ fsc}}{\Pi \cdot \boxplus_T \text{ ok}} \qquad \frac{\Pi \text{ ok} \qquad T \text{ ok}}{\Pi \cdot \boxminus_T \text{ ok}}$$

Note the different assumptions about T in the last two rules. This corresponds exactly to the different assumptions about T_1 in part 1 and part 2.

$$
\begin{array}{lcl}
[\![\textbf{let fun } f(\overline{x}).e_1 \textbf{ in } e_2]\!]_y & = & \textbf{let fun } f(\overline{x} \mathbin{@} z).[\![e_1]\!]_z \textbf{ in } [\![e_2]\!]_y \\
[\![\textbf{let } x = \oplus(\overline{y}) \textbf{ in } e]\!]_y & = & \textbf{let } x = \oplus(\overline{y}) \textbf{ in } [\![e]\!]_y \\
[\![\textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2]\!]_y & = & \textbf{if } x \textbf{ then } [\![e_1]\!]_y \textbf{ else } [\![e_2]\!]_y \\
[\![f\ (\overline{x})]\!]_y & = & f\ (\overline{x} \mathbin{@} y) \\
[\![\textbf{let } x = \iota \textbf{ in } e]\!]_y & = & \textbf{let } x = \iota \textbf{ in } [\![e]\!]_y \\
[\![\textbf{memo } e]\!]_y & = & \textbf{memo } [\![e]\!]_y \\
[\![\textbf{update } e]\!]_y & = & \textbf{update } [\![e]\!]_y \\
[\![\textbf{push } f \textbf{ do } e]\!]_y & = & \textbf{let fun } f'(z).\ \textbf{update} \\
\quad \text{when } \mathsf{Arity}(f) = n & & \quad\ \textbf{let } x_1 = \textbf{read}(z[1]) \textbf{ in } \cdots \\
& & \quad\ \textbf{let } x_n = \textbf{read}(z[n]) \textbf{ in} \\
& & \quad\ f\ (x_1, \ldots, x_n, y) \\
& & \textbf{in} \\
& & \textbf{push } f' \textbf{ do memo} \\
& & \quad\ \textbf{let } z = \textbf{alloc}(n) \textbf{ in } [\![e]\!]_z \\
[\![\textbf{pop } \overline{x}]\!]_y & = & \textbf{let } \_ = \textbf{write}(y[1], x_1) \textbf{ in } \cdots \\
\quad \text{when } |\overline{x}| = n & & \textbf{let } \_ = \textbf{write}(y[n], x_n) \textbf{ in} \\
& & \textbf{pop } \langle y \rangle \\
[\![\varepsilon]\!] & = & \varepsilon \\
[\![\rho[x \mapsto v]]\!] & = & [\![\rho]\!][x \mapsto v] \\
[\![\rho[f \mapsto \textbf{fun } f(\overline{x}).e]]\!] & = & [\![\rho]\!][f \mapsto \textbf{fun } f(\overline{x} \mathbin{@} y).[\![e]\!]_y]
\end{array}
$$

Figure 4.13: Destination-passing-style (DPS) conversion.

4.5 Destination-Passing Style

In Section 4.4, we defined the CSA property that the tracing machine requires of all programs for consistency. In this section, we describe a *destination-passing-style* transformation and show that it transforms arbitrary IL programs into CSA IL programs, while preserving their semantics. The idea is as follows: A DPS-converted program takes an additional parameter x that acts as its destination. Rather than return its results directly, the program then instead writes them to the memory specified by x.

Figure 4.13 defines the DPS transformation for an expression e and a destination variable x, written $[\![e]\!]_x$. Naturally, to DPS-convert an expression closed by an environment ρ, we must DPS-convert the environment as well, written $[\![\rho]\!]$. In order to comply with our

assumption that all function and variable names are distinct, the conversion actually has to thread through a set of already-used names. For the sake of readability we do not include this here.

Most cases of the conversion are straightforward. The interesting ones include function definition, function application, **push**, and **pop**. For function definitions, the conversion extends the function arguments with an additional destination parameter z (we write $\overline{x}\,@\,z$ to mean $\overline{x}$ appended with z). Correspondingly, for application of a function f, the conversion additionally passes the current destination to f. For **push**es, we allocate a fresh destination z for the **push** body; we memoize this allocation with a **memo** point. When the **push** body terminates, instead of directly passing control to f, the program calls a wrapper function f' that reads the destination and finally passes the values to the actual function f. Since these **read**s may become inconsistent in subsequent runs, we prepend them with an **update** point. For **pop**s, instead of directly returning its result, the converted program writes it to its destination and then returns the latter.

As desired, the transformation yields CSA programs (here and later on we assume that n is the arity of the program being transformed):

Theorem 4.5.1 (DPS programs are CSA)**.**

$\mathsf{CSA}(\sigma, [\![\rho]\!], \textbf{let } x = \textbf{alloc}(n) \textbf{ in } [\![e]\!]_x)$

Moreover, the transformation preserves the extensional semantics of the original program:

Theorem 4.5.2 (DPS preserves extensional semantics)**.**

If $\sigma_1, \varepsilon, \rho, e \xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{v}$

then $\sigma_1, \varepsilon, [\![\rho]\!], \textbf{let } x = \textbf{alloc}(n) \textbf{ in } [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma_1' \uplus \sigma_2', \varepsilon, \varepsilon, \ell$

with $\sigma_2'(\ell, i) = v_i$ *for all* i.

Because it introduces destinations, the transformed program allocates additional store locations σ_2'. These locations are disjoint from the original store σ_1', whose contents are

preserved in the transformed program. If we follow one step of indirection, from the returned location to the values it contains, we recover the original results $\overline{v}$.

A small example: composed function calls. As a simple illustrative example, consider the source-level expression `f(max(*p,*q))`, which applies function `f` to the maximum of two dereferenced pointers `*p` and `*q`. Our front end translates this expression into the following:

```
push f do
  update
    let x = read(p[0]) in
    let y = read(q[0]) in
    if x > y then pop x else pop y
```

Notice that the body of this **push** is not store agnostic—when the memory contents of either pointer is changed, the **update** body can evaluate to a different return value, namely the new maximum of x and y. To address this, the DPS transformation converts this fragment into the following:

```
let fun f′(m). update
  let m′ = read(m[0]) in f(m′, z)
in
push f′ do memo
  let m = alloc(1) in
  update
    let x = read(p[0]) in
    let y = read(q[0]) in
    if x > y then
      let _ = write(m[0],x) in pop m
    else
      let _ = write(m[0],y) in pop m
```

Notice that instead of returning the value of either x or y as before, the body of the **push** now returns the value of m, a pointer to the maximum of x and y. In this case, the

push body is indeed store agnostic—though x and y may change, the pointer value of m remains fixed, since it is defined outside of the **update** body.

The astute reader may wonder why we place the allocation of m within the bodies of the **push** and **memo** point, rather than "lift it" outside the definition of function f'. After all, by lifting it, we would not need to return m to f' via the stack **pop**—the scope of variable m would include that of function f'. We place the allocation of m where we do to promote reuse of nondeterminism: by inserting this **memo** point, the DPS transformation effectively associates local input state (the values of p and q) with the local output state (the value of m). Without this **memo** point, every **push** body will generate a fresh destination each time it is reevaluated, and in general, this nondeterministic choice will prevent reuse of any subcomputation, since this subcomputation's local state includes a distinct, previously chosen destination. To avoid this behavior and to allow these subcomputations to instead be reused during change propagation, the DPS conversion inserts **memo** points that enclose each (non-deterministic) allocation of a destination.

4.6 Cost Models

We define a generic framework for modeling various dynamic costs of our IL abstract machines (both reference and tracing). By instantiating the framework with different concrete cost models, we show several cost equivalences between the IL reference machine and the IL tracing machine (Section 4.3), show that our DPS conversion (Section 4.5) respects the intensional reference semantics of IL up to certain constant factors, and give a cost model for our implementation (Chapters 5 and 6).

Cost Models A *cost model* is a triple $M = \langle C, \mathbf{0}, \gamma \rangle$ where:

- Type C is the type of costs.

- The zero cost $\mathbf{0} \in C$ is the cost of an empty step sequence.
- The cost function $\gamma : S \to C \to C$ assigns to each step $s \in S$, a function that maps the cost before the step s is taken to the cost after s is taken.
- Given an execution sequence $\overline{s} = \langle s_1, \ldots, s_n \rangle$, we define the cost function of $\overline{s}$ under M as the following composition of cost functions:

$$\gamma\ \overline{s} = (\gamma\ s_n) \circ \cdots \circ (\gamma\ s_1)$$

  By assuming zero initial cost, we can evaluate this composition of cost functions to a yield *total cost* for $\overline{s}$ as $\gamma\ \overline{s}\ \mathbf{0} = c \in C$.

Step Counts

- Cost model $M_s = \langle C_s, \mathbf{0}_s, \gamma_s \rangle$ counts total steps taken.
- $C_s = \mathcal{N}$
- $\mathbf{0} = 0$
- $\gamma_s\ s\ n = n + 1$

Store Usage

- Cost model $M_\sigma = \langle C_\sigma, \mathbf{0}_\sigma, \gamma_\sigma \rangle$ measures store usage as the number of allocations (a), reads (r), and writes (w), respectively.
- $C_\sigma = \mathcal{N}^3$
- $\mathbf{0} = \langle 0, 0, 0 \rangle$

- Assuming that step s_{alloc} allocates in the store, that step s_{read} reads from the store and that step s_{write} writes to the store, we define the following cost function:

$$\begin{aligned}
\gamma_\sigma\ s_{\text{alloc}}\ \langle a, r, w\rangle &= \langle a+1, r, w\rangle \\
\gamma_\sigma\ s_{\text{read}}\ \langle a, r, w\rangle &= \langle a, r+1, w\rangle \\
\gamma_\sigma\ s_{\text{write}}\ \langle a, r, w\rangle &= \langle a, r, w+1\rangle \\
\gamma_\sigma\ s_{\text{nostore}}\ \langle a, r, w\rangle &= \langle a, r, w\rangle
\end{aligned}$$

- For the reference machine:

  $s_{\text{alloc}} =$ **R.6**/**S.1**, $s_{\text{read}} =$ **R.6**/**S.2** and $s_{\text{write}} =$ **R.6**/**S.3**.

- For the tracing machine:

  $s_{\text{alloc}} =$ **E.1**, $s_{\text{read}} =$ **E.2** and $s_{\text{write}} =$ **E.3**.

Stack Usage

- Cost model $M_\kappa = \langle C_\kappa, \mathbf{0}_\kappa, \gamma_\kappa\rangle$ measures the stack usage as the number of times the stack is pushed (u) and popped (d), the current stack height (h), and the maximum stack height (m).

- $C_\kappa = \mathcal{N}^4$

- $\mathbf{0}_\kappa = \langle 0, 0, 0, 0\rangle$

- Assuming that step s_{push} pushes the stack, step s_{pop} pops the stack and s_{nostack} does neither, we define the following cost function:

$$\begin{aligned}
\gamma_\kappa\ s_{\text{push}}\ \langle u, d, h, m\rangle &= \langle u+1, d, h+1, \max(m, h+1)\rangle \\
\gamma_\kappa\ s_{\text{pop}}\ \langle u, d, h, m\rangle &= \langle u, d+1, h-1, m\rangle \\
\gamma_\kappa\ s_{\text{nostack}}\ \langle u, d, h, m\rangle &= \langle u, d, h, m\rangle
\end{aligned}$$

- For the reference machine:

  $s_{\text{push}} = \textbf{R.9}$ and $s_{\text{pop}} = \textbf{R.11}$

- For the tracing machine:

  $s_{\text{push}} = \textbf{E.6}$ and $s_{\text{pop}} = \textbf{E.8}$

  Note that the stack is actually popped by **R.11** rather than **R.10**, and **E.8** rather than **E.7**. The latter steps—which each evaluate a **pop** expression to a sequence of closed values—always precede the actual stack pop by one step.

Reference versus Tracing Machines. The following result establishes several cost equivalences between the reference machine and the tracing machine.

Theorem 4.6.1. *Fix an initial machine state* $\sigma, \epsilon, \rho, e$*. Run under the reference machine to yield step sequence* $\bar{s}^u$*. Run under the tracing machine with an empty reuse trace to yield step sequence* $\bar{s}^t$*. The following hold for* $\bar{s}^u$ *and* $\bar{s}^t$*:*

- *The step counts under* M_s *are equal.*
- *The stack usage under* M_σ *is equal.*
- *The store usage under* M_κ *is equal.*

DPS Costs. Recall that our DPS transformation (Section 4.5) ensures that programs are compositionally store agnostic, a property that is needed to ensure the consistency of our tracing machine. Below we bound the overhead introduced by this transformation in the reference machine. However, since the two machines have the same intensional semantics under M_s, M_σ and M_κ (as shown above) the result below applies equally well to the tracing machine too.

Theorem 4.6.2 (Intensional semantics (mostly) preserved)**.**

Consider the evaluations of expression e and $[\![e]\!]_\chi$ as given in Theorem 4.5.2. The following hold for their respective step sequences, $\bar{s}$ and $\bar{s}'$:

- *The stack usage under M_κ is equal. Let d and u be the number of* **push***s and* **pop***s performed in each, respectively.*
- *The number of allocations under M_σ differ by exactly d, the number of reads and writes under M_σ differ by at most $n \cdot d$ and $n \cdot u$, respectively, where n is the maximum arity of any* **pop** *taken in $\bar{s}$.*
- *The number of steps taken under M_s differ by at most $(n+4) \cdot d + n \cdot u$.*

Realized costs. Realized costs closely resemble those of a real implementation. We model them with $M_t = \langle C_t, \mathbf{0}_t, \gamma_t \rangle$, which partitions step counts of the tracing machine into evaluation (e), undo (u) and propagation (p) step counts. As in previous work, our implementation does *not* incur any cost for any propagation steps taken—these steps are effectively skipped. Therefore, we define the *realized* cost of $\langle e, p, u \rangle \in C_t$ as $(e + u) \in \mathcal{N}$. These realized costs are proportional to the actual work performed by IL programs compiled by our implementation (Chapters 5 and 6)[4]. Each cost is a triple:

$$C_t = \mathcal{N}^3$$

$$\mathbf{0}_t = \langle 0, 0, 0 \rangle$$

$$\gamma_t\ s_{\text{eval}}\ \langle e, p, u \rangle = \langle e + 1, p, u \rangle$$

$$\gamma_t\ s_{\text{prop}}\ \langle e, p, u \rangle = \langle e, p + 1, u \rangle$$

$$\gamma_t\ s_{\text{undo}}\ \langle e, p, u \rangle = \langle e, p, u + 1 \rangle$$

(Here s_{eval} matches steps **E.0–8**, s_{prop} matches steps **P.1–8**, and s_{undo} matches steps **U.1–4**.)

4. The implementation cost may involve an additional logarithmic factor, e.g., to maintain a persistent view of the store for every point in the trace.

CHAPTER 5

COMPILATION

This chapter describes a compilation strategy for self-adjusting machines. First, we give an overview of the role of compilation as a bridge from the abstract machine (Chapter 4) to the run-time system (Chapter 6) and outline the reminder of the chapter in greater detail (Section 5.1). We describe how a C-based self-adjusting program consists of multiple levels of control, which our compiler automatically separates (Section 5.2). We give a C-based run-time library interface for constructing execution traces (Section 5.3). We represent programs during compilation using IL and CL (Section 5.4). CL is a representation closely-related to IL that augments programs with additional static control structure. We use static analyses to inform both our basic compilation and our optimizations (Section 5.5). We give a basic translation (from IL to CL) for basic compilation (Section 5.6). We give a graph algorithm for normalization, which statically relocates certain CL blocks as top-level functions (Section 5.7). To improve upon this compilation approach, we describe several optimizations (Section 5.8). Finally, we give further notes and discussion (Section 5.9).

5.1 Overview

We give an overview of our compilation stategy for self-adjusting machines. The role of compilation is to prepare an intermediate langauge program for interoperation with the C runtime library, by translating and transforming it.

Requirements. Some work performed by the compiler is required, and some is optional optimization. First, we outline the requirements:

- *Requirement:* Programs must be compositionally store agnostic (Section 4.4).

This requirement is imposed by our abstract machine semantics, and the run-time system which implements it.

- *Requirement:* Primitives must be translated.

  The self-adjusting primitives **memo** and **update** of IL must be replaced with generated C code that uses the interface provided by the run-time system. Similarly, the traced instructions of IL must be translated into C, replacing these instructions with calls to their C-based implementations.

- *Requirement:* IL expressions must be decomposed into globally-scoped functions.

  Specifically, where we have **update** points, we want to create traces that store thunks (suspended computations). Operationally, these thunks can reenter the program at fine-grained control points that are not programmed as top-level level functions in IL, but must be top-level functions in the target C program (as required by low-level languages; see Section 1.6.4).

Transformations. We address the requirements above by transforming the program. In doing so, we introduce a variant of IL that is somewhat closer to C, which we call CL, for *control langauge*, since it lets us distinguish between local and non-local control flow (Section 5.4). In Section 5.5, we use the additional control-flow structure of CL to statically analyze our programs' dynamic behavior.

Using CL as a target, we transform IL in a number of stages, listed below. At each stage, we note the possibility for optmization (using the analyses mentioned above).

(a) The *destination-passing style (DPS) transformation* yeilds target programs that are provably store agnostic. We refer to Section 4.5 for the basic definition.

(b) *Translation of IL into CL* lowers the self-adjusting primitives **memo** and **update** into their C-based implementations. Similarly, we translate traced instructions into their implementations. We describe this translation in Section 5.6.

(c) The *normalization transformation* of CL yields target programs where certain local code blocks are lifted into a global scope as functions. We describe this transformation in Section 5.7.

Optimizations. Based on the transformations above, we make several refinements to mitigate and amortize the overhead of our approach.

We note that the normalization transformation comes with relatively little direct overhead at run time: it changes some local control flow into non-local control flow (i.e., practically, this is the difference between mutating a stack frame versus replacing it). However, by partitioning local blocks into distinct global functions, normalization reduces statically-known information, which has consequences for later optimizations. Hence, the normalization algorithm is designed to retain local (statically-known) control flow whenever possible.

Using this static information, we refine our translation and transformation with two major optimizations:

- *Optimization:* Selective destination-passing style.

  As described in Section 4.6, the basic DPS transformation comes with certain constant-factor overheads, which can be reduced significantly with some simple static analysis.

- *Optimization:* Trace node layout and sharing.

  In the basic translation, each primitive instance is traced independently, and its associated run-time overhead is not amortized across consecutive instances. We refine this with an optimization that shares trace resources among the primitive instances.

5.2 Multiple levels

In Chapter 4 we study self-adjusting machines in isolation. In practice, however, a self-adjusting machine does not exist in a vaccum: it is enclosed by an *outer program*, variously referred to as the *meta level* or *meta program* in past literature. Hence, a fuller, more practical picture of self-adjusting machines zooms back one step, and accounts for these machines within the context of an enclosing environment that has code and data of its own. Hence, there are (at least) two levels to account for: the outer (non-self-adjusting) program, and the inner (self-adjusting) program.

In this section we describe these levels from the viewpoint of compilation and run-time implementation. We describe how distinct levels can be viewed as distinct agents of control, whose interaction must observe certain restrictions (Section 5.2). We describe how we implement each agent via coordinated compilation and run-time techniques (Section 5.2).

Multiple agents of control. Past approaches to self-adjusting computation impose the dichotomy of the *meta* versus *core* on code and data (Hammer et al., 2009). Meta-level programs are those programs that construct and update self-adjusting computations; core-level programs are the programs whose traces are constructed and updated by change propagation. This terminology extends from code to data: data allocated within the execution of core code is *core data*, and data allocated outside this execution is *meta data*. In the context of a self-adjusting machine, we use the terms *outer* and *inner*, as opposed to *meta* and *core*. The *outer program* (meta program) constructs and updates a self-adjusting machine via change propagation; in turn, the self-adjusting machine executes an *inner program* (core program), furnished by the outer program.

The *inner data* consists of all data allocated during the execution of the inner program. The *outer data* consists of all global data shared by the outer and inner programs, as well as all data dynamically allocated by the outer program, and all data that participates in

output-to-input feedback within the inner program. We return to these definitions when we define the run-time system, in Chapter 6.

In addition to the outer and inner code, the CEAL compiler also accepts interleaving *foreign C code* with CEAL code (Section 3.9). Since it does not require compilation, this foreign C code is neither inner nor outer code. However, the foreign code can be called by both inner and outer code, and can itself call outer code. After compilation, both the inner and outer code become C code that use the run-time interface. Specifically, they use this interface to construct self-adjusting machines and interact with their modifiable memory. While doing this manually within CEAL via foreign C code is possible, compilation avoids the burdens of doing so; it also allows the run-time system to reasonably demand that additional static information be extracted and organized by the compiler for improved run-time efficiently.

Compilation and run-time via separation. Our compiler seperates the levels described above. A CEAL program consists of declarations of static information (viz. type definitions, function prototypes and function definitions) as well as declarations of global storage (viz. global variables and hidden library state). Unless explicitly marked as foreign C code, all global storage is outer data[1]. Depending on a function's calling context, many CEAL functions can be interpreted as either inner code, or by ignoring certain keywords such as **memo**, as outer code. When functions are used in multiple contexts, the compiler duplicates and specializes them. In doing so, it strips out self-adjusting primitives (**memo** and **update**) when functions are duplicated and used outside of the inner code.

To perform this separation and duplication, the compiler first performs an analysis that computes three callgraphs of the source code, rooted at three distinct sets of entry points:

1. The header files of standard C libraries sometimes declare and use global state. According to its configuration settings, our compiler distinguishes these header files containing foreign C code and data from those that contain CEAL code and data.

- For foreign C code, it roots the graph such that every public (non-`static`) foreign C function is a potential entry point.
- For outer code, it roots the graph such that every public (non-`static`) function is a potential entry point.
- For inner code, it roots the graph such that the code block for each self-adjusting machine is a potential entry point.

As mentioned above, the caller-callee relationships of the multi-level program have additional structure. Foreign code cannot directly call inner code, but it can directly call outer code. In turn, outer code calls inner code each time it constructs a self-adjusting machine. The outer code indirectly interacts with the inner code through shared memory structures and change propagation. Both the outer and inner code can call foreign code.

In this chapter, we focus entirely on compiling the inner code, which is the most interesting and complex of the different levels, from the viewpoint of compilation. The outer programs require only a light translation to replace their use of certain operations with appropriate calls to the run-time system; since they lack **memo** and **update** points, this translation is entirely straightforward (Section 5.3). After this translation, the outer program is equivalent to foreign C code. The foreign C code does not require any special steps: It can be compiled directly by a standard C compiler.

The callgraph analysis is simplified by the assumption that the program is first-order, which our implementation currently imposes. We do not foresee that support for function pointers would be overly complex. Due to the uncertainty of statically predicting which are required, higher-order functions will generally require that all possible versions be generated for each function. When indirect calls (higher-order function applications) do not preserve the caller's level compared with that of the code being called, the programmer would supply annotations to explicitly switch levels; for instance, from the inner level to

the foreign level. This level-switching can be implemented with as a projection from a record that consists of all versions of the function (Chen et al., 2011, 2012).

5.3 Run-time interface

By translating IL programs to CL programs that use the run-time library interface, the compiler generates code that constructs execution traces. We discuss central concepts in the design of this interface: trace hook closures (Section 5.3), trace node attributes (Section 5.3), trace nodes (Section 5.3) and in particular, their static descriptors (Section 5.3).

Trace hook closures. A *trace hook closure* (THC) is a record of a traced operation that contains data and code used by the self-adjusting machine. Specifically, these records store the trace information necessary for the machine to redo or undo the operation during change propagation. In this section we discuss the general signature of a trace hook closure, which is implemented by the run-time system library for each traced instruction in IL.

THCs provide a general framework for defining and extending self-adjusting machines. We express both built-in primitives as well as library-provided extensions as THCs, including the allocation, reading and writing of modifiable memory ((Section B) gives listings implementing modifiable references, in terms of the implementations' THCs). In a sense, the role of the compiler is to combine all the THCs of a program by combining their component parts, and use these THC instances to customize a general-purpose run-time library implemention of the self-adjusting machine semantics.

Figure 5.1 gives the signature of a trace hook closure for a hypothetical operation that has one argument of type `arg_t` and the return type `ret_t`. This pattern can easily be generalized to multiple arguments, variable-length arguments and so on. We explain the THC components of this operation below.

Figure 5.1: The TRACE_HOOK_CLOSURE signature.

```
#module_begin TRACE_HOOK_CLOSURE
#imports
  open [ TRND_ATTRIBUTES ] -- trace node attributes
  type trnd_t               -- trace node type
#exports
  val atts : Atts.t        -- attributes required of hooks below
  type clos_t              -- closure type for closing hooks below

  val foreign :                    arg_t -> ret_t  -- outside of machine
  val invoke  : trnd_t*, clos_t*,  arg_t -> ret_t  -- within the machine
  val consis  : trnd_t*, clos_t*         -> bool_t -- callback: consistency
  val revoke  : trnd_t*, clos_t*         -> void   -- callback: undo
  val revinv  : trnd_t*, clos_t*,  arg_t -> ret_t  -- callback: redo
#module_end
```

The module that implements a trace hook closure consists of a statically-fixed attribute set (`atts`), which we discuss further below. The rest of the module consists of a type for closure records (`clos_t`), a function that gives the foreign behavior (`foreign`), and a number of "hooks" that work in concert to give the machine behavior (`invoke`, `consis`, `revoke` and `revinv`).

Most of the hooks act as callbacks: they are installed into the self-adjusting machine to collectively provide the self-adjusting behavior of this operation, within the context of both traced execution and the change propagation algorithm. However, every traced operation can be invoked in (up to) two contexts. These contexts refer to the levels described in Section 5.2:

Outer context: Untraced, outside the context of a self-adjusting machine's execution

Inner context: Traced, within the context of a self-adjusting machine's execution

Using space provided by the machine (of type `clos_t`), the `invoke` hook saves information associated with the operation for use later by the other hooks. However, this closure information is not saved when the operation is invoked within an untraced, foreign context

Figure 5.2: The TRACE_NODE_ATTRIBUTES signature.

```
#module_begin TRACE_NODE_ATTRIBUTES
 type att_t = [ MEMO | UDPATE              -- self-adj primitives
              | TIME | INTERVAL            -- temporal ordering
              | MEMOTBL | MACHINE | LIVE   -- save/restore info
              | BLK_ALLOC | DELAY_FREE  ]  -- memory management

 type att_off_t = { att : att_t ;        -- attribute
                    off : unsigned int } -- offset of footprint in trace node

 module Offs = List with hd_t := att_off_t -- opt: implement with pointer arith
 module Atts = List with hd_t := att_t     -- opt: implement as a bit field
#module_end
```

via `foreign`. The other hooks allow the machine to check the operation's consistency during change propagation (`consis`), for it to be undone (`revoke`) and for it to be undone and redone (`revinv`). In addition to its own private closure space of type `clos_t`, the hooks also get access to the trace node, which carries additional static and dynamic information.

Different THCs vary in what the features they require of the underlying self-adjusting machine and its trace. As a few examples, consider the following:

- Some primitives require knowing about the global structure of the trace, (e.g., its ordering or its nesting) and others do not.
- Some primitives may become inconsistent and require reexecution; others will never become inconsistent.
- Some primitives require dynamic allocation, while others do not.

We categorize the run-time requirements of THCs using a system of attributes, given in Figure 5.2. Trace node attributes signify that the THC's trace node:

- has extra fields, for storing extra dynamic information
- provides additional operations to the closed hook code

Specifically, each THC definition provides a set of trace node attributes of type `Atts.t`. Conceptually, this type is a list (or set) of attributes, but since it is statically-determined and small in size, we can optimize its representation in several ways. First, we represent the presence or absence of attributes on a trace node with boolean flags. Next, when these features require that certain run-time system state be stored in the trace node, the footprint of that data (i.e., its size and offset) is statically determined.

Below we consider the attributes that trace nodes may carry:

The memo attribute (`MEMO`) indicates that the local continuation should be guarded by a **memo** point and that the trace node should contain a field of type `memo_pt_t`; it also implies the `MEMOTBL` attribute, the `INTERVAL` attribute and the `LIVE` attribute.

The update attribute (`UPDATE`) indicates that the local continuation should be guarded by an **update** point and that the trace node should contain a field of type `update_pt_t`. The this field furnishes the node with the ability to be enqueued for reexecution within the machine. This attribute implies the `INTERVAL` attribute and the `LIVE` attribute.

Temporal attributes guide the machine's use of dynamic time stamps.

- The `TIME` attribute indicates that the trace node should have a unique start time. The trace node stores and furnishes access to this timestamp.
- The `INTERVAL` attribute indicates that the trace node should have a reference to its time interval. The trace node stores and furnishes access to this interval, which consists of a (unique) start time and a (shared) end time. This attribute implies the `TIME` attribute.

Save & restore attributes indicate that certain information about the machine's state should be saved in the trace to supplement the information within the THC's closure record.

- The `MEMOTBL` attribute indicates that the trace node should save a pointer to the current memo table. The trace node furnishes access to this memo table.
- The `MACHINE` attribute indicates that the trace node should save a pointer to the currently-executing machine. The trace node provides access to the machine.
- The `LIVE` attribute indicates that the trace node should save the local contination's live variables. The trace node furnishes access to these live variables, as a structure with one field per variable.

Allocation attributes guide memory management with respect to change propagation.

The `BLK_ALLOC` attribute indicates that the trace node performs dynamic allocations in the trace. It furnishes the trace node with a list to keep track of these, and furnishes the hooks with an allocation function for extending this list with new allocations. Like the space for the closure record, this memory is managed automatically by change propagation. It can be used internally to enrich the (fixed-size) closure record with additional dynamic structure. It need not remain only within the internal representation of the machine; it can also be returned to the inner program.

The `DELAY_FREE` attribute indicates that the trace node should not be reclaimed until change propagation is complete. This attribute must be used, for instance, if pointers to memory allocated within the closure can escape into the inner program. For allocation and reclamation to be sound with respect to change propagation, we must not recycle memory until all traced references to it have been updated, when the trace is consistent with a full reexecution. (See Sections 4.4 and 6.2).

Run-time system implementors choose a subset of the available attributes above, and indicate that their trace hooks require them when they implement a trace hook closure. They need not worry about the actual job of composing trace hook closures into trace

Figure 5.3: The TRACE_NODE_DESCRIPTOR signature.

```
#module TRACE_NODE_DESCRIPTOR
  type desc_t = {   -- trace node descriptor; compiler-generated.
    atts   : Atts.t ;            -- flags: features required by the trace hooks
    offs   : Offs.t ;            -- offsets: storage offsets for attribute data
    size   : size_t ;            -- size of instances; includes THC closures
    consis : trnd_t* -> bool_t ; -- hook: cheap, conservative consistency check
    revinv : trnd_t* -> dst_t  ; -- hook: rerun using space from a prior trace
    revoke : trnd_t* -> void   ; -- hook: undo prior trace
    dbinfo : dbinfo_t ;          -- source code file and line info
  }
#module_end
```

nodes—this is the job of the compiler, and in particular, one of its optimization features (Section 5.8).

Trace nodes. As a first cut, a *trace node* is a single trace hook closure (THC) instance. However, by analyzing the CL program structure, we can group THC instances, putting them into a single shared trace node to amortize their costs (Section 5.8). Since each THC requires a different subset of run-time features, each grouping also requires a different subset of run-time features. Since these extra fields can be shared among many THC instances, their time and space costs are amortized across the grouped instances.

Figure 5.3 lists the structure of trace node descriptors, which contain (compiler-generated) static information about trace nodes. The descriptor specifies what attributes the trace node carries (`atts`), statically-computed offests for data associated with these attributes (`offs`), a statically-computed size of trace node instances (`size`), including the space they require for their trace hook closures, the composed trace hooks themselves (compiler-generated code that composes the node's hook code) (`consis`,`revinv`,`revoke`), and finally, static debugging information (source code location, such as a file name and line number) (`dbinfo`).

The descriptor's attribute information can be represented in a number of ways. As two obvious optimizations, the attribute flags can be encoded using a bit field (instead of a list, or other aggregate structure) and the statically-computed offsets can be compiled into pointer arithmetic in the generated code. Our compiler implements both of these basic optimizations.

5.4 Program representation

We represent suface-level programs in CEAL and intermediate-level programs, those internal to the CEAL compiler, in IL. To facilitate the basic transformations listed above, and particularly their optmizations, we also use additional static control-flow information, with which we augment the program representation in IL. In particular, this augmented representation consists of the IL program as well as local control-flow graph information. One can safely think of IL syntax and its associated control-flow graph as distinct structures with well-defined inter-translations; however, our implementation (Section 5.4) maintains them as views of a common super-structure.

Once in our intermediate language IL, compilation performs the transformations outlined above. In doing so, we rely on properties of the CFG-like structure of CL programs, and exploit standard techniques for extracting this structure from IL. Specifically, we rely on the dominator relation (telling us about local control-flow structure) and the live variable relation (telling us about local data-flow structure).

Control language: CL**.** We introduce a language closely to our intermediate language IL called CL. The role of CL is to augment the structure of IL with some extra control flow structure, namely:

- Control blocks that group code sharing a common trace node and stack frame.

Function names	f, g	$\in$	$\mathsf{FunId} \subset \mathsf{VarId}$
Control block labels	l	$\in$	$\mathsf{LabelId}$
Local variables	x, y, z	$\in$	VarId
Storage types	A, B, C	$\in$	Type
Primitive operation	o	$\in$	Prim

Programs	P	::=	$\overline{F}.b$	Flat, recursive global scoping
Function definitions	F	::=	$f\ (\overline{x : A}).b$	Globally-scoped, first-class block
Control blocks	b, d, p, q	::=	**let** $\overline{l.q}$ **in** p	Local block definitions
		|	$\{x \leftarrow p\}\,; q$	Block sequencing (IL **push**)
		|	**ret** x	Block return (IL **pop**)
		|	$c\ ;\ b$	Command sequencing, single successor
		|	**if** x **then** p **else** q	Conditional, multiple successors
		|	j	Inter-block successor, via jump
Primitive commands	c	::=	$x \leftarrow o\ (\overline{y})$	Primitive operation
		|	$f \leftarrow \mathbf{lift}_{(\overline{x})}(b)$	Lift block into function
Jumps	j	::=	**goto** l	Local jump to labeled block
		|	**tail** $f\ (\overline{x})$	Non-local jump to function

Figure 5.4: The sytnax of CL

- A **lift** primitive that (statically) generates top-level functions from a control block.

Abstract syntax. We give abstract sytnax for CL in Figure 5.4. The sets VarId, FunId, Type and Prim consist of different classes of unique identifiers, where function names can be treated as variables (but, type names and primitive operation names cannot). Programs consist of a set of globally-scoped, mutually recursive functions. Functions consist of formal arguments (variables) and a body which consists of a control block.

Control blocks. Control blocks (or simply *blocks* when clear from context) are similar to statements in C, except that they impose additional control structure. They consist of nested, labeled block definitions (defined in a mutually-recursive fashion), sequenced

blocks, block return, command sequencing, intra-block conditionals and inter-block transfers via indirect jumps (using labels or function names for named indirection).

Commands. Commands are more primitive than control blocks. A *primitive operation* consists of a built-in C primitive (such as arithmetic or logical operation) or a primitive provided by the run-time library or an extension. The **lift** command (statically) promotes a control block into a globally-scoped function, which it names as a first-class value. The variable parameters of the **lift** command specify the live variables of the promoted block[2], which become the formal arguments of the lifted function.

Jumps. Jumps facilitate indirect control-flow. A **goto** transfers control to a labeled block. A **tail** call transfers control to a global function. We elaborate on the contrast between functions and blocks below.

Execution semantics. A central design consideration of CL is to provide a idealized version of C, where a distinction exists between the global control flow of top-level functions and the local control flow of their bodies. In particular, CL stratifies control-flow into two categories:

Functions are first-class; their name space is flat, and recursively-scoped.

Blocks are not first-class; their name space is nested, and lexically-scoped.

We explore the differences between blocks and functions in greater depth below. These differences are analogous to (though not exactly the same as) the differences in C between functions and statements.

Since CL is closely related to IL, we only give an informal account of its operational semantics. We focus first on the contrast between blocks and functions, especially as they

2. The compiler computes this set using a standard live variable analysis.

Table 5.1: Functions versus blocks in CL

	CL **Blocks**	CL **Functions**
Language status	Not a value	First-class value
Static structure	Nested, lexical scoping	Flat, global scoping
Transfer patterns	Comparatively static	Highly dynamic
Transfer operation	Mutates stack frame	Replaces stack frame
Trace structure	Trace node	Recurring trace node pattern

relate to the ultimate target language of compilation, C. We also discuss the semantics of CL's additional command: **lift**.

Blocks versus functions. Table 5.1 contrasts blocks and functions tabularly. While functions are first class, blocks are not. While blocks can nest, functions do not. Blocks transfer control locally to other blocks, while functions can be treated like first-class data, making their control-flow more difficult to analyze. In terms of the underlying C machine, a block transfers control by changing the instruction pointer (i.e., the program counter); a function transfers control by changing the instruction pointer and replacing the current stack frame with a new one. Due to having more static information about control flow, blocks can share trace resources by sharing *trace nodes*; function invocations do not participate in such sharing with each other.

Nesting and dominance. While it may be uncommon to think of (basic) blocks as having a *nested* structure, Appel (1998b) points out this nested structure is indeed present, in both the dominator tree of a control flow graph of basic blocks, as well as in the syntactic nesting of their corresponding functional description. So, when we say that a block is nested within another, we mean to say that a block is *dominated* by another in their corresponding control

flow graph, or *nested within* another in their syntactic (or pictorial) description. This relation gives us static information about control flow, which helps us allocate machine resources (e.g., the registers, stack and trace).

Laying out the call stack. Stack frames are an allocation strategy for a more general structure—the structure of dynamic environments. Due to their not being first-class, the control flow of basic blocks is easier to statically analyze than that of functions, and leads to common optimizations in this layout. It is instructive to consider how C benefits from this distinction between local and non-local control flow by using standard compiler technology. As one example, the compiler may perform register allocation, which is generally a part of stack frame layout: when the registers are insufficient in number, excess live data is "spilt" onto the stack. Typically, C compilers do such optimizations based on the knowledge of local control and data flow (often summarized in control flow graphs) that can often be readily extracted for local control flow, but which is not known statically for global control flow (control flow at the level of functions).

Due to more precise contextual information, local control flow between basic blocks is cheaper and easier to analyze than non-local control flow between functions. As a result, resource allocation (such as registers and the call stack) are typically done intra-procedurally, making local control dynamically cheaper than non-local control. The result of these local analyses is that local variable assignment and variable access in C becomes very economical: rather than consult a dynamic environment (as our abstract machine does), the variable content is either allocated on the stack (at a fixed, statically-known offset from the stack pointer) or better still, in a C machine register.

We note that in CL, as in IL (Section 4.3), we decompose function calls into tail calls and separate stack operations, which save and restore the caller's local state. Therefore, when we consider the operational cost of function application, we are considering that of only tail-recursive applications that lack this local saving step. Even so, the cost of

transfering control between basic blocks is cheaper still, since it generally requires a small (not wholesale) change to the current stack frame, if any. By contrast, a tail call generally requires replacing the current stack frame with a new one.

Laying out the trace. Beyond the stack, in the context of self-adjusting machines, we have an additional resource to consider: the execution trace. *Trace nodes* provide an allocation strategy for the trace. As with stack frames, trace nodes are applicable when certain information is known statically. Specifically, this information consists of the sizes, offsets and lifetimes of certain spatial resources (the trace storage required by each primitive operation). When known, this information can be used to statically group traced primitives and amortize their combined run-time tracing cost. As with stack frames, trace nodes are statically allocated by considering static information about control and data flow.

In CL, we express this grouping syntactically by using the notion of a block. Blocks have the right properties for such groupings: enough of their control-flow is known statically to supply each of their instructions with a unique, statically-computed offset and footprint in the trace node's payload. By contrast, the trace of a function generally consists of dynamically recurring patterns of the trace nodes, whose overall structure and size is statically unknown and is generally undecidable. Where statically-known structure can be discovered in the non-local control flow, it can be encoded (via function-inlining) into static control flow, and thus exposed to our optimizations. For instance, the layout of code in statically-structured control blocks leads to a statically-structured layout of its trace nodes.

Semantics of primitive operations. We give an informal account of CL primitive operations. The **lift** operation promotes a control block into a top-level function. In particular, we translate the **update** points of IL by enclosing the guarded code with the **lift** primitive of CL. When the **lift** primitive is expanded at compile time, this transformation lifts local code guarded by an **update** point into the global scope, so that it can be used to construct

a function pointer and closure record. In Section 5.7, we give a normalization algorithm that expands these compile-time **lift** operations and puts the program into a "fully-lifted" normal form.

CFG representation. Our compiler is implemented as a series of tree walks, which take advantage of tree shape of the program's dominance structure. To implement this representation, we use an applicative zipper interface inspired by Ramsey and Dias (2006). As an instance of this tree-based approach, our normalization algorithm alters the dominance structure in order to statically expand CL **lift** commands. See Section 5.7 for the graph version of the algorithm. In the tree transformation, the units computed by the graph algorithm correspond to subtrees of the dominator tree, and thus, subtrees of the syntactic encoding. The tree transformation consists of relocating the unit subtrees as global functions.

Our implementation as a tree transformation relies on the dominance structure being encoded syntactically in order to function correctly. In turn, through unifying the dominator tree and the syntax tree of the program, the implementation of analyses and transformations of our compiler can be expressed concisely.

5.5 Static analyses

Dominator and live variable analysis are needed to generate IL and CL programs in the first place, since IL requires the static single-assignment property[3] and both use syntax to encode domination. We discuss these standard analyses within the context of our compilation strategy. We give a simple but nonstandard analysis that we use to translate the IL program and apply the DPS transformation; it statically analyzes the program's placement of **update** and **memo** points.

3. Recall that IL uses functional notation, which is closely related to the SSA form of a program.

Domination & post-domination. In the context of control-flow graphs, domination and post-domination relations give us statically-known facts about a program's dynamic control-flow across time, and have been a standard part of compiler technology for decades (Appel, 1991; Muchnick, 1997; Appel, 1998a). From the viewpoint of a node in the control-flow graph, the dominator relation tells us about the past; specifically, it tells us from where we must have traveled. If block d *dominates* block b and control is currently at block b, then we know that block d was executed *in the past*. This relation is built by analysing the predecessor relation of the CFG, which tells us for each node, which other nodes precede it.

Dually, the post-dominator relation is parameterized by the sucessor relation of the CFG, and it gives a dual set of facts about the future: If block p *post-dominates* node b and control is currently at node b, then we know that node p will be executed *in the future*.

In the context of constructing self-adjusting machines as C code, these properties of domination and post-domination are useful throughout:

1. To statically layout trace nodes (Section 5.3).
2. To estimate the lifetime of local variables (Section 5.5)
3. To translate IL **memo** and **update** points into CL (Section 5.6)
4. To selectively apply destination-passing style to stack operations (Section 5.8).
5. To allocate registers and layout the stack frame (done by any standard C compiler).

In CL, domination is encoded syntactically: if a syntactically contains b then block b is dominated by node a. The converse may not necessarily be true, but can be enforced as a compiler-wide invariant (see below).

Post-domination is also encoded syntactically, in various forms. First, each CL command is post-dominated by its body. In sequential composition of CL control blocks (b ; p), the

the second block of the sequence (p) is a post-dominator of the first (b). In terms of IL, a similar pattern of post-domination is encoded by **push** operations: the pushed function post-dominates the **push** body. In both CL and IL, post-domination can be encoded with labeled blocks. If some block l contains the only return of a function with no other non-local control flow, then block l post-dominates every block of the function. No matter what control path is taken, control must reach l to return.

Some regions of code share a common dominator and post-dominator. These are known as single-entry, single-exit (SESE) regions. The bodies of C functions are single-entry, single-exit regions. For structured control flow (in the absence of labels and gotos), C statements are also instances of single-entry, single-exit regions. By sharing a dominator and post-dominator, these blocks of code have a common lifetime, allowing the possibility for optimizations to exploit this shared lifetime to amortize run-time costs, such as tracing overhead (Section 5.3).

Enforcing the dominance invariant. While syntactic containment always implies dominance, the converse is not always true: syntactic independence does not imply the lack of dominance. However, since it simplifies correspondances, we would like to tacitly assume that the converse holds, that is to say, that when a block b is dominated by another block d, then the dominated block b will be syntactically within the dominating block d. However, strictly speaking this does not hold after arbitrary program transformations. As an example, consider the destination-passing-style (DPS) transformation.

But we can remedy this. By recomputing the dominator tree of the control-flow graph, this property can be restored: we transform the program syntactically (not semantically), and renest domianted blocks under the blocks that dominate them. Our compiler performs this transformation to maintain the syntactic encoding of the dominator tree. In turn, the correspondance between syntax (tree structure) of the program and its dominator tree

simplifies transformations that rely on this information, since they can often be phrased as tree walks (Section 5.4).

Live variable analysis. In the informal semantics of CL, as in the semantics of IL, the local state maps variables to dynamic values. In IL, the environment is static single-assignment: each local program variable is unique, and is only assigned once. In CL, as in C, this environment is more like a mutable stack, and it may not observe the static single assignment rule. In both cases, the purpose of live variable analysis is to give information about the future use of this dynamic local environment, from the point of view of each (static) control point.

At each block in the control-flow graph, live variable analysis gives us a set of local variables that (conservatively) may be used in the future, upon entry to the block. If a variable is missing from this set, then it will not be used, regardless of how program data is changed. Most compiler texts give the formal and practical details, which we straighforwardly adapt to IL and CL Appel (1991); Muchnick (1997); Appel (1998a).

In the context of self-adjusting machines, this live variable information is useful for translating **memo** and **update** points, for determining which variables to save in the trace. In our compilation strategy, this shows up in translating IL to CL (Section 5.6).

Analysis of memo and update points. To inform both our translation and selective destination-passing style (DPS) transformation, we use a simple analysis for conservatively determining the dependencies of return values upon the modifiable control-flow of **memo** and **update** points.

First, the translation of **memo** points requires knowledge of the address of the returned content (the memoized results, which are returned rather than recomputed); this knowledge is easily satisfied through coding in a destination-passing style (DPS). Similarly,

update points require a destination-passing style as well, for the soundness of change propagation. Meanwhile, Section 4.5 gives a DPS transformation for IL, but it converts all stack operations without any selectivity.

Below, we give MU-analysis, which informs more a selective technique. Depending on the placement of **memo** and **update** points, a return value is tranformed in up to three ways:

- The identity transformation, where the return value is given directly on the stack. This is possible when no **memo** or **update** points lie within the region.
- The DPS transformation, with indirection via stable (non-modifiable) memory;
- The DPS transformation, with indirection via modifiable memory.

In both cases of indirection, the memory is provided within and managed with the trace. These cases only differ in whether the memory is traced as a modifiable or not. The translation of **update** points (conservatively) requires modifiable memory, while **memo** points do not.

MU-Analysis. We use a context-sensitive control-flow analysis to statically-estimate the placement of **memo** and **update** points. The contexts and results of our analysis are both drawn from simple lattices:

$$\mathcal{L}_2 = \{0, 1\}$$

$$\mathcal{L}_{\mathsf{mu}} = \mathcal{L}_2 \times \mathcal{L}_2$$

The lattice $\mathcal{L}_{\mathsf{mu}}$ consists of pairs of binary numbers, ordered such that $(0, 0)$ is the bottom element and $(1, 1)$ is the top element. The first binary digit encodes whether there *may be* a preceeding **memo** point. Similarly, the second digit encodes whether there *may be* a preceeding **update** point. Both estimates are static, and hence conservative.

Based on these definitions, the analysis follows a standard lattice-based fixed point formula: At each control block, for each possible context (an element of $\mathcal{L}_{\mathsf{mu}}$), we produce a context for each successor, which we unify with the other predecessors of the successor (via their *join*, i.e., least upper bound). Throughout the analysis, for each top-level CL function, for each possible $\mathcal{L}_{\mathsf{mu}}$ context, we have an associated $\mathcal{L}_{\mathsf{mu}}$ element that indicates (a conservative estimate of) its post condition. This lets us perform the analysis globally, across the entire CL program. We continue updating the lattice elements at each control block until we reach a fixed point, which is guaranteed by virtue of the finite height of the lattice structures. Similarly, the small, finite size of the lattice structure means that there are only a small number of possible contexts to consider for each function.

When complete, we inspect the $\mathcal{L}_{\mathsf{mu}}$ element at each IL **push** and **pop** operation. Depending on the estimate of the analysis, we either apply the DPS conversion in one of the two ways outlined above, with or without modifiable memory, or do nothing at all. Since each function is analyzed with every possible calling context, we specialize functions with up to three versions. Rather than use dynamic dispatch, we statically specialize call sites according to this context[4].

5.6 Translation

We translate the IL and its primitives into CL and primitives from the run-time library, which gives a C-based implementation of self-adjusting machines. We give an overview of the translation (Section 5.6). We describe the primitives provided in the target language. We describe the cases for translating traced operations, control-flow constructs, stack operations and the incremental operators **memo** and **update**. Finally, we describe the generation of redo and undo code.

4. Our current implementation lacks support for indirect function application, i.e., function pointers. In those cases, an explicit annotation (e.g., such as a qualified type for the function pointer) would guide the analysis and translation.

Overview. The role of translation from IL to CL is to distinguish between local and non-local control flow, to replace uses of traced operations with the bindings of their implementations as trace hook closures (Section 5.3), and to expand the other core primitives of the machine—**memo**, **update**, **push** and **pop**—into CL constructs and with certain compile-time and run-time primitives.

The forms **memo** and **update** each save and restore the local state of an IL program—an environment ρ and an IL expression e. To compile these forms, the following questions arise: How much of the environment ρ should be recorded in the runtime trace and/or memo table? Once an IL expression e is translated into a target language, how do we reevaluate it during change propagation?

Saving and restoring ρ. First, we address how we save the environment ρ. At every incremental checkpoint operation (either **memo** or **update**) we use a liveness analysis (Section 5.5) to approximate the live variables $\mathsf{LV}(e)$ at each such e, and then save not ρ, but rather $\rho|_{\mathsf{LV}(e)}$, i.e., ρ limited to $\mathsf{LV}(e)$. This analysis has two important consequences: we save space by not storing dead variables in the trace, and we (monotonically) increase the potential for **memo** matches, as non-matching values of dead variables do not cause a potential match to fail.

Fine-grained reevaluation of (ρ, e). Second, we address the issue of fine-grained reevaluation. This poses a problem since languages such as C do not allow programs to jump to arbitrary control points, e.g., into the middle of a procedure. We use a low-level λ-lifting technique to flatten the program at these points. That is, we translate **update** points using the **lift** operation of CL. The **lift** operation can be used to express the lifting of local code guarded by **update** into the global scope, so that it can be used to construct a function pointer and closure record. Separately, we give an algorithm that we call normalization,

which puts the program into a "fully-lifted" normal form and removes these compile-time operations (Section 5.7).

Destinations as contextual information. The translation rules use the metavariable δ to range over three cases that characterize the destination of the expression being translated:

$$\begin{array}{lllll} & \mathrm{d} & \in & \mathsf{VarId} \\ \text{Destination} & \delta & ::= & \circ \mid \bullet \mid \mathrm{d} \\ \text{Translation} & [\![\cdot]\!]\ \cdot & : & \mathsf{IL\text{-}Exp} \rightarrow \mathsf{Dest} \rightarrow \mathsf{CL\text{-}Block} \\ & [\![e]\!]\ \delta & : & \mathsf{CL\text{-}Block} \end{array}$$

The MU-analysis and destination-passing style guide the translation between these cases. The first case $\delta = \circ$ signifies that the expression has not been transformed into destination-passing style. This case is possible if the expression contains no **memo** or **update** points (neither syntactically, nor semantically). The next case $\delta = \bullet$ signifies that the expression has a destination, but that it has not yet been determined (i.e., has not yet been allocated or otherwise bound to a variable). The final case $\delta = \mathrm{d}$ signifies that the expression has a destination and that it is bound to the variable d.

CL **target primitives.** The translation uses the primitives below. These primitives involve run-time representations of traces, in terms of trace hook closures, trace nodes and trace node descriptors. Section 5.3 discusses the run-time system interface in greater detail.

These primitives are statically-expanded at compile time, to code that is statically-specialized for a trace node descriptor D:

desc is expanded at compile time. It is a meta-level operator that sends IL primitives, as well as other traced operations provided by library extensions, to their statically-generated trace node descriptors.

$\mathsf{invoke}_{\mathrm{D}}$ expands to the `invoke` function for the trace hook closure implementation, whose trace node descriptor is D.

$\mathsf{clos\_of}_{\mathrm{D}}$ is partially-expanded at compile time. It gives a static offset for the footprint of the trace hook closure within the trace node footprint.

$\overline{\mathrm{x}} \leftarrow \mathsf{clos\_vars}_{\mathrm{D}}$ is partially-expanded at compile time. It restores and rebinds live variable values that are saved in the closure of the given trace node.

The primitives below are provided by the run-time library to build and manipulate traces within the context of a self-adjusting machine. We refer the reader to Sections 6.4 and 6.5 for more details about this interface; below, we give a high-level description.

mk_{D} dynamically instantiates (allocates and initializes) a trace node for a statically-determined trace node descriptor D.

memo_chkpt provides a memoization check-point: it searches for trace nodes in the current memo table. It returns a reusable match, or `NULL` if none is found. If none is found, it adds the trace node argument to the table for use in future cycles of change propagation.

frame_memo takes a trace node whose live variables and program text match the current local state (as returned by memo_chkpt). It edits the trace to reuse the given trace node, and change-propagates the remaining trace for the current frame. The caller provides the current destination is as a second argument, which is returned as a result. In this style, this primitive is consistent with the run-time system techniques (tracing, memoization and change propagation) being properly tail-recursive.

frame_push takes stack-allocated space for the frame, and trace-allocated space for the time stamp. It uses the frame space to store certain internal cursor information, and uses the time stamp to create an interval and store its end time. When a region of code

Table 5.2: Translation of IL to CL: traced operations and control-flow

$$
\begin{array}{rcl}
[\![\textbf{let } x = o(\overline{x}) \textbf{ in } e]\!]\ \delta & = & \left[\begin{array}{l} t \leftarrow \mathsf{mk}_D() \ ; \\ x \leftarrow \mathsf{invoke}_D(t, \mathsf{clos\_of}_D(t), \overline{x}) \ ; \\ [\![e]\!]\ \delta \\ \quad \textit{where } D = \mathsf{desc}(o) \end{array}\right. \\
[\![\textbf{let } f(\overline{x}).e_1 \textbf{ in } e_2]\!]\ \delta & = & \textbf{let } l_f.[\![e_1]\!]\ \delta \textbf{ in } [\![e_2]\!]\ \delta \\
[\![f(\overline{y})]\!]\ \delta & = & \left\{\begin{array}{ll} \overline{x} \leftarrow \overline{y} \ ; \textbf{ goto } l_f & \textit{when } f \textit{ is local} \\ \textbf{tail } f(\overline{y}) & \textit{when } f \textit{ is global} \end{array}\right. \\
[\![\textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2]\!]\ \delta & = & \textbf{if } x \textbf{ then } [\![e_1]\!]\ \delta \textbf{ else } [\![e_2]\!]\ \delta
\end{array}
$$

contains **memo** or **update** points, it is put into destination-passing style (DPS) by the compiler, and the stack frame induces an interval that the translation traces using this operation. When a region of code does not contain **memo** or **update** points, it does not require an interval within the trace, and this operation can be avoided. As mentioned above, the MU-analysis guides the translation between these cases.

frame_pop is the complement to frame_push. It marks the end of the lifetime of the machine stack frame.

In addition to these operations, which concern trace nodes, the translation additionally uses the following primitives provided by either the C language or the self-adjusting machine implementation:

`alloca` allocates space directly on the stack, rather than in the heap. The translation uses it to signify that certain memory should be stack-allocated. The implementation of this primitive is built directly into standard C compilers; here, we treat it abstractly.

Traced operations and control-flow. Table 5.2 gives the translation for traced operations and control-flow constructs (viz. function abstraction, function application and conditionals).

For traced operations, the translation uses the trace node interface. The translation uses mk to instantiate a trace node for the descriptor D, which consists of the static information for the traced operation o. The translation invokes the traced operation according to the trace node descriptor; the invocation receives the trace node, space within the trace node reserved for its closure, and the operands $\overline{x}$. The traced operation consists of all the hooks associated with the operation, including the code necessary to redo and undo the operation (Section 5.3).

For function abstraction and function application, the translation encodes control flow in one of two ways: locally, as a control block; or globally, as a top-level function. Local control blocks are possible when use of the function is not first-order [5]; functions are possible when the function is globally-scoped (not nested within the local scope of another function). The translation of conditional control flow is the identity homomorphism (i.e., we use C conditionals in a straightforward manner).

Stack operations: push and pop. Table 5.3 gives the translation for stack operations. The cases vary depending on how the selective destination-passing-style (DPS) transformation was applied to the program. Expressions that contain **memo** and/or **update** points return their results indirectly, via a destination. For these expressions, we thread a $\bullet$ to signify that a destination will be determined; for all other expressions, we thread a $\circ$ to signify that there is no destination (nor any **update** or **memo** points). The destinations are determined by the **memo**-guarded allocations that the DPS-transformation introduces. The translation handles that case in Table 5.4, described below.

5. though, CL provides the compile-time operator **lift** to overcome this limitation.

Table 5.3: Translation of IL to CL: **push** and **pop** forms

$$
\begin{array}{rcl}
[\![\textbf{let } f(x).e_2 \textbf{ in push } f \textbf{ do } e_1]\!]\ \delta & = & \left[\begin{array}{l} \{x \leftarrow [\![e_1]\!]\ \circ\}\ ;\ [\![e_2]\!]\ \delta \\ \quad \textit{where } e_1 \textit{ is } \textbf{update}\textit{- and } \textbf{memo}\textit{-free} \end{array}\right. \\
[\![\textbf{pop } \overline{x}]\!]\ \circ & = & \textbf{ret } \overline{x} \\
[\![\textbf{let } f(x).e_2 \textbf{ in push } f \textbf{ do } e_1]\!]\ \delta & = & \left[\begin{array}{l} t \leftarrow \mathsf{mk}_D()\ ; \\ f \leftarrow \texttt{alloca(sizeof(frame\_t))}\ ; \\ \mathsf{frame\_push}(f, \mathsf{clos\_of}_D(t))\ ; \\ \{\ x \leftarrow [\![e_1]\!]\ \bullet\ \}\ ; \\ \mathsf{frame\_pop}(); \\ [\![e_2]\!]\ \delta \\ \quad \textit{where } D = \mathsf{desc}(\textbf{alloc}_{\texttt{(Time.t)}}) \end{array}\right. \\
[\![\textbf{pop } d]\!]\ d & = & \textbf{ret } d
\end{array}
$$

Incremental operators: memo and update. Table 5.4 gives the translation for the **memo** and **update** points. The translation has two cases for **memo** points which vary on whether the destination has been determined or not.

In the first **memo** case, the destination is not determined by the context of the **memo** point. In these cases the **memo** point guards the allocation of the destination, which immediately follows the **memo** point. This **memo; alloc** sequence is a pattern introduced by the DPS transformation in order to encode the indexed allocation of destinations. The translation handles this pattern specially (since the destination is being determined, not predetermined); the rule given combines the **memo** point with the allocation of its destination. The trace descriptor (D) combines a descriptor the memo point ($\overline{x}.e$) and with a descriptor for the allocation of the destination. The memo trace node is introduced using mk and is initialized with the values of the current set of live variables $\overline{x}$.

Table 5.4: Translation of IL to CL: **memo** and **update** forms

$$
\begin{array}{rcl}
[\![\textbf{memo let } \mathrm{d} = \textbf{alloc}(\mathrm{n}) \textbf{ in } e]\!]\ \bullet & = & \left[\begin{array}{l}
\mathrm{m}_1 \leftarrow \mathsf{mk}_{\mathrm{D}}(\overline{\mathrm{x}})\ ; \\
\mathrm{m}_2 \leftarrow \mathsf{memo\_chkpt}_{\mathrm{D}}(\mathrm{m}_1)\ ; \\
\textbf{if } \mathrm{m}_2 \textbf{ then } \mathsf{frame\_memo}(\mathrm{m}_2) \\
\textbf{else } \mathrm{d} \leftarrow \mathsf{mk}_{\mathsf{desc}(\textbf{alloc}(\mathrm{n}))}(\,);\ [\![e]\!]\ \mathrm{d} \\
\quad \textit{where } \mathrm{D} = \mathsf{desc}(\textbf{memo}_{(\overline{\mathrm{x}}.e)}) \\
\quad \textit{and } \overline{\mathrm{x}} = \mathsf{live}(e)
\end{array}\right. \\
[\![\textbf{memo } e]\!]\ \mathrm{d} & = & \left[\begin{array}{l}
\mathrm{m}_1 \leftarrow \mathsf{mk}_{\mathrm{D}}(\overline{\mathrm{x}})\ ; \\
\mathrm{m}_2 \leftarrow \mathsf{memo\_chkpt}_{\mathrm{D}}(\mathrm{m}_1)\ ; \\
\textbf{if } \mathrm{m}_2 \textbf{ then } \mathsf{frame\_memo}(\mathrm{m}_2) \textbf{ else } [\![e]\!]\ \mathrm{d} \\
\quad \textit{where } \mathrm{D} = \mathsf{desc}(\textbf{memo}_{(\overline{\mathrm{x}}.e)}) \\
\quad \textit{and } \overline{\mathrm{x}} = \mathsf{live}(e) \text{ with } \mathrm{d} \in \overline{\mathrm{x}}
\end{array}\right. \\
[\![\textbf{update } e]\!]\ \mathrm{d} & = & \left[\begin{array}{l}
\mathrm{t} \leftarrow \mathsf{mk}_{\mathrm{D}}(\overline{\mathrm{x}}) \\
\mathrm{f} \leftarrow \textbf{lift}_{(\mathrm{t})}(\overline{\mathrm{x}} \leftarrow \mathsf{clos\_vars}_{\mathrm{D}}(\mathrm{t})\ ;\ [\![e]\!]\ \mathrm{d}) \\
[\![e]\!]\ \mathrm{d} \\
\quad \textit{where } \mathrm{D} = \mathsf{desc}(\textbf{update}_{(\mathrm{f}\ ;\ \overline{\mathrm{x}})}) \\
\quad \textit{and } \overline{\mathrm{x}} = \mathsf{live}(e)
\end{array}\right.
\end{array}
$$

The memo point searches for a matching trace node using memo_chkpt, passing the trace node t as a key. If a match m is found, then we retrieve the destination from the match using get and yield control to the self-adjusting machine using frame_memo. Otherwise, if t fails to match, then we bind the destination d using the space in trace node t.

When a destination is already variable-bound, we translate a **memo** point by using that destination. Hence, the two cases of **memo** translation are very similar, but also crucially

distinct: In the first case the destination is being determined; in the second case, the destination is already fixed.

To translate an **update** point, we use mk to allocate and initialize a trace node that saves a redo function f for the expression and the values of the expression's live variables $\overline{x}$. Using get, the lifted update function f restores the values of the live variables from the trace node argument. We use the **lift** primitive to make the undo and redo thunk code accessible to the run-time library, as function pointers. We discuss these details separately, below.

Redo and undo code generation. Above we present the *invocation channel* of the translation: for each traced operation, we translate its code into the code to invoke it. Specifically, we elide the details of code generation for the trace node's redo and undo thunks. To show how we translate the lifted code, we should also give the *redo*, *undo* or *consistency-check channels*, where we use not *invoke* but instead the `revinv`, `revoke` or `consis` operations, respectively. Here, we describe our implementation of these thunks at from a high level.

We use a tree traversal to perform the translation; upon a top-down and bottom-up traversal of an IL term, we generate four CL terms. During translation, we refer to these generated terms as the *channels* of the translation's output:

1. *The invocation channel* consists of a CL term that gives the translation, as presented above;

2. *The normalized invocation channel* consists of a CL term that gives a normalized translation which we generate as we traverse—we do so by precomputing the set of blocks that must be lifted by the normalization algorithm (i.e., those lifted via **lift**), and by generating seperate functions for these blocks as we traverse them;

3. *The redo, undo and consistency-checking channels* consists of a triple of CL terms that gives the current redo, undo and consistency-check thunk bodies, i.e., the composed

`revinv`, `revoke` and `consis` calls for each traced operation being combined into a trace node.

At some points during traversal, the last triple may be nonexistent—i.e., when no traced operation is locally involved in the translation. When we emit a trace node descriptor, we also emit code for its hooks: the final triple returned by the traversal provides this code. The second channel of terms emitted by the traversal—the normalized version of the program—is required since the hooks may require reentering the program's functions at arbitrary control points. We separately provide the first channel of terms—terms that have not be normalized—to emphasize that normalized terms are only required for reexecution, and not for initial trace execution and trace generation. We refer the reader to Section 5.4 for more information on our implementation's internal representation of programs and term transformations.

5.7 Normalization

We say that a CL program is in *normal form* if and only if every it is free of **lift** commands. In this section, we describe an algorithm for normalization that represents programs with control flow graphs and uses dominator trees to restructure them.

Rooted control-flow graphs. We represent CL programs with a particular form of rooted control flow graphs, which we shortly refer to as a *program graph* or simply as a *graph* when it is clear from the context.

The graph for a program P consists of nodes and edges, where each node represents a top-level function definition, a control block, or a distinguished root node. The tag of the node carries the information necessary to reconstruct the syntactic representation of the node (as a function definition, control block or root node). As a matter of notational convenience, we name the nodes with the label of the corresponding basic block or the

name of the function, e.g., u_l or u_f. For blocks that have no label, we create a unique label based on the label of the block that contains it (or name of the function that contains it), and its path within this contained block.

The edges of the graph represent block containment and control transfer. For each function node u_f whose first block is u_l, we have an edge from u_f to u_l; similarly, for each block u_l contained with another block u_k, we have an edge from u_l to u_k. For each **goto** jump from block l to block k we have an edge from node u_l to node u_k, tagged as a **goto**. For each **tail** jump from block l to function f we have an edge from node u_l to node u_f, tagged as a **tail**.

Once lifted, a block can be entered as a top-level function. We call a node a *entry node* if it corresponds to a block being lifted via a use of the **lift** primitive, or a top-level function. In the rooted control-flow graph, we make the control flow of lifts graphically explicit: For each entry node u_l, we insert into the graph an edge from the root node to node u_l. We call this edge an *entry edge*.

There is a (efficiently) computable isomorphism between a program and its graph that enables us to treat programs and graphs as a single object. In particular, by changing the graph of a program, our normalization algorithm effectively restructures the program itself.

Lemma 5.7.1 (Programs and Graphs)**.** *The program graph of a CL program with* n *blocks can be constructed in expected* $O(n)$ *time. Conversely, given a program graph with* m *nodes, we can construct its program in* $O(m)$ *time.*

Proof. We construct the graph in two passes over the program. In the first pass, we create a root node and create a node for each block. We insert the nodes for the blocks into a hash table that maps the block label to the node. In the second pass, we insert the edges by using the hash table to locate the source and the target nodes. Since hash tables require

expected constant time per operation, creating the program graph requires time in the number of blocks of the graph.

To construct the program for a graph, we follow the outgoing edges of the root. By the definition of the program graph, each function node is a target of an edge from the root. For each such node, we create a function definition. We then generate the code for the function by generating the code for each block that is reachable from the function node via **goto** edges. Since nodes and edges are tagged with the CL blocks that they correspond to, it is straightforward to generate the code for each node. Finally, the ordering of the functions and the blocks in the functions can be arbitrary, because all control transfers are explicit. Thus, we can generate the code for a program graph by performing a single pass over the program code. □

Dominator trees and units. Let $G = (V, E)$ be a rooted program graph with root node u_r. Let $u_k, u_l \in V$ be two nodes of G. We say that u_k *dominates* u_l if every path from u_r to u_l in G passes through u_k. By definition, every node dominates itself. We say that u_k is an *immediate dominator* of u_l if $u_k \neq u_l$ and u_k is a dominator of u_l such that every other dominator of u_l also dominates u_k. It is easy to show that each node except for the root has a unique immediate dominator. The immediate-dominator relation defines a tree, called a *dominator tree* $T = (V, E_I)$ where by $E_I = \{(u_k, u_l) \mid u_k \text{ is an immediate dominator of } u_l\}$.

Let T be a dominator tree of a rooted program graph $G = (V, E)$ with root u_r. Note that the root of G and T are both the same. Let u_l be a child of u_r in T. We define the *unit of* u_l as the vertex set consisting of u_l and all the descendants of u_l in T; we call u_l the *defining node* of the unit.

Lemma 5.7.2 (Cross-Unit Edges)**.** *Let* $G = (V, E)$ *be a rooted program graph and* T *be its dominator tree. Let* U_k *and* U_m *be two distinct units of* T *defined by vertices* u_k *and* u_m

respectively . Let $u_l \in U_k$ *and* $u_n \in U_m$ *be any two vertices from* U_k *and* U_m*. If* $(u_l, u_n) \in E$*, i.e., a cross-unit edge in the graph, then* $u_n = u_m$*.*

Proof. Let u_r be the root of both T and G. For a contradiction, suppose that $(u_l, u_n) \in E$ and $u_n \neq u_m$. Since $(u_l, u_n) \in E$ there exists a path $p = u_r \rightsquigarrow u_k \rightsquigarrow u_l \rightarrow u_n$ in G. Since u_m is a dominator of u_n, this means that u_m is in p, and since $u_m \neq u_n$ it must be the case that either u_k proceeds u_m in p or vice versa. We consider the two cases separately and show that they each lead to a contradiction.

- If u_k proceeds u_m in p then $p = u_r \rightsquigarrow u_k \rightsquigarrow u_m \rightsquigarrow u_l \rightarrow u_n$. But this means that u_l can be reached from u_r without going through u_k (since $u_r \rightarrow u_m \in G$). This contradicts the assumption that u_k dominates u_l.

- If u_m proceeds u_k in p then $p = u_r \rightsquigarrow u_m \rightsquigarrow u_k \rightsquigarrow u_l \rightarrow u_n$. But this means that u_n can be reached from u_r without going through u_m (since $u_r \rightarrow u_k \in G$). This contradicts the assumption that u_m dominates u_n.

□

Example. Normalization is made possible by an interesting property of units and cross-unit edges. In particular, normalization lifts the units of the dominator tree that correspond to the new unit-defining nodes, including the entry nodes marked by the **lift** commands. We identify the new unit-defining nodes using the dominator tree; once identified, we normalize the program by lifting the units of all new unit-defining nodes into top-level functions, and redirecting cross-unit edges into tail calls.

Figure 5.5 shows a rooted control-flow graph (CFG) where the root node (labeled 0) is connected to the nodes 1, 3, 11 and 12. Figure 5.6 shows the dominator tree for the rooted CFG. The entry nodes (viz. 1, 3, 11 and 12) are top-level units; node 18 is also a top-level unit, as it is required as a shared dependency by the other units. Figure 5.7 shows the normalized CFG, where new function nodes (viz a, b, c and d) are inserted for

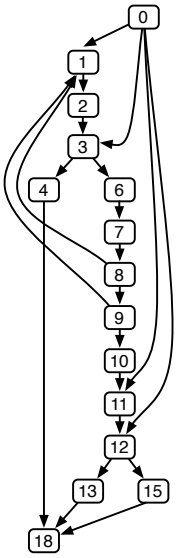

Figure 5.5: Rooted CFG.

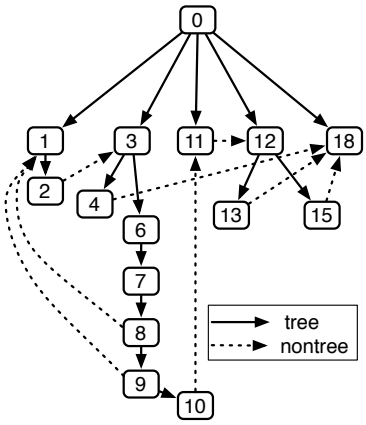

Figure 5.6: Dominator tree.

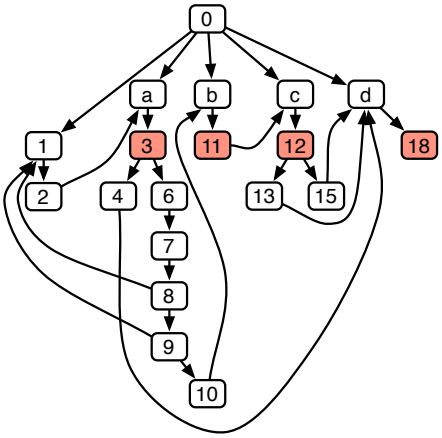

Figure 5.7: Normalized CFG.

each new top-level unit (whose critical nodes are highlighted), and existing edges into the units are redirected from these critical nodes to instead target the new function nodes.

NORMALIZE**: a graph algorithm.** Figure 5.8 gives the pseudo-code for our normalization algorithm, NORMALIZE. At a high level, the algorithm restructures the original program by creating a function from each unit and by changing control transfers into these units to tail jumps as necessary.

Given a program P, we start by computing the graph G of P and the dominator tree T of G. Let u_r be the root of the dominator tree and the graph. The algorithm computes the normalized program graph $G' = (V', E')$ by starting with a vertex set equal to that of the graph $V' = V$ and an empty edge set $E' = \emptyset$. It proceeds by considering each unit U of T.

Let U be a unit of T and let the node u_l of T be the defining node of the unit. If u_l is a not a function node, then we call it and all the edges that target it *critical*. We consider two cases to construct a set of edges E_U that we insert into G' for U. If u_l is not critical, then E_U consists of all the edges whose target is in U.

Figure 5.8: Pseudo-code for the NORMALIZE algorithm

```
let G = (V, E) = rooted-graph(P, u_r)          — graph rooted at u_r
let T = dom-tree(G)                            — tree rooted at u_r
G' ← (V', E'), where V' ← V and E' ← ∅         — empty edge set
for each unit U of dominator tree T do         — iterate over units of T
    u_l ← defining node of U                   — consider each defining node u_l
    if u_l is a function node then             — u_l is not critical
        E_U ← {(u_l, u) ∈ E | (u ∈ U)}
    else                                       — u_l is critical
        suppose f ∉ P and x̄ = live(l)          — create a fresh function
        V' ← V' ∪ {u_f}
        tag(u_f) ← f(x̄)
        — add function edges, and intra-unit edges that do not target u_l
        E_U ← {(u_r, u_f), (u_f, u_l)} ∪ {(u_1, u_2) ∈ E | u_1, u_2 ∈ U, u_2 ≠ u_l}
        for each critical edge (u_k, u_l) ∈ E do   — consider edges targeting u_l
            if u_k ∉ U then                    — case: cross-unit edge
                E_U ← E_U ∪ {(u_k, u_f)}       — redirect to f
                tag((u, u_f)) ← tail f(x̄)      — args are the live variables x̄
            else                               — case: intra-unit edge
                E_U ← E_U ∪ {(u_k, u_l)}       — do not redirect to f
                E' ← E' ∪ E_U
```

If u_l is critical, then we define a function for it by giving it a fresh function name f and computing the set of live variables $\bar{x}$ at block l in P. The set $\bar{x}$ becomes the formal arguments to the function f.

We then insert a new node u_f into the normalized graph and insert the edges (u_r, u_f) and (u_f, u_l) into E_U as well as all the edges internal to U that do not target u_l. This creates the new function $f(\bar{x})$ whose body is defined by the control blocks of the unit U. Next, we consider critical edges of the form (u_k, u_l). If the edge is a cross-unit edge, i.e., $u_k \notin U$, then we replace it with an edge into u_f by inserting (u_k, u_f) into E_U and tagging the edge with a tail jump to the defined function f representing U. If the critical edge is an intra-unit edge, i.e., $u_k \in U$, we insert the edge into E_U, effectively leaving it intact. Although the algorithm only redirects critical edges Lemma 5.7.2 shows this is sufficient: all other edges in the graph are contained within a single unit and hence do not need to be redirected.

Synopsis. At a high level, the algorithm computes the program graph and the units of the graph. It processes each unit so that it can assign a function to each unit. If a unit's defining vertex is a function node, then no changes are required and all non-dominator-tree edges incident on the vertices of the unit are inserted.

If the units defining vertex is not a function node, then the algorithm creates a function for that unit by inserting a function node u_f between the defining node and the node. It then makes a function for each unit and replaces cross unit edges with tail-jump edge to this function. Since all cross unit have the defining node of a unit as their target, this replaces all cross-unit control transfers with tail jumps, ensuring that all control branches (except for function calls, tail or not) remain local.

Analysis of NORMALIZE**.** We state and prove some properties about the normalization algorithm and the (normal) programs it produces. The normalization algorithm uses a live variable analysis to determine the formal and actual arguments for each fresh function. We let $T_L(P)$ denote the time required for live variable analysis of a CL program P. The output of this analysis for program P is a function $\mathsf{live}(\cdot)$, where $\mathsf{live}(l)$ is the set variables which are live at (the start of) block $l \in P$. We let $M_L(P)$ denote the maximum number of live variables for any block in program P, i.e., $\max_{l \in P} |\mathsf{live}(l)|$. We assume that each variable, function name, and block label require one word to represent.

We relate the size of a CL program before and after normalization (Theorem 5.7.3) and bound the time required to perform normalization (Theorem 5.7.6). Using some simple technical development, we show that NORMALIZE preserves certain semantic properties of the graphs that it transforms (Theorem 5.7.6).

Theorem 5.7.3 (Size of Output Program)**.** *If CL program P has n blocks and $P' = \mathit{NORMALIZE}(P)$, then P' also has n blocks and at most n additional function definitions. Furthermore if it takes m words to represent P, then it takes $O(m + n \cdot M_L(P))$ words to represent P'.*

Proof. Observe that the normalization algorithm creates no new blocks—just new function nodes. Furthermore, since at most one function is created for each critical node, which is a block, the algorithm creates at most one new function for each block of P. Thus, the first bound follows.

For the second bound, note that since we create at most one new function for each block, we can name each function using the block label followed by a marker (stored in a word), thus requiring no more than $2n$ words. Since each fresh function has at most $M_L(P)$ arguments, representing each function signature requires $O(M_L(P))$ additional words (note that we create no new variable names). Similarly, each call to a new function requires $O(M_L(P))$ words to represent. Since the number of new functions and new calls is bounded by n, the total number of additional words needed for the new function signatures and the new function calls is bounded by $O(m + n \cdot M_L(P))$. □

Theorem 5.7.4 (Time for Normalization)**.** *If CL program* P *has* n *blocks then running* *NORMALIZE*(P) *takes* $O(n + n \cdot M_L(P) + T_L(P))$ *time.*

Proof. Computing the dominator tree takes linear time (Georgiadis and Tarjan, 2004). By definition, computing the set of live variables for each node takes $T_L(P)$ time. We show that the remaining work can be done in $O(n + n \cdot M_L(P))$ time. To process each unit, we check if its defining node is a function node. If so, we copy each incoming edge from the original program. If not, we create a fresh function node, copy non-critical edges, and process each incoming critical edge. Since each node has a constant out degree (at most two), the total number of edges considered per node is constant. Since each defining node has at most $M_L(P)$ live variables, it takes $O(M_L(P))$ time to create a fresh function. Replacing a critical edge with a tail jump edge requires creating a function call with at most $M_L(P)$ arguments, requiring $O(M_L(P))$ time. Thus, it takes $O(n + n \cdot M_L(P))$ time to process all the units. □

To reason about the correctness of normalization, we first define the notions of *safe control* and *program dilation*, which characterize two important guarantees that we prove about our normalization algorithm: that it does not introduce invalid control-flow paths (i.e., ones that jump into the middle of function bodies), and that it transforms control-flow paths in a manner that preserves the program's semantics.

The safe control property codifies a key invariant commonly used by many compilers, including ours. In particular, our compiler assumes that control flow always enters a function at its (unique) *entry node*.

Definition 5.7.1 (Safe control)**.** We say that a program P has *safe control* if for all edges $u_1 \to u_2$ in P:

1. $u_1 \to u_2$ is a tail-call edge to f and u_2 is the entry node for f, or else
2. $u_1 \to u_2$ is a goto edge, and u_1 and u_2 exist within the same function of P.

Intuitively, dilating a program simply consists of changing `goto` edges into equivalent tail call edges, each of which may or may not preserve safe control.

Definition 5.7.2 (Program dilation)**.** We define the *dilation* of program P_1 to program P_2 inductively using three cases:

Reflexivity. If $P_1 = P_2$, then P_1 dilates to P_2

Transitivity. If P_1 dilates to P_3, and P_3 dilates to P_2 then P_1 dilates to P_2.

Single-edge dilation. Suppose that the graphs of P_1 and P_2 are (V_1, E_1) and (V_2, E_2), respectively. Then we have all of the following:

1. $V_2 = V_1 \uplus \{u_f\}$, where u_f is an entry for a function f not in P_1.
2. $V_2 \uplus \{u_1 \to u_2\} = E_1 \uplus \{u_1 \to u_f, u_f \to u_2\}$.
3. The formals of function f are $\overline{x}$, the variables live upon entry to u_2.

4. The edge $u_1 \to u_f$ in P_2 is a tail call with arguments $\overline{x}$.

Lemma 5.7.5 (Program dilation preserves program meaning)**.** *If program* P_1 *is dilated to program* P_2*, then every control and data-flow path in* P_1 *has a semantically-equivalent (albeit longer) one in program* P_2*.*

Proof. By induction over the dilation of P_1 into P_2. The reflexive and transitive cases are immediate. For the single-edge dilation case, we consider the control and data-flow separately.

First consider the control flow. We note that the path/edge $u_1 \to u_2$ in P_1 is converted to a path $u_1 \to u_f \to u_2$ in P_2. Since node u_f unconditionally passes control to u_2, these two paths are control-flow equivalent.

Similarly, we next consider the data-flow. We note that node u_f simply accepts (as formal arguments) the data live at u_2, its target. Hence, the data flow is also equivalent. This completes the proof. □

Finally, we show that our normalization algorithm has several correctness guarantees: it preserves safe control, it preserves program meaning, and it preserves program running time (up to a constant factor of two).

Theorem 5.7.6 (Correctness of Normalization)**.** *Suppose that* P_1 *and* P_2 *are CL programs such that* $P_2 = \mathit{NORMALIZE}(P_1)$*.*

Dilated control: *Program* P_2 *is a dilation of program* P_1*.*

Safe control: *If program* P_1 *has safe control, then so does* P_2*.*

Control overhead: P_1 *contains edge* $u_1 \to u_2$*, then* P_2 *contains a path* $p = u_1 \rightsquigarrow u_2$ *such that* $|p| \leq 2$*.*

Proof. The first result (*dilated control*) follows because NORMALIZE is a program dilation: Each time it lifts a unit to the top-level, it dilates all the incoming edges to this unit, promoting local **goto**s into non-local **tail** jumps.

The second result (*safe control*) follows because we are dilating all cross-unit edges—the only edges that can participate in control that is not safe.

The third result (*control overhead*) follows since the algorithm only dilates certain edges, and only by a constant factor of two. □

Corollary (Normalization preserves heap and stack space)**.** *Say* P1 *is a CL program and* P2 $=$ *NORMALIZE*(P2)*. Then we have that* P1 *and* P2 *require the same amount of heap and stack space.*

Proof. We know the execution sequences of functions in P1 and P2 are the same, and in particular, we know that they contain the same nodes, and hence, same push, pop and and allocation operations. Because they contain the same allocation operations, the heap space used by each is the same. Because they contain the same push and pop operations, the stack space used by each is the same. □

5.8 Optimizations

We refine the basic compilation strategy given above with two optimizations. Each uses statically-analyzed information (Section 5.5).

Sharing trace nodes. Just as stack frame layout is a compiler feature meant to speed up the program by using a machine's registers and call stack more economically, trace node layout has an analogous purpose: to mitigate and amortize certain storage costs associated with the self-adjusting machine's execution trace.

Figure 5.9: The `Trace_action` module.

```
#module_begin Trace_action
#body
 type act_t = { trnd : trnd_t* }              -- first field: a trnd pointer
 val act_compare : act_t*, act_t* -> totord_t -- compare temporal ordering
 ------------------------------------------------------------------------
 totord_t act_compare( a, b ) {                -- temporal ordering
   if( a->trnd != b->trnd )                    -- same trace node?
     return Trnd.compare( a->trnd, b->trnd );  -- ..no : compare trace nodes
   else return ptr_compare( a, b );            -- ..yes: compare storage offsets
 }
 ------------------------------------------------------------------------
#module_end
```

The basic translation (Section 5.6) assigns each traced operation to a distinct trace node. Since each trace node brings some overhead, it is desirable if entire CL blocks (each which consists of zero, one or more traced operations) can share a common trace node. However, this optimization is complicated by a few issues.

First, how do we compare the temporal order of traced operations that share a single trace node, and hence, a single time stamp? Below, we address this question. Second, how do we avoid incrementally breaking apart the trace node representing the CL block when the control flow of execution changes? This can happen in one of two ways:

- The machine reexecutes the node's **update** point, but execution takes different branch through the CL block than before.
- The machine attempts to match and reuse a node's **memo** point, but this point resides somewhere within the CL block for the node (not at its start).

We currently avoid these scenarios by assigning a CL block a single trace node only when the following criteria are met: if it contains a **memo** point, then it appears first in the block; if it contains an **update** point, then the code guarded by the point (its local continuation) consits of straight-line code.

Trace actions. As described above, multiple traced operations can be grouped into trace nodes, which allows them to share resources, such as time stamps and allocation lists. This sharing is desirable from the standpoint of performance since time stamps are one source of performance overhead. However, by sharing a time stamp, we introduce a new problem: How do we order the THCs that occur at distinct times (distinct points during execution), but which reside in the same trace node?

To address this, we use a trick that converts the problem of recalling a temporal ordering (in execution time) into the problem of recalling a spatial ordering (in trace memory). The compiler lays out the footprint of each trace node so that its THCs are ordered monotonically with respect to their (statically-known) invocation order. This way, we get the best of both worlds. First, distinct THC invocations can share a common start time. Second, the execution order of distinct THCs can be recovered from the trace, even if they do not have distinct time stamps.

Figure 5.9 lists the simple declarations and code associated with this trick. To compare their closure records temporally, the THC author first ensures that their closure record (an extension of the record `act_t` type) contains a pointer back to the trace node. Such a backpointer is needed by any THC that uses supplemental information stored by the trace node, including temporal information about timestamps; this interface only requires that the author place this field first, which is an arbitrary choice made for presentation purposes here, but this choice should be uniform to all THC closures that are to be inter-comparable.

When it groups their THC closure records into a shared trace node, the compiler gives the THC author a guarantee: Namely, the THC programmer can compare the temporal orderings of their closure records even when these closure records share a common trace node. In the case that distinct two closures share the same trace node, the closures' addresses can be compared physically to order them.

Selective destination-passing style. The DPS conversion introduces extra IL code for **push** and **pop** expressions: an extra **alloc**, **update**, **memo**, and some **write**s and **read**s (Section 4.5). Since each of these expressions are traced, this can introduce considerable overhead for subcomputations that do not interact with changing data. In fact, without an **update** point, propagation over the trace of e will always yield the same return values (Lemma C.1.19, in Section C.1). Moreover, it is clear from the definition of store agnosticism (Section 4.4) that any computation without an **update** point is trivially CSA, hence, there is no need to DPS-convert it. By doing a conservative static analysis (viz., MU-analysis, Section 5.5), our compiler estimates whether each expression e appearing in the form **push** f **do** e can reach an **update** point during evaluation. If not, we do not apply the DPS conversion to **push** f **do** e. We refer to this refined transformation as *selective* DPS conversion.

5.9 Notes

In this section, we provide discussions on topics related to the design of our compiler.

Keyed allocation. With our choice of self-adjusting machine primitives, we decompose the keyed allocation primitive of earlier designs into smaller steps. We note that this decomposition is possible since, unlike earlier compiler approaches, our self-adjusting machine primitives include (stack-delimited) memoization points, and (stack-based) data-flow (i.e., returning values via the stack). The decomposition of an keyed allocation is structured as follows: (1) push the stack; (2) perform a memo point that guardes the local continuation (we shrink the local environment to contain only those variables that live; i.e., the values that we use for initializing the allocation are the keys of the keyed allocation); (3) allocate a fresh memory block; (4) initialize the block using the local environment; and (5) return the address of the allocation, popping the pushed stack frame. We

note that this decomposition of a keyed allocation includes three resources (stack, memory and trace), and hence cross-cuts the primitives of our self-adjusting machine model:

- It allocates a fresh block of non-local (modifiable) *memory*, or reuses an existing allocation from the *execution trace*.
- It uses a *memoization* point, which implicitly keys this allocation by the local (non-modifiable) state, determined and controlled by the programmer's use of the *stack*.
- It returns this allocation to the allocator, either fresh or reused, via the stack.

In this dissertation, we only explore monotonic reuse of past traces (and their allocations) within the basic self-adjusting machine design[6].

Semantics preservation. Self-adjusting programs have one extensional semantics (defining input and output relationships) and simultaneously encode two intensional semantics: The first of these semantics corresponds to the program's fresh evaluation (where work is always performed fresh). The seccond semantics corresponds with its incremental behavior, as defined jointly by the program and by the (general-purpose) change propagation algorithm (where past work is reused). Hence, compilation of self-adjusting programs is fundementally different than compilation of ordinary programs: The safe compilation of self-adjusting programs requires optimizations and other transformations to respect not only the from-scratch semantics of the program, but also the self-adjusting semantics. Informally speaking, it is okay to apply an optimization that always monotonically improves the incremental behavior of a program (or leaves it unaffected), but it is not okay to implicitly apply a transformation that may destabilize the self-adjusting behavior and inhibit reuse during change propagation. We do not formally prove our compiler observes this (informal) specification, we argue this point informally, below.

6. In Section 8.1, we explore a relaxation of this requirement: non-monotonic reuse.

First, our compiler performs some limited forms of function inlining. We note that if inlining functions alters the stack profile of the program, then it also generally changes the self-adjusting behavior of this program. To avoid doing this, our machine model separates stack operations from function calls, which are treated orthogonally. When code is inlined, the program text is copied and specialized, but the stack operations that delimit this code are retained.

Next, in a naive DPS conversion (one without consideration for memoization), new non-determinisitic choices are inserted into the program (viz., the choice of non-local destinations); we can use a memo point to guard the allocation of the destination locations, thus guarding these non-deterministic choices (Section 4.5). Without this inserted memo point, the use of fresh, non-determinisitically-chosen allocations will generally prevent reuse, since they never appear in the trace checkpoints of past computations.

Finally, our selective DPS transformation (Section 5.8) must be conservative when estimating the program's (dynamic) use of update points, lest it violates the compositional store agnostic property required of our model of change propagation (Section 4.4). We stress that this property hinges on the placement of **update** points, which have no extensional meaning.

Dominators. The dominator relation has common use in compilers that perform program analysis and optimization (Aho et al., 1986; Cytron et al., 1991; Fluet and Weeks, 2001). There are a number of asymptotically efficient algorithms for computing dominators (Lengauer and Tarjan, 1979; Georgiadis and Tarjan, 2004). In practice simple but asymptotically inefficient algorithms also perform reasonably well (Cooper, Harvey, and Kennedy, Cooper et al.). Our implementation uses the simple algorithm described in many compiler books, e.g., Muchnick (1997).

Tail-recursion. In the past, we used a trampolining approach to support tail calls in a portable manner (Section 7.3 surveys our past implementations). Several other proposals to supporting tail calls in C exists (Tarditi et al., 1992; Peyton Jones, 1992; Baker, 1995; Guy L. Steele, 1978). Peyton Jones summarizes some of these techniques (Peyton Jones, 1998) and discusses the tradeoffs. The primary advantage of trampolining is that it is fully portable; the disadvantage is its cost. Most recently, proper support for tail-recursion in C has become a much less esoteric feature. Increasingly, this support is something that folks outside the functional language community have come to appreciate and value (Lattner and Adve, 2004). Our current compilation approach assumes a C compiler backend that supports proper tail recursion.

Single-entry, single-exit regions. Single-entry, single-exit regions are subgraphs of a control-flow graph that have a single entry node and a single exit node. Because of this structure, the entry node dominates the region, while the exit node post-dominates the region. The functions of CL are almost single-entry, single-exit regions: their entry nodes are the blocks that define them; they may have multiple exit nodes: their returns and tail calls. The stack frames in IL and CL correspond to control being within certain single-entry, single-exit regions that are parenthesized by stack operations.

Intra-procedural normalization. While presented as a whole-program transformation, normalization can actually be easily performed on a per-function basis. This fact follows from it being based on the dominator tree of the program graph (Section 5.7), in which functions are always indepedent units.

CHAPTER 6

RUN-TIME SYSTEM

This chapter describes a run-time system design for self-adjusting machines. First, we present an overview of the run-time system's role in the design of the larger system, and in particular, its role vis-à-vis the compiler; we also review the challenges inherent to low-level memory and stack management (Section 6.1). Within the context of memory management, we give a detailed discussion of our run-time system's interaction in memory as implemented with an abstraction we call the *trace arena* (Section 6.2). Based on certain assumptions and restrictions, we describe the basic data structures and memory management techniques for an efficient run-time implementation of self-adjusting machines in C (Section 6.3). With detailed code listings, we describe the design and implementation of the two central abstractions in our run-time system design: trace nodes and self-adjusting machines (Sections 6.4 and 6.5). In doing so, we pay careful attention to how memory and stack resources are created and destroyed, within the context of subtle internal invariants and state changes.

Notation. In this chapter (and others) we write implementation code in C, but for clearer specification of module boundaries, we use a pseudocode syntax to write modules. In doing so, we alter the syntax for types and global declarations from that of C. Our psuedo syntax improves clarity of exposition for readers familar with ML-like languages (e.g., Haskell, OCaml, or SML). We can derive corresponding C code using a straightforward macro-based rewriting.

6.1 Overview

Our run-time system design gives an efficient, practical account of self-adjusting machines for low-level languages. However, the compiler and run-time system each have a role

to play in constructing these machines. Generally speaking, the compiler focuses on the static aspects whereas the run-time library focuses on the dynamic aspects: The role of the compiler is to build, given a fixed program text, a translated program that exposes the static structure of the program's trace to the run-time library interface; the role of the run-time library is to manage the dynamic structure of the trace, within the context of an executing self-adjusting machine. Below, we further describe these roles and their interrelationships.

Role of the compiler. from the viewpoint of the run-time system, we review the role of the compiler. The compiler extracts and organizes certain static information from the program, and in doing so, translates its primitives into uses of the run-time interface. Our run-time interface design is based around the abstraction of a *trace node*. First, the compiler seperates inner self-adjusting code that generates traces from the outer non-self-adjusting code that does not. Next, the compiler statically analyses the traced operations of the inner code and organizes the trace of these operations into trace nodes. Finally, it generates *trace node descriptors* that describe the static structure and behavior of each such trace node. It emits compiled code that combines the run-time interface with these descriptors.

The gensis of trace node descriptors by the compiler exploits the static structure of the program text, and uses this information to compose library extensions that implement each traced operation. By analysing how traced operations are used in the program, the compiler groups instances that have similar trace lifetimes (i.e., those operations with a shared dominator and post-dominator in the control-flow graph). Chapter 5 introduces trace hook closures (Section 5.3), trace nodes (Section 5.3) and trace node descriptors (Section 5.3) in the context of compilation.

We review these abstractions from the standpoint of run-time behavior. Each traced operation is implemented by a *trace hook closure* (THC), which augments the trace with

custom run-time behavior and storage. The new behavior is defined via functions that we refer to as *hooks*; the new storage is defined by a type that we refer to as a *closure* type. Trace nodes are formed by sequential compositions of trace hook closures. Each trace node consists of space for shared dynamic information (indicated by their THC attributes), a combination of the individual closure records, and the combined hook code. The hook code provides the behavior of redoing and undoing the traced operations as well as checking their consistency.

Role of the run-time library. The role of the run-time system (or *run-time* for short) is dual to that of the compiler: Whereas the compiler extracts the static structure of the self-adjusting machine (in terms of static trace node descriptors), the run-time manages the dynamic semantics of these machines (in terms of their traces, which consist of trace node instances). In doing so, it uses the static information furnished by compilation to give an efficient, operational account of self-adjusting computation in a low-level setting. Below, we review the central techniques of the self-adjusting computation stated in terms of trace nodes. See Section 1.6.2 for a general introduction to these concepts:

Trace generation creates trace node intances based on static descriptors.

Change propagation updates trace nodes that are inconsistent, via selective reexecution.

Memoization caches and reuses trace nodes from prior executions.

In particular, when a trace interval is reused via memoization, change propagation updates any inconsistencies that it contains. The change propagation algorithm consists of a loop that reexecutes (the update points of) inconsistent trace nodes, including their local continuations. Change propagation reexecutes the inconsistent trace nodes monotonically with respect to their original execution order.

Challenges. Generally, the efficient implementation of self-adjusting computation techniques in a run-time library requires special data structures and algorithms, to represent and update traces, respectively. We refer to Section 1.6.2 for a general introduction to existing techniques, which we briefly review above. Additional challenges that are unique to a low-level setting (as opposed to a high-level one) include the following:

Stack management is complicated by the need to conserve stack space. In particular, this management task is introduced by the existing techniques not being properly tail-recursive; the complexity of this task is compounded by the operational knot of self-adjusting computation, in which run-time techniques are applied recursively to the trace (Section 1.6.2).

Making existing run-time techniques tail-recursive requires changes to the approach, including the construction and representation of traces. For instance, nested intervals in the trace have distinct, staggered ending times; but when tail-recursive, these nested intervals share a common ending time. For intervals to share this ending point without backtracking through the trace to assign it[1], we must allocate this shared end time before it occurs—i.e., before beginning the execution of the nested intervals that it terminates. Sharing trace resources in this way is further complicated by issues surrounding memory management.

Memory management is complicated by the interaction of self-adjusting computation with low-level languages. In particular, this management task is introduced by the existing techniques relying on automatic garbage collection; the complexity of the task is compounded by the spatial knot of self-adjusting computation, in which run-time

1. It is interesting to note that the end-time assignment step shows up in our formal machine semantics, albeit only abstractly. Namely, the trace rewinding relation ($\circlearrowleft$) is used by the machine semantics whenever the end of a sub-trace is generated or encountered (Section 4.3). This relation models the process of backtracking through the prefix of the trace; one can also think of it as assigning end times to the intervals in this prefix that lack them. By pre-allocating an end time to be shared among the nested intervals in the trace prefix, our run-time design accomplishes this end-time assignment task in $O(1)$ time.

techniques give rise to heavily cyclic, imperative, higher-order memory structures (Section 1.6.2). In low-level languages, a module client and module implementation explictly coordinate to manage memory cooperatively. However, in the presence of traces and incremental reexecution, the client lacks the knowledge to determine when an allocation becomes unreachable.

Hence, this coordination must exploit the knowledge that is unique to the change propagation algorithm. Meanwhile, the change propagation algorithm sometimes has incomplete information, since outside of the trace, it is unaware of the structure of memory. Therefore, the change propagation algorithm's management of memory must in some cases be programmable, i.e., readily extensible by the programmer.

6.2 Memory management

In a low-level setting, we must coordinate dynamic memory management among the agents involved. In Section 5.2, we describe the agents involved from the viewpoint of compilation. Below, we describe how these agents are involved from the viewpoint of the run-time system (Section 6.2). In this context, we introduce memory management techniques to address different classes of inter-agent interaction. A central notion in the solution of this problem is that of a *trace arena*, which exploits temporal and causal structures inherent to the trace to guide the self-adjusting machine in reclaiming unused memory automatically during change propagation (Section 6.2). However, certain memory must exist outside the trace arena; we define this memory—and specifically modifiable memory in output-to-input feedback—and outline a unified plan for the incorporation of its memory management into an ecosystem of self-adjusting machines (Section 6.2).

Run-time agents and levels. Our run-time setting of self-adjusting machines consists of several interacting agents, organized by levels. Each level is represented by a different

body of code, which the compiler seperates, transforms and recombines. Ordered from innermost to outermost, these levels of agency consist of:

- the self-adjusting machine (the run-time library and its extensions, written in C);
- the inner self-adjusting program (written in CEAL, compiled to C);
- the outer non-self-adjusting program (written in CEAL, compiled to C); and,
- the foreign, non-self-adjusting program (written in C).

Since the outer and foreign non-self-adjusting programs are closely related (the former becomes the later through compilation), we focus on the interaction between the compiled inner code, the compiled outer code and the run-time library code implementing the self-adjusting machine. Each agent may have cause to allocate memory. Moreover, agents may share allocated memory with eachother. Our memory management protocol must systematize how this memory is managed.

Memory management within the self-adjusting machine. At the innermost level, within the self-adjusting machine implementation, our run-time library uses a standard approach to implement an internal, base memory manager (Section 6.3). The interface of this manager is consistent with that of the standard C library (`malloc` and `free`), which abstract away the details of managing a space of free (unallocated) addresses, but require clients to explicitly free memory before it becomes unreachable. When an agent allocates a block is using `malloc` and returns this resource to a client agent (or itself), the protocol stipulates that the client is responsible for calling `free`, lest they leak memory.

However, this convention is so complicated by tracing and incremental reexecution that it is only directly applicable in the setting of the run-time library itself, via internal knowledge of change propagation. Due to incremental tracing and reexecution, this protocol

becomes intractable for code at the inner level of the self-adjusting program and all outer levels that interact with this incremental, inner level.

Memory management within the trace arena. As mentioned above, the knowledge of when to reclaim memory can be spread between up to three agents. To formulate a robust management protocol, there are several cases to consider that vary depending on which agents are involved.

When only the machine and the inner program are involved, and when output-to-input feedback is absent, we say that a machine's allocation is confined to its *trace arena*. This arena consists of the trace, as an ordered timeline of trace nodes; it includes all inner data allocated by the inner code, subject to certain restrictions below.

Allocations in the trace arena have temporal positions in the trace with a special structure that is useful for memory management. At the conclusion of propagation, we exploit the consistency proven for the self-adjusting machine semantics: Because the trace is consistent with that of a full reevaluation, allocations whose trace nodes are revoked (undone) during change propagation must be garbage (Section 4.4). Based on this semantic reasoning, the trace arena is managed automatically by change propagation; two subcases below vary depending on whether the trace node can be reclaimed eagerly or not.

When the outer program is involved, or when the allocation participates in output-to-input feedback, an allocation escapes the boundary of the trace arena. These *escaped* allocations are powerful and complementary to trace arena: By escaping the trace arena, they increase the expressive power of the combined system. However, in doing so, they forgo memoization-based reuse (a la the trace arena); but in its place, they furnish the machine with fedback structures, where the output structure of one change propagation cycle becomes the input structure of the next cycle. Generally speaking, feedback is an interesting feature for incremental computations, as it can express incrementally-evolving

fixed-point computations, such as simulations and interactive systems (Demetrescu et al., 2011; Burckhardt et al., 2011).

In the presence of feedback, the structure of memory does not respect the temporal structure of the trace arena (Section 6.2). As a result, change propagation cannot collect the locations outside of the trace arena using the same semantic reasoning principles it uses to collect locations that remain within it. Instead, the machine manages this memory with the help of the programmer or library extension writer.

We review the three cases of management protocol in detail below:

Trace arena, eager subcase: when the allocation is hidden within the machine.

The location is internal to the machine (as defined by the base run-time library), or one of its traced operations (via a library extension). It does not escape into either the inner program nor the outer program.

In this case, the allocation is within the trace arena of memory, and we do not need to delay the reclamation of this allocation until the end of change propagation; it can be recycled (reclaimed and reallocated) in the middle of a change propagation cycle.

Trace arena, lazy subcase: when the allocation escapes from the machine into the inner program, but does not escape as feedback nor into the outer program.

In this case, the allocation is within the trace arena of memory, but until change propagation completes, there may remain references in the *suffix* of the arena; i.e., in the temporal future of the trace, but for that of the prior execution.

Consequently, such allocations must not be recycled until propagation completes, lest we create dangling pointers in the arena's suffix. After competing change propagation, from-scratch consistency states that all references will have been replaced.

Escaped case: when the allocation escapes the boundary of the trace arena.

Allocations escape the trace arena when they escape into either the outer program and/or into the modified input of the inner program, as fedback data.

When the outer program maintains pointers to these allocations, the outer program must participate in their management. Even when not referenced by the outer program, the possibility of locations being fedback requires that change propagation be given additional knowledge in order to soundly manage this memory.

While the trace arena can reference outer data, the trace arena should not be referenced from outside its boundary. This means that neither outer data nor fedback data can address structures within the trace arena, including reused allocations in the trace. A related restriction between inner and outer data applies to all implementations of self-adjusting computations, in both high- and low-level languages; we discuss it below (Section 6.2). Our run-time implementation exploits the restrictions of the trace arena to perform memory management during change propagation. From this memory management viewpoint, the outer data should not reference the trace arena, lest the trace changes and this leaves dangling pointers in the outer data.

Modifiable feedback. Feedback in a self-adjusting machine means that there are two notions of execution time, which are related but distinct: First, there is execution from the viewpoint of the inner program, which is consistent with that of a from-scratch run; and second, there is execution from the viewpoint of the outer program, which has a bigger frame of reference.

To see how these viewpoints differ, consider how a single trace arena allocation can appear differently at each level. From the inner viewpoint, allocations in the trace arena are always observably fresh, even if reused via memoization or keyed allocation. From the viewpoint of the outer program, reused allocations can be distinguished from truely fresh ones. In particular, by inspecting the memory of the trace arena before and after a cycle

of change propagation, the outer program can inspect the structure of the inner program's input and output and witness this reuse (lack of freshness). However, by contrast, from the vantage point of the inner program, this is not possible, due to from-scratch consistency (Section 4.4). These two distinct vantage points create a schism in the memory structure, which we describe below.

As a primary consquence of the trace arena's temporal structure, the *feedback restriction* says that the memory in the trace arena must not be fed back as modified input by the outer program. Though often not made explicit, this restriction is tacit in all prior work on self-adjusting computation[2].

From the viewpoint of the inner program, feedback in the trace arena can invalidate the causality of traced allocation effects, and when traces are acausal, they are not consistent with a from-scratch execution. In particular, for a trace to be from-scratch consistent, the trace of an allocation should always preceede the trace of its accesses (reads or writes). Specifically, to be causal, the traced access of an address must not occur before the trace of this address's allocation. However, when data allocated in one run is read as new input, such acausal patterns generally arise.

To prevent the casual problems above, fedback structures must be considered to be in distinct space from the trace arena. To preserve the consistency of the inner program, the interaction across this boundary must observe certain rules, listed below.

- Fedback memory does not reach the trace arena.
- Traced allocations of fedback memory are perpetually inconsistent.

The first rule says that fedback memory should be topologically distinct from the trace memory; i.e., that the transitive closure of fedback memory across pointer links is disjoint

2. Ley-Wild (2010) considers an extension to escape this limitation, based on the abstraction of *persistent modifiables*.

from the trace arena (though the converse need not hold). So, while the trace can reference fedback data, the fedback data should not (ever) reference the trace. This rule is needed for a certain kind of transitive consistency, i.e., consistency that is preserved across successive change propagation cycles and modifiable updates.

The second rule says that the traced allocation of fedback data must be freshed upon every cycle of change propagation. This behavior is demanded by the semantics of our self-adjusting machines: In the propagation of an allocation, the allocated location must not already be present in the current view of store (it must be fresh). We refer the reader to the freshness side condition of the reference semantics; rule **S.1** in Section 4.3. Meanwhile, by definition, any fedback data is reachable within every store view, and hence, its allocation in the trace cannot be further change-propagated and must be replaced.

For allocations within the trace arena, we sidestep this freshness requirement using a trick: Since the allocations within the arena are not fedback, we exploit this fact to simulate a fresh allocation with a reused one. In the absence of feedback and incoming pointers from the outer level, the inner level cannot witness this internal trick. In fact, the reuse of traced modifiable reference allocations is central to all current approaches of self-adjusting computation, in both high- and low-level languages. To achieve the combination of efficiency and semantic consistency (with respect to non-determinisitic, dynamic memory allocation), these techniques always rely on playing some version of this trick, and hence all introduce a schism in memory analogous to that of the trace arena.

Since fedback memory is distinct from the trace arena, the question arises as to how to best manage its reclamation. For some structures, this memory may not require complex reclamation reasoning, and can be manually coded in the outer level. To manage more complex memory of outer fedback structures that incrementally change, we propose using more self-adjusting machines. We leave the exploration of this idea to future work (Section 8.1).

Figure 6.1: The BASEMM signature.

```
#module_begin BASE_MEMORY_MANAGER
 val malloc : size_t        -> void*  -- malloc given size; no header is used
 val free   : size_t, void* -> void   -- reclaim allocation of given size
#module_end
```

6.3 Basic structures

We describe our base memory manager and the basic structures that efficiently implement the self-adjusting machine and its memory management of the trace arena.

Base memory manager. The run-time uses a custom base memory manager with a `malloc/free`-like interface, given in Figure 6.1. Unlike the standard implementation, however, this manager does not consume space to store a run-time header; to avoid this, `free` takes as an additional argument: the size of the block being reclaimed.

Among the static information that the compiler extracts for a trace node, it compiles a total size of a node instance, including the footprint for each THC closure record. This means that by storing a pointer to a trace node's static descriptor, we also store the size required to free its instances. Hence, the trace node descriptors store enough information to make additional allocation headers redundant and unnecessary. However, if the size of the allocation is unknown statically, this size must be stored by the allocating client and returned to the manager upon reclamation. In the trace arena, this size and a pointer to the dynamic allocation is stored in a linked structure associated with each trace node (the presence of which is indicated by the `BLK_ALLOC` attribute); the machine takes care to automatically reclaim these locations with their associated trace node. Outside the trace arena, the allocated size must be supplied by the client to later explicitly free the allocation.

Collection data structures. The run-time library uses the following collection-like data structures to implement self-adjusting machines and trace nodes. Here and throughout, *efficient* means either $O(\log n)$ or $O(1)$ time, depending on context.

- Sets, with and without order, implemented as lists and trees. These structures provide in-order, bi-directional iteration, efficient unordered insertion and efficient removal. Additionally, tree-based representations provide an efficient, $O(\log n)$-time search and ordered-insertion.

- Hash tables provide three basic operations, all in expected constant time: insert a new entry with a given hash value, find the first entry with a given hash value, delete an entry. To handle collisions gracefully, the hash tables use chaining (via lists or trees). To handle resizing, the tables grow and shrink in amortized $O(1)$ time (i.e., by doubling or halving in size to grow or shrink).

- Priority queues provide $O(\log n)$-time insertion, and $O(\log n)$-time removal of the minimum element.

- Order-maintenance data structures provide an implementation of time stamps. As such, we refer this structure as a timeline. It provides three operations, all in amortized constant time (Dietz and Sleator, 1987; Bender et al., 2002): insert a new timestamp after an existing one, delete a timestamp, compare the ordering of two inserted timestamps.

Usage patterns and memory management. In our internal implementation of the self-adjusting machine, the space required by the machine consists of two components: a small, fixed-size footprint (initially consisting of an uninitialized machine) and the trace arena of the machine. The trace arena, in turn, consists of a temporally-ordered, causally-connected

collection of trace nodes. The structures above are primarily used to furnish the trace arena of a self-adjusting machine with the data structures of an efficient implementation.

To sketch our memory management strategy of these internal structures, we review their usage patterns within the trace arena:

- For all insertions (into ordered lists, trees, hashtables, priority queues or the timeline), the space allocated for the insertion is of a small, statically-known size[3].
- Once removed, the structures do not maintain any reference to the inserted entry.

In all cases, these conditions allow the memory required by the insertion to be allocated in an unboxed fashion, within its trace node (again, assuming that the allocation is within the trace arena). Since all of the machine's internal state resides in the trace arena, this allows the automatic memory management of trace nodes via change propagation to handle the reclaimation of all the internal structures listed above. Moreover, it amortizes the run-time overhead of this management across potentially large blocks of code (viz. those that share a trace node).

6.4 Trace nodes

Figure 6.2 lists the interface of trace nodes exposed for the for library writers to implement traced operations (as trace hook closures) and the self-adjusting machine. It doubles as an interface used by the target of compilation. A number of features are optional (i.e., not furnished by every trace node); trace hook closures use attributes to indicate when certain features are required of their trace node (Section 5.3).

The functions `mk` and `memo_chkpt` are used in the target of compilation (Section 5.6). They instansiate and perform memo-lookups for trace nodes, respectively. Beyond the storage provided by a THC's fixed-size closure record, the trace node provides the `blk_alloc`

3. This rule excludes the hashtable's array of buckets; this array is the a minor exception to the rule.

Figure 6.2: The TRACE_NODE signature.

```
#module_begin TRACE_NODE
#imports
 type machine_t   -- self-adjusting machine
 type desc_t      -- trace node descriptor
 type update_pt_t -- update point of trace node

#exports
 type trnd_t      -- trace node
 type memo_pt_t   -- memo point of trace node
 type memotbl_t   -- memo table (a hash table of trace nodes)
 type dst_t       -- destination type (i.e., a pointer type)

 face Inner_target
  val mk            : desc_t* -> trnd_t* -- new node instance from descriptor
  val memo_chkpt    : trnd_t* -> trnd_t* -- requires: MEMO. does lookup / add

  val blk_alloc     : trnd_t*, size_t  -> Blks.nd_t* -- requires: BLK_ALLOC

  val compare       : trnd_t*, trnd_t* -> totord_t   -- requires: TIME
  val compare_time  : trnd_t*, Time.t* -> totord_t   -- requires: TIME

  val machine_of    : trnd_t* -> machine_t*   -- requires: MACHINE
  val start_time_of : trnd_t* -> Time.t*      -- requires: TIME
  val trnd_of_time  : Time.t* -> trnd_t*      -- requires: TIME
  val end_time_of   : trnd_t* -> Time.t*      -- requires: INTERVAL
  val memotbl_of    : trnd_t* -> memotbl_t*   -- requires: MEMOTBL
  val memo_pt_of    : trnd_t* -> memo_pt_t*   -- requires: MEMO
  val update_pt_of  : trnd_t* -> update_pt_t* -- requires: UPDATE

 face Machine
  val consis  : trnd_t* -> bool_t        -- compiled THC hook: consis
  val revinv  : trnd_t* -> dst_t         -- compiled THC hook: revinv
  val revoke  : trnd_t* -> void          -- compiled THC hook: revoke
  val has_att : trnd_t*, att_t -> bool_t -- test: true if node has given att
  val is_enq  : trnd_t*        -> bool_t -- test: true if node is enqueued
  val as_blks : trnd_t* -> Blks.nd_t*    -- restructure node into blks & sizes
#module_end
```

function to allocate memory within the trace arena, associated with the given trace node. By exploiting the structure of the trace arena, this memory is reclaimed when its trace node is revoked. The trace node supports this allocation operation when the BLK_ALLOC attribute is present; this attribute furnishes the trace node with storage space for an additional field to hold its list of dynamically-sized allocations.

For implementations of traced operations, trace nodes expose a number of additional (optional) features. The trace node can be enqueued for update via change propagation (`update`), and it can save and later provide access to its machine (`machine_of`). The `compare` function uses the the timeline data structure to efficiently compare start times of two trace nodes, within the temporal order of their common trace arena (it is an error to compare trace nodes from distinct machines). The `compare_time` function compares a trace node with a timestamp. The `start_time_of` function accesses the start timestamp of a node; the `trnd_of_time` function uses pointer arithmetic to recover the trace node that occurs at given timestamp. All features above involving the start time require the `TIME` attribute. The `end_time_of` function accesses the end timestamp of a node; it requires the `INTERVAL` attribute. Likewise, the `memotbl_of` function and the `update_pt_of` function access the memo table and update points of a trace node; they require the attributes `MEMOTBL` attribute and `UPDATE` attribute, respectively. The trace node provides a thin wrapper around the compiled hook code emitted for each trace descriptor. These hooks augment the behavior of self-adjusting machine, allowing it to test the trace node's consistency (the `consis` function), redo it (the `revinv` function) or undo it (the `revoke` function). The `as_blks` function coerces the trace node into a list of memory blocks that are ready to be enlisted for reclamation; it overwrites its original header with a new one that consists of its static size and pointers to the allocated blocks associated with the trace node, if any.

6.5 Self-adjusting machine

We describe concrete self-adjusting machines, by describing a C-based implementation.

Figure 6.3: The SELF_ADJUSTING_MACHINE signature.

```
#module_begin SELF_ADJUSTING_MACHINE
#imports
 type trnd_t      -- trace node
 type desc_t      -- trace node descriptor
 type memo_pt_t   -- memo point of trace node
 type memotbl_t   -- memo table (a hash table of trace nodes)
 type dst_t       -- destination type (i.e., a pointer type)

#exports
 type update_pt_t -- update point field of trace node (to schedule updates)
 type frame_t     -- stack frame of machine (with trace node cursor)
 type machine_t   -- self-adjusting machine (with root of trace arena)

 face Outer_target
  val create       : machine_t* -> void -- initialize an empty machine
  val set_machine  : machine_t* -> void -- set machine, a hidden global var
  val get_machine  : void -> machine_t* -- get machine

  val cycle        : void -> void -- change-propagate trace, one full cycle
  val destroy      : void -> void -- reclaims space used by internals & trace

 face Inner_target
  val ins_trnd     : trnd_t* -> void    -- insert new trace node; advance cursor
  val get_trnd     : void    -> trnd_t* -- get the current trace node at cursor

  val frame_push   : frame_t*, Time.t* -> void -- begins new trace interval
  val frame_pop    : void              -> void -- pops stack; ends interval

  val set_memotbl  : memotbl_t* -> void -- set memo table, a hidden stack var
  val get_memotbl  : void -> memotbl_t* -- get memo table

  val frame_memo   : trnd_t*, dst_t -> dst_t  -- reuse node's interval
  val sched_update : trnd_t*        -> void   -- schedule node for update

 face Machine
  val the_machine  : machine_t*         -- hidden global variable
  val load_root    : frame_t* -> void   -- load an empty stack w/ root of trace
  val revoke_until : Time.t*  -> void   -- undo from cursor to time
  val free_revoked : void     -> void   -- reclaim mem. blocks of revoked trace
  val frame_prop   : dst_t    -> dst_t  -- change-propagate remaining frame
  val frame_load   : frame_t*, trnd_t* -> void  -- push a reloaded frame
  val frame_jump   : frame_t*, trnd_t* -> void  -- overwrite frame with trnd
#module_end
```

6.5.1 *Interfaces*

Figure 6.3 gives the interface of the run-time machine, the SELF_ADJUSTING_MACHINE signature. We first discuss the types mentioned in the interface. The interface imports a

number of abstract types from other parts of the run-time system; these include types for the following: trace nodes, trace node descriptors, memo points, update points, memo tables and destinations. The interface exports types for stack frames and machines. These types are both of a small, fixed-size; they both reference the trace arena, but neither type provides storage for any portion of the trace arena itself, which conists entirely of trace nodes. Stack frames are stack-allocated; machines are allocated and managed explicitly by the outer program.

We divide the operations of the interface into three sub-interfaces: one for the target code of the outer level (`Outer_target`), one for the target code of the inner level (`Inner_target`), and one for the machine's implementation to use internally (`Machine`).

The outer target interface consists of operations for managing a machine from the outside; these operations include the following: Create a new, empty machine (`create`); set and get the current machine, a hidden global variable (`set_machine` and `get_machine`); change-propagate the machine's trace one full cycle (`cycle`); and, destroy the machine and reclaim the space for its trace arena (`destroy`).

The inner target interface consists of operations for managing a machine from the inside, during traced execution. All operations make reference to the *trace cursor*, an internal stack-allocated finger that points into the trace arena at the currently-focused trace node. The operations consist of the following: insert a new trace node and advance the trace cursor, placing the inserted node under focus (`ins_trnd`); get the trace node at the trace cursor position (`get_trnd`); push a new stack frame, creating a new interval for the frame in the trace (`frame_push`); pop the current stack frame, marking the end of the frame's interval in the trace (`frame_pop`); set the current memo table, a stack-allocated variable (`set_memotbl`); get the current memo table (`get_memotbl`); memo-match the frame of an existing trace node, and give up control to the machine to update this interval (`frame_memo`); and, finally, mark a trace node as inconsistent and schedule its update point

for reexecution via change propagation (`sched_update`).

As explained in Chapter 5, the compiler transforms the inner program into *destination-passing style* (DPS). To interoperate with this DPS-converted inner program while respecting its stack profile, the machine provides the `frame_memo` function in a compatible DPS form: The `frame_memo` function accepts and returns the current frame's destination, of `dst_t` type. This destination is a pointer whose memory content (viz., the frame's return result) is accessed by the inner program, but not the machine's internal mechanics, which never directly dereferences or stores it. Rather, the machine simply passes this pointer around as an opaque object.

The internal machine interface consists of machine operations necessary to implement the other operations listed above, as well as a global variable holding the current machine. Every operation either directly or indirectly references this global variable, with the notable exception of (`create`), which operates only on the machine given as its argument. The internal operations consist of the following: load the root trace interval into the (empty) stack of the machine (`load_root`); revoke trace nodes from the cursor to a given time, moving the cursor to the given time (`revoke_until`); free all previously-revoked blocks (`free_revoked`); change-propagate the current stack frame's interval in the trace (`frame_prop`); reload a stack frame from a trace node, pusing as the top-most frame (`frame_load`); and, jump within the current frame, moving the cursor ahead to the given trace node (`frame_jump`). As with `frame_memo` function, the `frame_prop` function operations in destination-passing style.

6.5.2 Internal structures

Figure 6.4 gives the internal structure of the run-time machine, specifying the internal fields of the machine and its stack frames. Priority queues consist of inconsistent trace nodes, ordered by their start times. We describe these further below.

Figure 6.4: The `Self_adj_machine` module.

```
#module Pq = [ Priority_queue ]    -- for enqueueing inconsistent trace nodes
  with key_t       := Trnd.t*      -- opt: don't store ptr; use ptr arithmetic
  and  key_compare := Trnd.compare -- ordering based on trace node start time

#type interval_t = { start : Time.t* ;   -- interval of timeline
                     end   : Time.t* ; } -- .. ( represents a subtree of trace )

#type frame_t = {        -- stack frame: the local state of the machine
  cur      : interval_t  ;  -- trace cursor; an interval
  trnd     : trnd_t*     ;  -- current trace node
  memotbl  : memotbl_t*  ;  -- current memo table
  outer_fr : frame_t*    ;  -- next outer frame (end time may be uninit'd)
  endtm_fr : frame_t*    ;  -- next outer frame with an init'd end time
}

#type machine_t = {       -- self-adjusting machine:
  fr        : frame_t*    ;  -- current stack frame / trace interval
  pq_past   : Pq.t*       ;  -- the priority queue holding nodes _before_ cur
  pq_future : Pq.t*       ;  -- the priority queue holding nodes _after_ cur
  revoked   : Blks.t      ;  -- blocks to be reclaimed when next cycle completes
  pqs       : Pq.t [2]    ;  -- space for double-buffering priority queues
  tr_root   : Time.t [2]  ;  -- outermost interval / root of trace tree
}
```

A stack frames consists of the following fields: the trace interval currently under focus (`cur`); the trace node currently udner focus (`trnd`); the current memo table (`memotbl`); the next outermost frame, whose end time may not yet be established (`outer_fr`); and the nearest outer frame whose end time is established (`endtm_fr`);

Machines internally consist the following fields: the topmost stack frame (`fr`); the priority queue of inconsistencies in the trace prefix (`pq_past`); the priority queue of inconsistencies in the trace suffix (`pq_future`); the set of revoked trace nodes that require reclamation (`revoked`); storage for the footprints of two empty priority queues (`pqs`); and, finally, storage for the time stamps for the root interval (`tr_root`);

Two queues of inconsistencies. As mentioned above, each machine uses two priority queues, whose content is temporally partitioned by the trace cursor. The `pq_past` field stores inconsistencies in the *trace prefix* (the past, temporally); the `pq_future` field stores

inconsistencies in the *trace suffix* (the future, temporally). In the absense of modifiable feedback, the trace prefix is always consistent (implying that `pq_past` is always empty); the suffix contains inconsistencies that have yet to be updated via change propagation. In the presense of modifiable feedback, however, the trace prefix may contain inconsistencies due to the feedback of modifiable write effects; these inconsistencies are stored for later in `pq_past`.

During change propagation, the `pq_past` field is inactive, and the `pq_future` field is active. That is to say, change propagation always dequeues from `pq_future`, and never from `pq_past`. In this way, we always change-propagate monotonically, moving the trace cursor into the future, through the trace suffix; however, when the change propagation cycle completes, we swap the role of the two queues. For this purpose, we use an extra level of indirection for their representation within the machine.

Trace nodes furnish the storage required to be inserted into a priority queue. They furnish this storage at a fixed-offset using a field of `Pq.nd_t` type, whose presence is indicated by the `UPDATE` attribute). As an optimization, from this fixed-offset storage within the trace node one can recover a pointer to the trace node itself using simple pointer arithmetic; this means that the priority queue nodes need not store an explicit backpointer to their corresponding trace nodes.

6.5.3 Outer-level target code

We describe the machine interface used by the target code of the outer level.

The `create` function. Figure 6.5 lists the code for creating a new, empty machine with no trace. Being empty, this machine has no stack, its priority queues are both empty, its list of revoked blocks is empty, and the root interval is empty. That is, the root interval consists of only the initial and final sentinel time stamps, whose storage is provided within

Figure 6.5: The `create` function.

```
#val create : machine_t* -> void  -- create empty machine; no trace

void create ( m ) {
  m->fr        = NULL;         -- empty means no stack
  m->pq_past   = m->pqs[0];    -- indirection allows swapping queues, after cycle
  m->pq_future = m->pqs[1];    -- ''
  Pq.init( & m->pq_past );     -- initially empty; no inconsistencies
  Pq.init( & m->pq_future );   -- ''
  Blks.init( & m->revoked );   -- initially empty; no reclaimable blocks
  Time.init( & tr_root[0] );   -- init first time stamp; an initial sentinel
  Time.insert( & tr_root[0],   -- insert final stamp; a final sentinel
               & tr_root[1] );
}
```

Figure 6.6: The `cycle` function.

```
#val cycle : void -> void    -- one complete cycle of change propagation

void cycle ( ) {
  frame_t root_fr ;          -- space for root stack frame
  load_root ( & root_fr );   -- load the root interval of trace
  frame_prop ( NULL );       -- propagate the entire trace
  frame_pop ( );             -- pop the root frame, doing final revocations
  free_revoked ( );          -- free all garbage (delayed revocations)
  swap_ptrs( & m->pq_past,   -- change propagation cycle ends with rebirth:
             & m->pq_future );  -- ..past queue becomes future queue
}
```

the machine. We create a timeline with the initial sentinel; we insert after this, the final sentinel. Initially, the trace contains no other timestamps.

The `cycle` function. Figure 6.6 lists the code for performing a full cycle of change propagation. These steps are abstracted by other operations, defined below. The machine stack-allocates space for the root stack frame. It loads the root interval of the trace onto (the empty) machine stack. The machine propagates the root interval. The machine frees all the revoked trace nodes whose reclaimation may have been delayed until the end of change propagation. Finally, it swaps the roles of the past and future queues.

Figure 6.7: The `destroy` function.

```
#val destroy : void -> void    -- reclaim the trace of the machine
-------------------------------------------------------------------
void destroy ( ) {
  frame_t root_fr ;            -- space for root stack frame
  load_root ( & root_fr );     -- load the root interval of trace
  revoke_until( root_fr.end ); -- revoke the entire trace
  free_revoked ( );            -- reclaim the entire trace
}
```

Figure 6.8: The `frame_push` function.

```
#val frame_push : frame_t*, Time.t* -> void -- push a fresh frame, new end time
-------------------------------------------------------------------------------
void frame_push ( fr, end_time ) {
  frame_t* outer    = get_machine()->fr; -- top frame becomes outer frame
  fr->outer_fr      = outer              -- link to outer frame
  get_machine()->fr = fr;                -- insert new frame
  fr->trnd          = outer->trnd;       -- copy
  fr->cur.start     = outer->cur.start;  -- copy
  fr->cur.end       = end_time;          -- allocated; not inserted
  fr->memotbl       = outer->memotbl;    -- copy
  fr->endtm_fr      = outer->endtm_fr;   -- copy
}
```

The `destroy` function. Figure 6.7 lists the code for reclaiming the trace arena of a machine. This step is the primary step in disposing of a machine; the other step consists of simply freeing the space for the fixed-size machine footprint (of `machine_t` type). The machine loads the root interval of the trace onto its (empty) stack. The machine revokes the entire root interval of the trace. Finally, the frees all the revoked trace nodes.

6.5.4 Inner-level target code

We describe the machine interface used by the target code of the inner level.

The `frame_push` function. Figure 6.8 gives pseudocode for pushing a new frame. The client provides pointers for storing two records: a stack frame and a timestamp. To allocate the frame, the calling client can use space stored on the C call stack. To allocate the

Figure 6.9: The `frame_pop` function.

```
#val frame_pop : void -> void       -- pop the top-most stack frame

void frame_pop ( ) {
  frame_t* fr = get_machine ()->fr; -- the topmost frame of stack
  assert( fr->outer_fr != NULL );   -- assert: top frame has an outer frame
  if( fr == fr->endtm_fr )          -- frame cases: reloaded or fresh
    revoke_until ( fr->cur.end );   -- case 1: reloaded; revoke interval suffix
  else                              -- case 2: fresh ..
    Time.insert ( fr->cur.start,      -- after current time insert ..
                  fr->cur.end ) ;     -- .. previously-uninitialized end time
  fr = ( m->fr = m->fr->outer_fr ); -- both cases: pop the frame ..
  fr->cur.start = fr->cur.end ;     -- .. and advance outer cursor
}
```

end timestamp, the client should find space in the trace arena, within a trace node. The machine initializes the provided frame space, inserting it as the new top-most stack frame. The machine saves a pointer to the timestamp provided by the client; once this pushed frame is later popped, this timestamp will mark the end time of the frame.

The machine delays insertion of the end time to respect the monotonicity of the trace timeline. Specifically, it delays the initialization and insertion of this end timestamp to the `frame_pop` function, when the pushed frame is popped from the machine stack. While the machine knows that the frame's end time will come in the future (after the cursor), until the frame is popped, the machine does not know the ordering of this end time with respect to the suffix of the trace. This uncertaintly is due to reuse: trace actions that follow the cursor may be reused while running within the pushed frame. Recall that the client indicates this reuse by passing control to the `frame_memo` function, when and if it matches a trace node. The insertion of this end time must follow the time stamps reused by such matches.

The `frame_pop` function. Figure 6.9 shows the detailed psuedocode for popping a frame. There are two cases to consider. In the first case, the frame is reloaded from an existing

trace via `frame_load`, e.g., from within change propagation. In the second case, the frame is fresh via `frame_push`, and its end time has yet to be initialized.

We differentiate between these cases by comparing the frame to its `endtm_fr` field; this field is setup to mark the end of the next outermost frame that has an end time that is initialized and inserted, in which case this field creates a reference to its own frame. These frames are reloaded, via `frame_load`.

If reloaded, then the end time already exists in the timeline. Moreover, there may be trace nodes still inserted between that point and the current one. As such, we revoke whatever remaining trace nodes were not reused, in the interval between the current time, and the end time of the reloaded frame. In the reloaded case, control returns to the change propagation algorithm (i.e., to either reload the next inconsistent frame, monotonically, or complete a full cycle of trace).

If fresh, we initialize and insert the end timestamp into the timeline. We advance time, moving the outer frame's cursor to the end of the popped frame.

In both cases, we unlink the machine from top-most frame, effectively popping it. The memory management protocol used for this space is that it is client-provided, and client-reclaimed. In our compiler's translation, we use the native C stack space for this purpose. Hence, we need not worry about explicitly reclaiming this frame memory.

The `frame_memo` function. Figure 6.10 lists the code for memo-matching an existing trace node. The interval of the matching trace node is reused as the continuation of the current frame. For this to make sense, the interval of the matching trace node should fall in the current interval on the stack. For illustrative purposes, we make this sanity check explicit: We test that the start time of the matching trace node occurs after the cursor; and, we test the end time of the matching trace node occurs before the nearest-enclosing end time that has been defined, i.e., the end time of the stack frame held in `endtm_fr`

Figure 6.10: The `frame_memo` function.

```
#val frame_memo : Trnd.t*, dst_t -> dst_t  -- memo-match the given trace node

void frame_memo ( m, d ) {
  { -- sanity check that matching node is valid, temporally:
    frame_t*   fr      = get_machine()->fr ;       -- the topmost stack frame
    Time.t*    m_start = Trnd.start_time_of ( m ); -- the start of the match
    Time.t*    m_end   = Trnd.end_time_of   ( m ); -- the end of the match
    totord_t ord_start = Time.compare( fr->cur.start ,      m_start );
    totord_t ord_end   = Time.compare( m_end, fr->endtm_fr->cur.end );
    assert (( ord_start == LESS || ord_start == EQUAL) &&  -- cursor <= match
            ( ord_end   == LESS || ord_end   == EQUAL) );  -- match  <= end time
  }
  revoke_until ( m_start );  -- revoke prefix; reuse begins at memo-match
  return frame_prop ( d );   -- change-propagation, in tail-position
}
```

field. These two checks ensure that reuse in the trace arena is monotone with respect to the ordering of time stamps.

To handle the match, the machine revokes the interval between the cursor and the start of the matching node (via `revoke_until`). Afterward, the trace cursor is left with the matching trace node under focus. In tail-position, the machine finishes processing the match by switching modes, into change propagation (via `frame_prop`).

6.5.5 Internal machine operations

We describe operations internal to the machine.

The `load_root` function. Figure 6.11 gives the code for loading the root interval of the trace into the machine stack. It is an error to load the root frame if the stack of the machine is not empty. The root frame reflects the stack of an empty machine, as created by `create`, except that the trace of the machine is not generally empty. Namely, the root frame has no trace node under focus, consists of the intitial and final sentinels of the timeline, has no current memo table. The root frame itself contains the outermost end time, and hence its `endtm_fr` field is a backpointer. There is no frame outside of the root frame.

Figure 6.11: The `load_root` function.

```
#val load_root : frame_t* -> void  -- load frame for root of trace
-------------------------------------------------------------------
void load_root ( fr ) {
  machine_t* m = get_machine ();
  assert( m->fr == NULL );         -- machine should have no stack
  m->fr          = & fr;           -- load the frame as root; initialize it:
  fr->trnd       = NULL;           -- .. no trace node
  fr->cur->start = m->tr_root [0]; -- .. start is initial sentinel
  fr->cur->end   = m->tr_root [1]; -- .. end   is final   sentinel
  fr->memotbl  = NULL;             -- .. no memo table
  fr->endtm_fr = & fr;             -- .. a backpointer to root frame
  fr->outer_fr = NULL;             -- .. no outer frame
}
```

Figure 6.12: The `revoke_until` function.

```
#val revoke_until : Time.t* -> void           -- revoke trace until given time
-------------------------------------------------------------------
void revoke_until ( stop ) {
  machine_t* m = get_machine ();              -- get current machine
  Time.t* cur = m->frame->cur.start;          -- get its trace cursor
  while( cur != stop ) {                      -- loop until given end time
    Trnd.t* trnd    = Trnd.of_time( cur );    -- trace node at time (ptr arith.)
    cur             = Time.next( cur );       -- move cursor
    Trnd.revoke( trnd );                      -- revoke trace node / time stamp
    Blks.nd_t* blks = Trnd.as_blks( trnd );   -- reorganize into list of blocks
    if( Trnd.has_att( trnd , DELAY_FREE )     -- cases: free trace node now?
        || Trnd.is_enq( trnd ) )              -- .. check if node is enqueued
         Blks.insert ( & m->revoked, blks ) ; -- case: delay, references remain
    else Blks.free ( blks );                  -- case: now, no references remain
  }                                           -- loop finished:
  m->frame->cur = stop;                       -- move cursor to stopping time
}
```

The `revoke_until` function. Figure 6.12 gives the code for revoking an interval of the trace. The function consists of a loop that iterates until the cursor reaches the specified stopping time (`stop`), provided by the client. For each time within the revocation interval, the machine retrieves the associated trace node (`trnd`). As an optimization, the trace node with a particular start time can be recovered using pointer arithmetic; this start time occupies storage at a fixed-offset within its associated trace node, as specified by the `TIME` attribute. The machine revokes the trace node and removes it from the timeline (via `Trnd`

`.revoke`), and restructures the trace node into a list of the memory blocks that had stored it (via `Trnd.as_blks`). Whether the machine can free the trace node immediately depends on whether they may exist remaining references to it. The machine checks for this by checking if the trace node is currently enqueued in either priority queue (see below), or if it carries the `DELAY_FREE` attribute; in either case, it enlists the blocks for reclaimation later. Otherwise, the machine reclaims the trace node immediately using the base memory manager (via `Blks.free`). This loop continues until the machine reaches the stopping time. Once the stopping time is reached, the machine moves its trace cursor to this stopping time.

We chooose a slightly more complex revocation logic in favor of a simpler priority queue implementation—i.e., one that does not require removing entries other than the minimum. Rather than remove trace nodes from the queue when they are revoked, we delay this step, and wait until they are popped as the minimum. Towards this end, when the machine revokes a trace node (via `Trnd.revoke`), the machine overwrites and nullifies the node's timestamps. When nullified, these timestamps do not contain dangling pointers, and they compare minimally against all non-null time stamps. This ensures that they still respect the structure of the priority queue (no reorderings of priority queue entries are required for this nullification step).

The `frame_prop` function. Figure 6.13 lists the code for propagating the current trace interval. To be supportive of tail-recursive patterns, the function takes a destination argument of `dst_t` type, which consists of a pointer to the current frame's return results, stored within the trace arena. The `frame_prop` function conists of a loop that iterates until either no inconsistencies remain, or they remain but all are beyond the the current frame. Within the body of the loop, the machine dequeues the next inconsistent trace node. There are several cases and subcases to check.

Figure 6.13: The `frame_prop` function.

```
#val frame_prop : dst_t -> dst_t                    -- change-propagate frame

void frame_prop ( d ) {
   machine_t* m    = get_machine ();              -- current machine
   Time.t*    end  = m->cur->end ;                -- end of current frame
   trnd_t*    trnd = Pq.peek_min( m->pq_future ); -- next trace inconsistency
   while( Trnd.start_time_of( trnd ) &&           -- does it exist in trace?
          Trnd.compare_time( trnd, end ) == LESS ) { -- is it in current frame?
     Pq.dequeue_min( machine->pq_future );        -- pop it from queue.
     if( Trnd.start_time_of( trnd ) != NULL &&    -- still live?
         ! Trnd.consis( trnd ) ) {                -- still inconsistent?
         if( Trnd.end_time_of( trnd ) == end ) {  -- cases: in tail-pos.?
             frame_jump ( machine->fr, trnd );    -- case: yes, in tail-pos.
             return Trnd.revinv ( trnd );         -- reexecute the rest
         } else {                                 -- case: no, non-tail pos.
           frame_t fr;                            -- rspace to reload frame
           frame_load ( &fr, trnd );              -- ..reload traced interval
           Trnd.revinv ( trnd );                  -- reexecute the interval
           frame_pop ( );                         -- pop the reloaded frame
   } } }                                          -- no more inconsistencies
   return d ;                                     -- return the destination
}
```

In the first case, the trace node is already revoked; this is indicated by it having a nullified start time (via `Trnd.revoke`). In this case, the trace node does not belong to the current interval; rather, it was revoked from the trace prefix, but left in the queue since the dangling reference from the queue still remained. Now popped, the trace node is no longer referenced by the queue, and is only referenced by the machines list of revocations (i.e., the list held by its `revoked` field).

In the next case, the trace node is no longer inconsistent; this can occur when invalidating operations make a node inconsistent, but then before its reexecution, other operations counterbalance the invalidating operations. The machine checks for this case by querying and checking the trace node's consistency (via `Trnnd.consis`). This function is conservative; for many traced operations it always returns `FALSE`. Unlike the abstract machine semantics of change propagation (Section 4.3), the run-time system's machine does not traverse the trace blindly, checking every traced operation for its consistency. Rather, it

only performs consistency checks on trace nodes that have been explicitly scheduled for an update.

In the final case, the trace node is still in the trace arena and moreover, it is also within the interval of the current frame. Two subcases remain to be distinguished. In the first reexecution subcase, the inconsistent trace node is in tail-position within the interval of the current frame; the machine checks this by checking its end time, comparing against the end time of the current frame. If equal, then change propagation should proceed by passing control in tail-position, lest it grows the stack and consequently alters the stack profile of its self-adjusting program. To perform the control transfer, the machine places the inconsistent trace node under focus (via `frame_jump`); it reexecutes this trace node in tail position (via `Trnd.revinv`).

In this subcase, the machine does not return destination `d` directly, but only indirectly. To do so, it relies on two facts: that the trace node `trnd` is already closed on the value of `d`, and that `Trnd.revinv` will return this value as its result. As it turns out, these facts follow from the reexecuted program being compositionally store agnostic (CSA), a property that we enforce by statically transforming it during compilation (Section 5.1).

In the second reexecution subcase, the inconsistent trace node is not in tail-position within the interval of the current farm; rather, the node is within a strictly-smaller subinterval, with a distinct end time, occuring before the end of the current interval. To propagate this case, we perform a similar jumping step as in the tail-position case above, except that we do so in a distinct stack frame that we push (via `frame_load`). Within this reloaded stack frame, we revoke, then invoke the trace node to make it consistent (via `Trnd.revinv`); this generally consists of also reexecuting the local contination of the trace node. When this completes, the reloaded stack frame is popped (via `frame_pop`), revoking whatever trace suffix within local continuation was not reused. As in the first subcase, in this subcase we also rely on the program being CSA: The machine does not propagate the return

Figure 6.14: The `frame_load` function.

```
#val frame_load : frame_t*, Trnd.t* -> void    -- reload frame from trace node

void frame_load ( fr, trnd ) {
  fr->endtm_fr  = fr ;                         -- indicates frame is reloaded
  machine_t* m  = get_machine ();              -- get current machine
  fr->outer_fr  = m->fr;                       -- save outer frame
  m->fr         = fr;                          -- push reloaded frame
  frame_jump ( fr , trnd );                    -- set the trnd info in frame
}
```

Figure 6.15: The `frame_jump` function.

```
#val frame_jump : frame_t*, Trnd.t* -> void    -- set frame info from trace node

void frame_jump ( fr, trnd ) {
  fr->trnd      = trnd;                        -- restore saved trace node
  fr->cur.start = Trnd.get_start_of( trnd );   -- .. saved start time
  fr->cur.end   = Trnd.get_end_of( trnd );     -- .. saved end time
  fr->memotbl   = Trnd.get_memotbl_of( trnd ); -- .. saved memo table
}
```

value of `Trnd.revinv`, but rather ignores it. The CSA structure of the program guarantees that this return value will not be affected by changes to modifiable state, and hence can be safely ignored in this way during change propagation.

The `frame_load` function. Figure 6.14 lists the code for pushing a stack frame that consists of reloaded information from the trace. The arguments consist of a trace node and temporal space for a frame to push; the allocation of this frame argument follows the same guidelines as that of `frame_push` (Section 6.5.4). Unlike in `frame_push`, however, all reloaded frames indicate this status by containing a backpointer in their `endtm_fr` field: This indicates that the reloaded frame ends the reloaded interval, just as with the root frame, as pushed by `load_root` (Section 6.5.5). To reload the other stack fields from the trace node, the machine uses `frame_jump`, defined below.

The `frame_jump` function. Figure 6.15 lists the code for fast-forwarding the trace cursor to a given trace node. Frst, The machine sets the trace node to the given one. Next, the machine sets the current interval to be that of the trace node, and the current memotable to be that of the trace node.

6.5.6 Cost analysis

The inner-level operations performed by the run-time implementation of the self-adjusting machine consist of those functions discussed in Sections 6.5.4, 6.5.3 and 6.5.5. These correspond to the concrete realizations of the transition rules described in Chapter 4, with the cost model described in Section 4.6; in this model, reevaluation steps have unit cost, undoing steps have unit cost, and change propagation steps have zero cost. Based on this intensional cost model, it is straightforward to verify that these operations can be implemented with (ammortized, expected) $O(1)$ overhead in space, and in time, as defined within the run-time implementation of the machine semantics (Sections 6.5.4, 6.5.3 and 6.5.5). To argue this, we assume that various measures are constant sized. In particular, we assume all of the following:

- The size of the priority queue is constant bounded.
- The number of effects on any one modifiable reference is constant bounded.
- The number of memoization keys in any one equivalence class is constant bounded.

By using standard structures to implement the basic data structures described in Section 6.3, one can implement each inner-level primitive with (ammortized, expected) $O(1)$ overhead in space and time.

Sometimes one or more of the assumptions above do not hold, i.e., when certain sizes are not constants. In these cases, the overhead factor of $O(1)$ is increased to be a logarithmic factor $O(\log n)$, where n is the non-constant size or number. We explain how to

apply this reasoning, case by case. First, as the number of inconsistent operations in the trace grows, the operations on the priority queue also become a logrithmic factor. Second, for modifiable operations (viz., reading and writing) on modifiable references that can be written arbitrarily many times, additional logrithmic factors are associated with searching (e.g., within a splay tree) for the value associated with the current time. Finally, non-unique memoization keys can be distinguished and ordered by their temporal position in the execution trace (e.g., by a splay tree).

CHAPTER 7

EMPIRICAL EVALUATION

We study the performance of our current and past implementations. We evaluate our current compiler for self-adjusting computation, based on the theory and practice of self-adjusting machines. We empirically evaluate our current system by considering a number of benchmarks written in the current version of CEAL, and compiled with our current compiler. Our experiments are very encouraging, showing that our latest approach still yields asymptotic speedups, resulting in orders of magnitude speedups in practice; it does this while incurring only moderate overhead (in space and time) when not reusing past computations. We evaluate our compiler and runtime optimizations (Section 5.8), showing that they improve performance of both from-scratch evaluation as well as of change propagation. Comparisons with previous work, including our own and the DeltaML language shows that our approach performs competitively. Finally, we briefly survey our current and past implementions Section 7.3.

7.1 Experimental setup

We describe the setup of our empirical experiemnts, including the measurements, benchmarks, compilation targets and machine configuration used.

Measurements. For each benchmark described below, we consider *self-adjusting* and *conventional* versions. All versions are derived from a single source program. We generate the conventional version by replacing modifiable references with conventional references. Unlike modifiable references, the operations on conventional references are not traced—the code that uses them acts as ordinary, non-self-adjusting C code. The resulting conventional versions are essentially the same as the static C code that a programmer would write for that benchmark.

For each self-adjusting benchmark we measure the time required for propagating a small modification by using a special *test mutator*. Invoked after an initial run of the self-adjusting version, the test mutator performs two modifications for each element of the input: it deletes the element and performs change propagation, it inserts the element back and performs change propagation. We report the average time for a modification as the total running time of the test mutator divided by the number of updates performed (usually two times the input size).

For each benchmark we measure the from-scratch running time of the conventional and the self-adjusting versions; we define the *overhead* as the ratio of the latter to the former. The overhead measures the slowdown caused by the dependence-tracking techniques employed by self-adjusting computation. We measure the *speedup* for a benchmark as the ratio of the from-scratch running time of the conventional version divided by the average modification time computed by running the test mutator.

Benchmarks. Our benchmarks consist of expression tree evaluation, some list primitives, two sorting algorithms and several computational geometry algorithms.

For each benchmark, we measure the *from-scratch time*, the time to run the benchmark from-scratch on a particular input, and the average *update time*, the average time required by change propagation to update the output after inserting or deleting an element from its input. We compute this average by using the test mutator described above: It iterates over the initial input, deletes each input element, updates the output by change propagation, re-inserts the element and updates the output by change propagation.

List primitives. These benchmarks include filter, map, reverse, minimum (integer comparison), and sum (integer addition), and the sorting algorithms quicksort (string comparison) and mergesort (string comparison). We generate lists of n (uniformly) random integers as input for the list primitives. For filter and map, the filtering and mapping function consists

of a small number of integer operations (three integer divisions and two integer additions). For sorting algorithms, we generate lists of n (uniformly) random, 32-character strings. We implement each list benchmark mentioned above by using an external C library for lists, which our compiler links against the self-adjusting code after compilation.

Computational geometry. These benchmarks include quickhull, diameter, and distance; quickhull computes the convex hull of a point set using the standard quickhull algorithm; diameter computes the diameter, i.e., the maximum distance between any two points of a point set; distance computes the minimum distance between two sets of points. Our implementations of diameter and distance use quickhull to compute first the convex hull and then compute the diameter and the distance of the points on the hull (the furthest away points lie on the convex hull). For quickhull and distance, input points are selected from a uniform distribution over the unit square in $\mathbb{R}^2$. For distance, we select equal numbers of points from two non-overlapping unit squares in $\mathbb{R}^2$. We represent real numbers with double-precision floating-point numbers. As with the list benchmarks, each computational geometry benchmark uses an external C library; in this case, the external library provides geometric primitives for creating points and lines, and computing simple properties about them (e.g., line-point distance).

Benchmark targets. In order to study the effectiveness of the compiler and runtime optimizations (Section 5.8), for each benchmark we generate several *targets*. Each target is the result of choosing to use some subset of our optimizations. Table 7.1 lists and describes each target that we consider. Before measuring the performance of these targets, we use regression tests to verify that their self-adjusting semantics are consistent with conventional (non-self-adjusting) versions. These tests empirically verify our consistency theorem (Theorem 4.4.3).

Target	Optimizations used
no-opt	Our optimizations are not applied. (however, C target code is still optimized by "`gcc -O3`")
share	Like no-opt, but with trace node sharing (Section 5.8).
seldps	Like no-opt, but with selective DPS transformation (Section 5.8).
opt	Both seldps and share are used.

Table 7.1: Targets and their optimizations (Section 5.8).

Benchmark	N	Conv (sec)	FS (sec)	Overhead (FS / Conv)	Ave. Update (sec)	Speed-up (Conv / AU)
exptrees	10^6	0.18	1.53	8.5	1.3×10^{-5}	1.4×10^4
map	10^6	0.10	1.87	18.4	3.4×10^{-6}	3.0×10^4
reverse	10^6	0.10	1.81	18.4	2.6×10^{-6}	3.8×10^4
filter	10^6	0.13	1.42	10.7	2.7×10^{-6}	4.9×10^4
sum	10^6	0.14	1.35	9.6	9.3×10^{-5}	1.5×10^3
minimum	10^6	0.18	1.36	7.7	1.3×10^{-5}	1.4×10^4
quicksort	10^5	0.40	3.30	8.2	5.8×10^{-4}	6.9×10^2
mergesort	10^5	0.74	5.31	7.2	9.5×10^{-4}	7.8×10^2
quickhull	10^5	0.26	0.97	3.7	1.2×10^{-4}	2.2×10^3
diameter	10^5	0.26	0.90	3.4	1.5×10^{-4}	1.8×10^3
distance	10^5	0.24	0.81	3.4	3.0×10^{-4}	7.9×10^2

Table 7.2: Summary of benchmark results, opt targets

Machine configuration. We used a machine with four eight-core Intel Xeon X7550 processors running at 2.0GHz. Though our machine has many cores, all of our experiments are sequential. Each core has 32Kb each of L1 instruction and data cache and 256 Kb of L2 cache. Each processor has an 18Mb L3 cache that is shared by all eight cores. The system has 1Tb of RAM. It runs Debian Linux (kernel version 2.6.32.22.1.amd64-smp).

As our standard C compiler, we use GNU GCC, version 4.3.2. We compile all targets using `gcc -O3` after translation to C. When evaluating our compiler optimizations, we vary only those of our compiler, not those of GCC. That is to say, we use `gcc -O3` uniformly for all compilation targets (e.g., even for code that our compiler does not optimize).

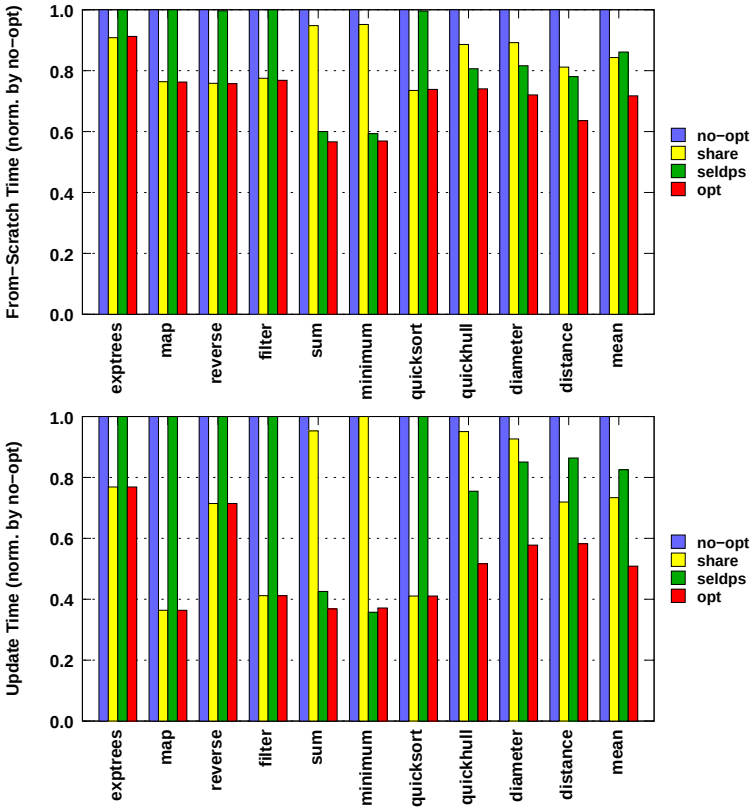

Figure 7.1: Comparison of benchmark targets.

7.2 Experimental results

Given the setup above, we discuss the results from our experimental evaluation.

Summary of results. Table 7.2 summarizes the self-adjusting performance of the benchmarks by comparing them to conventional, non-self-adjusting C code. From left to right, the columns show the benchmark name, the input size we considered (N), the time to run the conventional (non-self-adjusting) version (Conv), the from-scratch time of the self-adjusting version (FS), the *preprocessing overhead* associated with the self-adjusting version (Overhead is the ratio FS/Conv), the average update time for the self-adjusting version (Ave. Update) and the *speed-up* gained by using change propagation to update the output versus rerunning the conventional version (Speed-up is the ratio Conv/Ave. Update). All

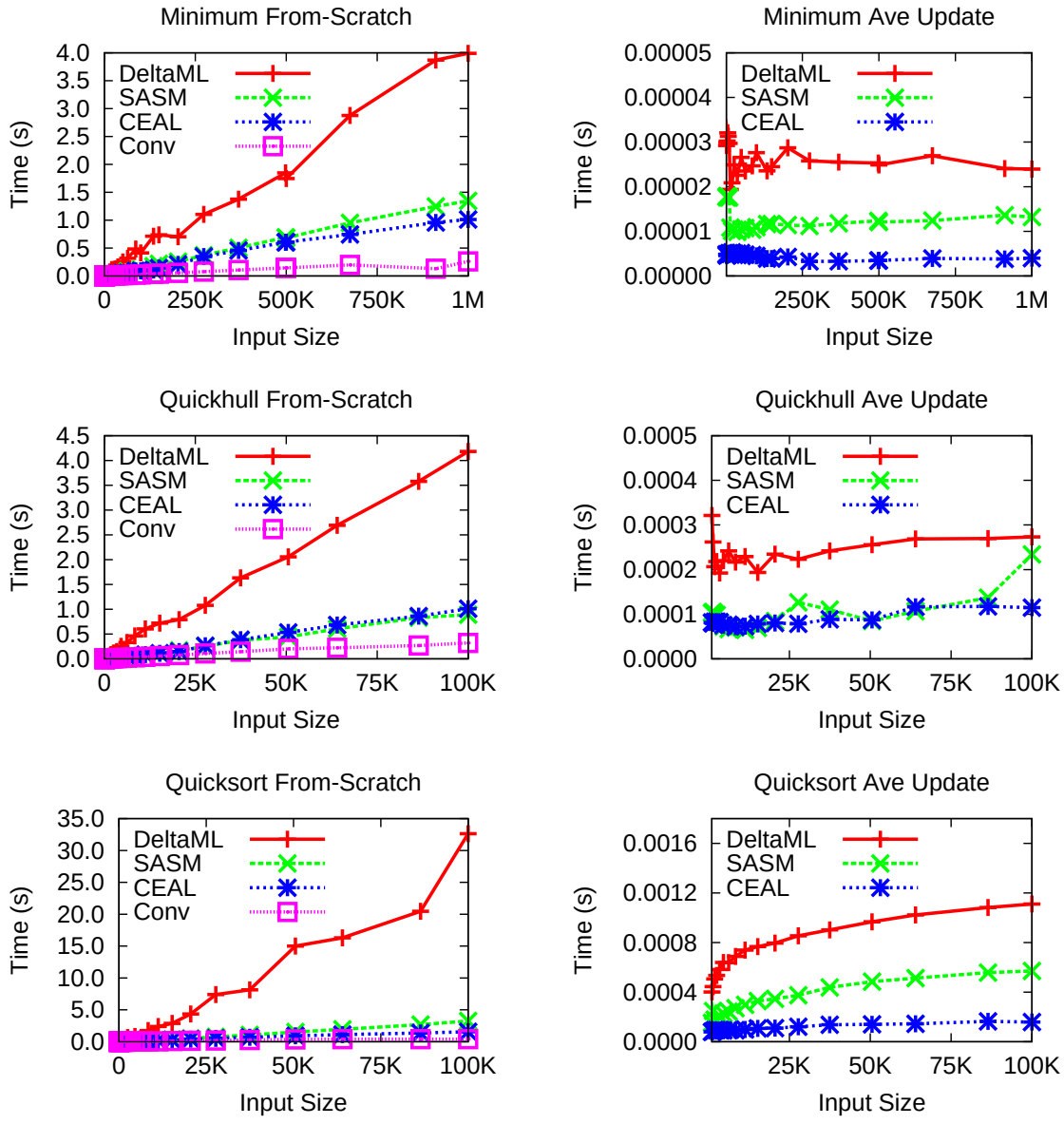

Figure 7.2: DeltaML versus stages two and three (“CEAL” and “SASM”).

reported times are in seconds. For the self-adjusting versions, we use the optimized (opt) target of each benchmark.

The preprocessing overheads of most benchmarks are less than a factor of ten; for simpler list primitives benchmarks, this overhead is about 18 or less. However, even at these only moderate input sizes (viz. 10^5 and 10^6), the self-adjusting versions deliver speed-ups of two, three or four orders of magnitude. Moreover, as we illustrate in the example above (Section 7.2), these speedups increase with input size.

Comparison of optimizations. Figure 7.1 compares our targets' from-scratch running time and average update time. Each bar is normalized to the no-opt target. The rightmost column in each bar graph shows the mean over all benchmarks. To estimate the efficacy of an optimization X, we can compare target no-opt with the target where X is turned on.

In the mean, the fully optimized targets (opt) are nearly 30% faster from-scratch, and nearly 50% faster during automatic updates (via change propagation), when compared to the unoptimized versions (no-opt). These results demonstrate that our optimizations, while conceptually straightforward, are also practically effective: they significantly improve the performance of the self-adjusting targets, especially during change propagation.

Comparison to past work. To illustrate how our implementation compares with past systems, Figure 7.2 gives representative examples. It compares the from-scratch and average update times for three self-adjusting benchmarks across three different implementations: one in DeltaML (Ley-Wild et al., 2008a), one in the original CEAL compiler (Hammer et al., 2009) and the opt target of our current implementation. To distinguish results of the current system from those of the original CEAL work—we use the labels "SASM" and "CEAL", respectively. These labels consist are contractions of the original paper titles in which these systems were first published, in Hammer et al. (2009) and Hammer et al.

(2011), respectively. In the from-scratch graphs, we also compare with the conventional (non-self-adjusting) C implementations of each benchmark (labeled Conv).

The three benchmarks shown (viz. minimum, quickhull and quicksort) illustrate a general trend. First, in from-scratch runs, the SASM implementations are only slightly slower than that of CEAL, while the DeltaML implementations are considerably slower than both. For instance, in the case of quicksort, the DeltaML implementation is a factor of ten slower than our own. While updating the computation via change propagation, the performance of the SASM implementations lies somewhere between that of DeltaML and CEAL, with CEAL consistently being either faster than the others, or comparable to SASM. Although not reported here, we obtain similar results with other benchmarks.

7.3 Implementation

In this section we describe the implementation of our self-adjusting language system, based the theory and practice of self-adjusting machines, as described in this dissertation. Compared with earlier systems that we built, our current implementation is highly programmable: It can be readily used as is, or extended with new abstractions. In addition to change propagation, the *core programming primitives* of our current stack-based design consist of only **update** points, **memo** points and stack operations **push** and **pop**. Other self-adjusting primitives (such as those for modifiable references) are provided as library extensions. Similarly, other traceable data types (Acar et al., 2010) are instances of library extensions, as are other memory management protocols, including keyed allocation (Section 5.9).

Below, we give details about our current implementation, details about its current limitations, and details about its lineage in past systems that we built. Our system's evolution consists of two previous implementations, whose usability and choices of core primitives differ greatly from those of the present system.

Implementation details. Our current implementation consists of a compiler and an associated runtime system, as outlined in Chapters 5 and 6. After compiling and optimizing IL, our implementation translates it to C, which we can compile using any standard C compiler (currently, we use GNU GCC). In all, our compiler consists of a 11k line extension to CIL and our runtime system consists of about 8k lines of C code.

As a front-end to IL, we support a C-like source language, CEAL (Chapter 3). We use CIL (Necula et al., 2002) to parse CEAL source into a control-flow graph representation. To bridge the gap between this representation and IL, we utilize a known relationship between static single assignment (SSA) form and lexically-scoped, functional programming (Appel, 1998b). Before this translation, we move CEAL variables to the heap if either they are globally-scoped, aliased by a pointer (via CEAL address-of operator, `&`), or are larger than a single machine word. When such variables come into scope, we allocate space for them in the heap (via **alloc**); for global variables, this allocation only happens once, at the start of execution. As part of the translation to IL, we automatically place **update** points before each **read** (or consecutive sequence of **read**s). Though in principle we can automatically place **memo** points anywhere, we currently leave their placement to the programmer by providing a **memo** keyword in CEAL; this keyword can be used as a CEAL statement, as well as a wrapper around arbitrary CEAL expressions.

Current limitations. Our source language CEAL is more restricted than C in a few ways, though most of these restrictions are merely for technical reasons and could be solved with further compiler engineering. First, while CEAL programs may use variadic functions provided by external libraries (e.g., `printf`), CEAL does not currently support the definition of new variadic functions. Furthermore, function argument and return types must be scalar (pointer or base types) and not composite types (`struct` and `union` types). Removing these restrictions may pose engineering challenges, but should not require a fundamental change to our approach.

Second, our CEAL front-end assumes that the program's memory accesses are word aligned. This assumption greatly simplifies the translation of pointer dereferencing and assignment in CEAL into the **read** and **write** instructions in IL, respectively. To lift this restriction, we could dynamically check the alignment of each pointer before doing the access, and decompose those accesses that are not word-aligned into one (or two) that are.

Third, as a more fundamental challenge, CEAL does not currently support features of C that change the stack discipline of the language, such as `setjmp/longjmp`. In C, these functions are often used to mimic the control operators and/or exception handling found in higher-level languages. Supporting these features is beyond the scope of this dissertation, but remains of interest for future work.

Finally, to improve efficiency, programs written in CEAL can be mixed with foreign C code (e.g., from a standard C library). Since foreign C code is not traced, it allows those parts of the program to run faster, as they do not incur the tracing overhead that would otherwise be incurred within CEAL. However, mixing of CEAL and foreign C code results in a programming setting that is not generally sound, and contains potential pitfalls. In particular, in this setting, meta-level programs must adhere to the correct usage restrictions defined in Section 6.2.

Evolution of the system. We compare our past implementations' designs qualitatively based on:

- their choice of core programming primitives—i.e., the primitives that are needed to program basic self-adjusting computations (e.g., common benchmarks); and,
- the compiler support (or lack thereof) for working with these primitives.

We categorize our past implementations into three *stages*, which successively build on one another:

Stage one: our first run-time system gives a proof-of-concept for implementing tracing, change propagation and memory management of C programs (Section 7.3). The design and evaluation of this system was published in Hammer and Acar (2008).

Stage two: our first compiler design gives analysis and compilation techniques making the primitives of stage one less burdensome for the programmer (Section 7.3). The design and evaluation of this system was published in Hammer et al. (2009).

Stage three: guided by our abstract machine model, our current compiler and run-time system design are more programmable (usable and extensible) than earlier systems. (Section 7.3). The design and evaluation of this system was published in Hammer et al. (2011).

We survey details of this evolution below.

First run-time system. In the first run-time system, the core primitives consist of keyed allocation of non-modifiable memory, modifiable reference primitives (allocation, writing and reading).

This run-time system provides no compiler support. Rather, we program it using a collection of C macros that provide the programming primitives listed above, as well as forms for declaring static information and performing traced function calls.

Through these macros, the static structure of the trace is declared manually by the programmer, including the layout of each type of closure record. Keyed allocation also requires explicit, top-level declaration of initialization code, and is treated as a core feature, as are the operations on modifiable references (such as their allocation, reading and writing). Modifiables may only be accessed at certain function call sites, which are specially annotated; these call sites serve as memoization points. The programmer reorganizes the program so that function-call sites correspond to modifiable accesses, and all live data is explicitly made into plumbing. Moreover, functions must not return a value.

From this first evoluationary step, basic elements of the run-time system have survived essentially unchanged into the current system, most notably, our implementations of certain data structures (Deitz-Sleator order-maintanence data structure, priority queues, etc.). Other parts have changed significantly between each version, most notably the representation of the trace, its closures, its memoization points and its recorded operations on modifiable data.

First compiler design. At this stage, we provide the same primitives as the first run-time system (Section 7.3), but through compilation techniques, we provide these in a more usable form. At a conceptual level, the run-time of this stage adapts the design of the run-time system above, except that we use certain compilation steps (static analyses and transformations) to automatically produce the declarations that the programmer would otherwise produce by hand. As a result, a programmer using this compiler need not manually declare run-time closure layout, or decompose their program based on where modifiables are accessed.

Afforded by having a compiler, we experiment with simple optimizations in the trace representation, such as attempting to group reads that occur simultaneously when the programmer uses certain forms (viz., when reads occur in the argument positions of a multi-argument function call). Using the compiler, we also specialize the trace structure using static information (e.g., the statically-known arity, type and size of traced closures).

Current system design. The evolution from the first compiler to the current one changes the core programming primitives, offers a more programmable interface, and creates opportunities for further static analysis and optimizations. At this stage in design, we are semantically guided by a new machine model (self-adjusting machines). These machines suggest different core primitives, namely, the core programming primitives listed above

(viz. **memo**,**update**,**push** and **pop**). These core primitives are supplemented with library extensions which provide, among other abstractions, that of modifible references.

CHAPTER 8

CONCLUSION

At a mechanical level, a self-adjusting computation is a spatial structure that represents a dynamically-evolving computation. It evolves according to general-purpose rules of self-adjustment (i.e., change propagation) along with the rules of a specific program. By evolving its computational structure, the program responds efficiently to its incrementally-changing environment.

At a low-level of abstraction, we address the following questions:

- What does this self-adjusting structure consist of?
- By what operational rules does the structure adjust?

That is to say, we give the parts and mechanics that define a low-level self-adjusting computation. In doing so, we take into account the machine resources that are hidden at higher levels of abstraction, but exposed at a low level of abstraction. In particular, our self-adjusting machines provide an account of how change propagation interacts with stack-based evaluation and automatic memory management of the heap.

We describe a sound abstract machine semantics for self-adjusting computation based on a low-level intermediate language. We implemented this language by presenting compilation and optimization techniques, including a C-like front end. Our experiments confirm that the self-adjusting programs produced with our approach often perform asymptotically faster than full reevaluation, resulting in orders of magnitude speedups in practice. We also confirmed that our approach is highly-competitive, often using machine resources much more efficiently than competing approaches in high-level languages.

8.1 Future directions

We discuss the future direction of self-adjusting machines.

Incremental memory management outside the trace. For structures that reside outside of the trace arena, self-adjusting machines can conceivably mark and collect memory by using a programmer-provided reference-counting traversal of heap data. Within this traversal, the programmer counts references using an abstraction provided by a run-time library extension (Section 5.3). This (automatically incrementalized) traversal can be repeated whenever the programmer wishes to collect garbage. When and if a traversed block becomes "untraversed" (i.e., unreached by the traversal), the library extension marks this block as garbage; if it is reused, it is unmarked; at the conclusion of change propagation, all marked blocks are collected.

Cycles in memory are traditionally problematic for collectors based on reference counting. By virtue of having the programmer specify how to traverse their data structures, orphaned cycles can always be detected: Once the trace associated with traversing the cycle is absent in the trace (i.e., when this traversal is revoked), the cycle can be collected.

Opting-out of update points. As described in (Section 6.5.4), memo points are "opt-in": programmers explicitly insert them where they want. A future language feature consists of letting programmers "opt-out" of update points in the following way: Programmers would insert an explicit update point and forgo update points in the remainder of the local continuation using a (hypothetical) [1] *freeze* statement primitive: $S_1; \texttt{freeze}; S_2$. Extensionally, the meaning of this statement sequence is equivalent to $S_1; S_2$; intensionally, the freeze primitive *guards* statement S_2 with an update point, and ensures that if any update points within S_2 be redirected to the inserted update point rather than any explicitly or implicitly-determined ones. Intuitively, the `freeze` primitive has the intensional semantics of associating with its local continuation a frozen (coarse-grained) trace that cannot be incrementally modified. Since the trace of S_2 is frozen, time can only be distinguished

1. This feature is not implemented as a primitive, but is a conceivable extension to the current compiler and language via the extensible run-time framework that we present in Chapter 6.

as being either before, during or after S_2; within S_2, no incremental events are possible. Hence, the fine-grained temporal structure of S_2 can be collapsed to a point, treating all the effects of S_2 as occuring within an instant of time, rather than an interval of distinct time instants. The flattening of time intervals into instants suggests interesting space-time performance tradeoffs when these intervals are bound by a small factor of the total input size (e.g., a constant or logarithmic factor). We leave this primitive for future work.

Semantics preservation. As pointed out in Section 5.9, compilation of self-adjusting programs is different than compilation of ordinary programs. Ordinary programs typically have one intensional semantics realized by their compilers. In self-adjusting computation, however, the compiler realizes two intensional semantics: One for conventional evaluation, and one for incremental, self-adjusting evaluation (interposed with a run-time library for self-adjusting computation). The difference between these is witnessed by a cost model. In future work, compilers for self-adjusting machines should prove (either on paper, or via verified compilation) that the machines that they produce always *improve* the costs of the program (with respect to the cost model) compared with the costs of the program as written by the programmer.

Non-monotonic reuse. Our abstraction of a THC can be used to implement keyed allocations that steal from the trace in a non-monotonic way, as optimizations, when desired. Previous work explores non-monotonic reuse of keyed allocations (Ley-Wild et al., 2008b; Hammer and Acar, 2008). In a pure setting, Ley-Wild et al. (2011) explores non-monotonic reuse of larger pieces of execution traces (i.e., traces containing only a limited class of effects on modifiable references, which always have a corresponding pure execution). In future work, we suggest an additional approach to non-monotone reuse: decompose incremental tasks into *multiple* self-adjusting machines.

Self-adjusting machine ecosystems. Complex incremental systems should not be limited to being embodied by a single monolithic self-adjusting machine. Rather they should consist of an ecosystem of machines that communicate with one another through a shared, external space, e.g., global memory, external to all machines and owned by the outer program (or by the collective of machines, or some combination of both, etc.). Since each machine maintains its trace internally, each machine can reuse its trace independently of the reuse within other machines. It is conceivable that each machine in the ecosystem could spawn other machines, allowing dynamic (external) inter-dependency structures of incrementally-updating machines; juxtaposed with this inter-machine dependency structure, each machine has its own intra-machine dependencies, and its own internal trace. The concept of a self-adjusting machine ecosystem, wherein incremental machine agents cooperate and react to one another, suggests a role for self-adjusting machines in the pursuit creating larger, compositional self-adjusting ecosystems.

References

Abadi, M., B. W. Lampson, and J.-J. Lévy (1996). Analysis and caching of dependencies. In *International Conference on Functional Programming*, pp. 83–91.

Abbott, M., T. Altenkirch, C. McBride, and N. Ghani (2004). D for data: Differentiating data structures. *Fundam. Inf. 65*(1-2), 1–28.

Acar, U. A. (2005, May). *Self-Adjusting Computation*. Ph. D. thesis, Department of Computer Science, Carnegie Mellon University.

Acar, U. A., A. Ahmed, and M. Blume (2008). Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*.

Acar, U. A., G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan (2006). A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science 148*(2).

Acar, U. A., G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan (2009). An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys. 32*(1), 3:1–53.

Acar, U. A., G. E. Blelloch, M. Blume, and K. Tangwongsan (2006). An experimental analysis of self-adjusting computation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*.

Acar, U. A., G. E. Blelloch, and R. Harper (2002). Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pp. 247–259.

Acar, U. A., G. E. Blelloch, and R. Harper (2004, November). Adaptive memoization. Technical Report CMU-CS-03-208, Department of Computer Science, Carnegie Mellon University.

Acar, U. A., G. E. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Türkoğlu (2010). Traceable data types for self-adjusting computation. In *Programming Language Design and Implementation*.

Acar, U. A., G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu (2008, September). Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*.

Acar, U. A., M. Blume, and J. Donham (2007). A consistent semantics of self-adjusting computation. In *European Symposium on Programming*.

Acar, U. A., A. Cotter, B. Hudson, and D. Türkoğlu (2010). Dynamic well-spaced point sets. In *Symposium on Computational Geometry*.

Acar, U. A., A. Cotter, B. Hudson, and D. Türkoğlu (2011). Parallelism in dynamic well-spaced point sets. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*.

Acar, U. A., A. Ihler, R. Mettu, and O. Sümer (2007). Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*.

Aftandilian, E. E. and S. Z. Guyer (2009). Gc assertions: using the garbage collector to check heap properties. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, New York, NY, USA, pp. 235–244. ACM.

Agarwal, P. K., L. J. Guibas, H. Edelsbrunner, J. Erickson, M. Isard, S. Har-Peled, J. Hershberger, C. Jensen, L. Kavraki, P. Koehl, M. Lin, D. Manocha, D. Metaxas, B. Mirtich, D. Mount, S. Muthukrishnan, D. Pai, E. Sacks, J. Snoeyink, S. Suri, and O. Wolfson (2002). Algorithmic issues in modeling motion. *ACM Comput. Surv. 34*(4), 550–572.

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Appel, A. W. (1991). *Compiling with Continuations*. Cambridge University Press.

Appel, A. W. (1998a). *Modern compiler implementation in ML*. Cambridge University Press.

Appel, A. W. (1998b). SSA is functional programming. *SIGPLAN Notices 33*(4), 17–20.

Appel, A. W. and D. B. MacQueen (1991). Standard ML of New Jersey. In *PLILP*, pp. 1–13.

Baker, H. G. (1978). List processing in real-time on a serial computer. *Communications of the ACM 21*(4), 280–94. Also AI Laboratory Working Paper 139, 1977.

Baker, H. G. (1995). Cons should not cons its arguments, part II: Cheney on the MTA. *SIGPLAN Not. 30*(9), 17–20.

Bellman, R. (1957). *Dynamic Programming*. Princeton Univ. Press.

Bender, M. A., R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito (2002). Two simplified algorithms for maintaining order in a list. In *Lecture Notes in Computer Science*, pp. 152–164.

Berger, E. D., B. G. Zorn, and K. S. McKinley (2002, November). Reconsidering custom memory allocation. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA. ACM Press.

Bhatotia, P., A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini (2011). Incoop: MapReduce for incremental computations. In *ACM Symposium on Cloud Computing*.

Brodal, G. S. and R. Jacob (2002). Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pp. 617–626.

Burckhardt, S. and D. Leijen (2011). Semantics of concurrent revisions. In *ESOP*, pp. 116–135.

Burckhardt, S., D. Leijen, C. Sadowski, J. Yi, and T. Ball (2011). Two for the price of one: A model for parallel and incremental computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Carlsson, M. (2002). Monads for incremental computing. In *International Conference on Functional Programming*, pp. 26–35.

Chen, Y., J. Dunfield, and U. A. Acar (2012, Jun). Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. to appear.

Chen, Y., J. Dunfield, M. A. Hammer, and U. A. Acar (2011, September). Implicit self-adjusting computation for purely functional programs. In *Int'l Conference on Functional Programming (ICFP '11)*, pp. 129–141.

Chiang, Y.-J. and R. Tamassia (1992). Dynamic algorithms in computational geometry. *Proceedings of the IEEE 80*(9), 1412–1434.

Cohen, J. (1981, September). Garbage collection of linked data structures. *Computing Surveys 13(3)*, 341–367.

Collins, G. E. (1960, December). A method for overlapping and erasure of lists. *Communications of the ACM 3*(12), 655–657.

Cooper, K. D., T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm.

Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*(4), 451–490.

Danvy, O. and J. Hatcliff (1993). On the transformation between direct and continuation semantics. In *Proceedings of the Ninth Conference on Mathematical Foundations of Programming Semantics (MFPS)*, pp. 627–648.

Demers, A., T. Reps, and T. Teitelbaum (1981). Incremental evaluation of attribute grammars with application to syntax-directed editors. In *Principles of Programming Languages*, pp. 105–116.

Demetrescu, C., I. Finocchi, and A. Ribichini (2011). Reactive imperative programming with dataflow constraints. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Dietz, P. F. and D. D. Sleator (1987). Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pp. 365–372.

Diwan, A., D. Tarditi, and J. E. B. Moss (1993, December). Memory subsystem performance of programs with intensive heap allocation. Technical Report CMU-CS-93-227, Computer Science Department, Carnegie-Mellon University. Also appears as Fox Memorandum CMU-CS-FOX-93-07.

Efremidis, S. G., K. A. Mughal, and J. H. Reppy (1993, December). Attribute grammars in ML. Technical report.

Eppstein, D., Z. Galil, and G. F. Italiano (1999). Dynamic graph algorithms. In M. J. Atallah (Ed.), *Algorithms and Theory of Computation Handbook*, Chapter 8. CRC Press.

Field, J. and T. Teitelbaum (1990). Incremental reduction in the lambda calculus. In *ACM Conf. LISP and Functional Programming*, pp. 307–322.

Flanagan, C., A. Sabry, B. Duba, and M. Felleisen (1993). The essence of compiling with continuations. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 237–247.

Fluet, M. and S. Weeks (2001). Contification using dominators. In *Proceedings of the International Conference on Functional Programming*, pp. 2–13.

Friedman, D. P., C. Haynes, and E. Kohlbecker (1984, January). Programming with continuations. In P. Pepper (Ed.), *Program Transformation and Programming Environments*, Berlin, Heidelberg, pp. 263–274. Springer-Verlag.

Georgiadis, L. and R. E. Tarjan (2004). Finding dominators revisited: extended abstract. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 869–878.

Grossman, D., G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney (2002, June). Region-based memory management in Cyclone. See PLDI (2002), pp. 282–293.

Guo, P. J. and D. Engler (2011). Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, New York, NY, USA, pp. 287–297. ACM.

Guy L. Steele, J. (1978). Rabbit: A compiler for scheme. Technical report, Cambridge, MA, USA.

Hallenberg, N., M. Elsman, and M. Tofte (2002, June). Combining region inference and garbage collection. See PLDI (2002), pp. 141–152.

Hammer, M. and U. A. Acar (2008). Memory management for self-adjusting computation. In *International Symposium on Memory Management*, pp. 51–60.

Hammer, M., U. A. Acar, M. Rajagopalan, and A. Ghuloum (2007). A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*.

Hammer, M., G. Neis, Y. Chen, and U. A. Acar (2011). Self-adjusting stack machines. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Hammer, M. A., U. A. Acar, and Y. Chen (2009). CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Heydon, A., R. Levin, and Y. Yu (2000). Caching function calls using precise dependencies. In *Programming Language Design and Implementation*, pp. 311–320.

Hoover, R. (1987, May). *Incremental Graph Evaluation*. Ph. D. thesis, Department of Computer Science, Cornell University.

Hudak, P., S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson (1992). Report on the programming language haskell, a non-strict, purely functional language. *SIGPLAN Notices 27*(5), 1–.

Huet, G. (1997). The zipper. *Journal of Functional Programming 7*(5), 549–554.

Ingerman, P. Z. (1961, January). Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM 4*(1), 55–58.

Jacob, R. (2002). *Dynamic Planar Convex Hull*. Ph. D. thesis, Department of Computer Science, University of Aarhus.

Jones, R. E. (1996, July). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester: Wiley. With a chapter on Distributed Garbage Collection by R. Lins.

Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming Language, Second Edition*. Englewood Cliffs, New Jersey: Prentice-Hall.

Knuth, D. E. (1968, June). Semantics of context-free languages. *Theory of Computing Systems 2*(2), 127–145.

Lattner, C. and V. Adve (2004, March). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

Lengauer, T. and R. E. Tarjan (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems 1*(1), 121–141.

Ley-Wild, R. (2010, October). *Programmable Self-Adjusting Computation*. Ph. D. thesis, Computer Science Department, Carnegie Mellon University.

Ley-Wild, R., U. A. Acar, and G. Blelloch (2011). Non-monotonic self-adjusting computation. Submitted for publication.

Ley-Wild, R., U. A. Acar, and M. Fluet (2008, July). A cost semantics for self-adjusting computation. Technical Report CMU-CS-08-141, Department of Computer Science, Carnegie Mellon University.

Ley-Wild, R., U. A. Acar, and M. Fluet (2009). A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*.

Ley-Wild, R., M. Fluet, and U. A. Acar (2008a). Compiling self-adjusting programs with continuations. In *Int'l Conference on Functional Programming*.

Ley-Wild, R., M. Fluet, and U. A. Acar (2008b). Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming (ICFP)*.

Lieberman, H., C. Hewitt, and D. Hillis (1983). A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 419–429.

Liu, Y. A. and T. Teitelbaum (1995). Systematic derivation of incremental programs. *Sci. Comput. Program. 24*(1), 1–39.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM 3*, 184–195.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (Eds.), *Computer Programming and Formal Systems*, pp. 33–70. North-Holland, Amsterdam.

Michie, D. (1968). "Memo" functions and machine learning. *Nature 218*, 19–22.

Milner, R., M. Tofte, and R. Harper (1990a). *Definition of standard ML*. MIT Press.

Milner, R., M. Tofte, and R. Harper (1990b). *Definition of standard ML*. MIT Press.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Necula, G. C., S. McPeak, S. P. Rahul, and W. Weimer (2002). CIL: Intermediate language and tools for analysis and transformation of C programs. In *Int'l Conference on Compiler Construction*, pp. 213–228.

Norrish, M. (1998). *C formalised in HOL*. Ph. D. thesis, University of Cambridge.

O'Neill, M. E. and F. W. Burton (2006). Smarter garbage collection with simplifiers. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, New York, NY, USA, pp. 19–30. ACM.

Overmars, M. H. and J. van Leeuwen (1981). Maintenance of configurations in the plane. *Journal of Computer and System Sciences 23*, 166–204.

Peyton Jones, S. (1998). C--: A portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*. Springer Verlag.

Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming 2*, 127–202.

Pilato, M. (2004). *Version Control With Subversion*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

PLDI (2002, June). *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin. ACM Press.

Pugh, W. and T. Teitelbaum (1989). Incremental computation via function caching. In *Principles of Programming Languages*, pp. 315–328.

Ramalingam, G. and T. Reps (1993). A categorized bibliography on incremental computation. In *Principles of Programming Languages*, pp. 502–510.

Ramsey, N. and J. Dias (2006). An applicative control-flow graph based on Huet's zipper. *Electron. Notes Theor. Comput. Sci. 148*(2), 105–126.

Reichenbach, C., N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer (2010, October). What can the gc compute efficiently?: a language for heap assertions at gc time. *SIGPLAN Not. 45*(10), 256–269.

Reps, T. (1982a, August). *Generating Language-Based Environments*. Ph. D. thesis, Department of Computer Science, Cornell University.

Reps, T. (1982b). Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pp. 169–176.

Reps, T. W. and T. Teitelbaum (1984). The synthesizer generator. In *Software Development Environments (SDE)*, pp. 42–48.

Reps, T. W. and T. Teitelbaum (1989). *The synthesizer generator - a system for constructing language-based editors*. Texts and monographs in computer science. Springer.

Rudiak-Gould, B., A. Mycroft, and S. L. P. Jones (2006). Haskell is not not ml. In *ESOP*, pp. 38–53.

Ruggieri, C. and T. P. Murtagh (1988). Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, New York, NY, USA, pp. 285–293. ACM.

Shankar, A. and R. Bodik (2007). DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Programming Language Design and Implementation*.

Sleator, D. D. and R. E. Tarjan (1985). Self-adjusting binary search trees. *Journal of the ACM 32*(3), 652–686.

Sümer, O., U. A. Acar, A. Ihler, and R. Mettu (2011). Fast parallel and adaptive updates for dual-decomposition solvers. In *Conference on Artificial Intelligence (AAAI)*.

Sundaresh, R. S. and P. Hudak (1991). Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pp. 1–13.

Tarditi, D. and A. Diwan (1994). Measuring the cost of storage management. In *Lisp and Symbolic Computation*, pp. 323–342.

Tarditi, D., P. Lee, and A. Acharya (1992). No assembly required: compiling Standard ML to C. *ACM Letters on Programming Languages and Systems 1*(2), 161–177.

Tarjan, R. E. (1987, March). Algorithm design. *Communications of the ACM 30*(3).

Tofte, M., L. Birkedal, M. Elsman, and N. Hallenberg (2004, September). A retrospective on region-based memory management. *Higher-Order and Symbolic Computation 17*(3).

Tofte, M. and J.-P. Talpin (1997, February). Region-based memory management. *Information and Computation*.

Vesperman, J. (2003). *Essential CVS - version control and source code management*. O'Reilly.

Wadler, P. (1995). Monads for functional programming. In J. Jeuring and E. Meijer (Eds.), *Advanced Functional Programming*, Volume 925 of *Lecture Notes in Computer Science*, pp. 24–52. Springer Berlin / Heidelberg.

Weeks, S. (2006). Whole-program compilation in mlton. In *ML '06: Proceedings of the 2006 workshop on ML*, pp. 1–1. ACM.

Wilson, P. R. (1992, 16–18 September). Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen (Eds.), *Proceedings of International Workshop on Memory Management*, Volume 637 of *Lecture Notes in Computer Science*, University of Texas, USA. Springer-Verlag.

Yellin, D. M. and R. E. Strom (1991, April). INC: a language for incremental computations. *ACM Transactions on Programming Languages and Systems 13*(2), 211–236.

Zorn, B. (1993). The measured cost of conservative garbage collection. *Software Practice and Experience 23*, 733–756.

Appendix A
Surface language code listings

Figure A.1: The `List` module.

```
#module_begin List
#imports
  type hd_t                       -- parametric head type
  qual ptr_q                      -- qualifier for internal pointers
#body
  type cons_t                     -- abstract cons cell
  type t      = cons_t* ptr_q ;   -- a list is a qualified cons_t pointer
  type cons_t = { hd : hd_t ;     -- the cons head
                  tl : t    ; }  -- the cons tail
  val nil    : t                  -- defined as NULL
  val cons   : void        -> t  -- allocates new cons; no init.
  val consh  : hd_t        -> t  -- like cons, but with head init
  val consht : hd_t, t     -> t  -- like consh, but with tail init.
  val mcp    : cons_t*     -> t  -- memoized copy of cons cell
  val tlx    : t*, cons_t* -> t* -- tail-extend a list, rets extended tl ptr
#module_end
```

Figure A.2: The `List_map` module.

```
#module_begin List_map
#imports
  module A = [ List ]
  module B = [ List ]
  val fn : A.hd_t -> B.hd_t
#body
  val map  : A.t -> B.t
  val mapt : A.t, B.t* -> void
-------------------------------------------------------------------------------
  B.t map ( in ) {
    if( in ) {
      memo;
      hd_t hd = fn( in->hd );     -- 1. map hd
      B.t  tl = map( in->tl );    -- 2. map tl
      return B.consht( hd , tl ); -- 3. allocate cons; write hd and tl
    }
    else
      return B.nil;
  }

  B.t mapt ( in, d ) { -- tail-recursive version of map, uses destination arg d
    if( in ) {
      B.t c; -- key diff: we allocate the cons cell before recurring, not after
      memo { hd_t hd = fn( in->hd );   -- 1. map hd
             c       = B.cons( hd ); } -- 2. allocate cons, write hd
      *d = c;                          -- write prev tl: cons case.
      memo;
      mapt( in->tl, & c->tl );   -- 3. map tl
    }
    else *d = B.nil;             -- write prev tl: nil case.
  }

-------------------------------------------------------------------------------
#module_end
```

Figure A.3: The `List_util` module.

```
#module_begin List_util
#imports
  module L = [ LIST ]
#body
  val len     : L.t         -> size_t       -- compute list length
  val len_lte : L.t, size_t -> bool_t       -- test list length
  val rev     : L.t         -> L.t          -- list reversal
  val rev_r   : L.t, L.t    -> L.t          -- recursive accumulator-based def.
 ------------------------------------------------------------------------------
 size_t len ( in ) {
   size_t a = 0;                             -- initialize accumulator
   while( in ) { a++; in = in->tl; }         -- walk the list
   return a; }                               -- return accumulated value

 bool_t len_lte ( in, n ) {
   if      ( in == L.nil ) return TRUE;      -- basecase 1: empty list
   else if ( n == 0 )      return FALSE;     -- basecase 2: not( len <= 0 )
   else memo; len_lte ( in->tl, n-1 ); }     -- recur; both args decrease

 L.t rev ( in ) {
   return rev_r( in, L.nil ); }              -- use rev_r with empty acc

 L.t rev_r ( in, r ) {                       -- r is the reversed accumulator
   if( in == L.nil ) return r;               -- basecase: return rev acc
   else { L.t c = L.mcp( in );               -- memo copy of first cons
          c->tl = r;                         -- c is new prefix of rev acc
          memo;                              -- memo point guards recursion
          rev_r( in->tl, c ); }              -- reverse the tail
 ------------------------------------------------------------------------------
#module_end
```

Figure A.4: The modules `List_pred1` and `List_pred2` .

```
#module_begin List_pred1 -- unary predicate
#imports
  module L = [ LIST ]
  val pred : L.hd_t* -> bool_t
#body
  val filter : L.t,  L.t*       -> void     -- one in, one out
  val split  : L.t,  L.t*, L.t* -> void     -- one in, two out
  ------------------------------------------------------------------------
  void filter ( in, d ) {
    if ( in == L.nil ) *d = L.nil;           -- basecase: empty.
    else { if( pred( & in->hd ) )            -- conditioned on predicate,
             d = L.tlx( d, L.mcp( in ) );    -- ..tail-extend with memo copy
           memo; filter( in->tl, d ); }      -- memo-guarded recursion on tail

  void split ( in, d1, d2 ) {
    if ( in == L.nil ) *d = L.nil;           -- basecase: empty.
    else { L.t c = L.mcp( in );              -- memo copy cons,
      memo;                                  -- ..memo point, guards recursion
      if( pred( & in->hd ) )                 -- which output to extend?
           split( in->tl, L.tlx( d1, L.mcp( in ) ), d2 );    -- extend first
      else split( in->tl, d1, L.tlx( d2, L.mcp( in ) ) ); }} -- extend second
  ------------------------------------------------------------------------
#module_end

#module_begin List_pred2 -- binary predicate
#imports
  module L = [ LIST ]
  val pred  : L.hd_t*, L.hd_t* -> bool_t    -- compare heads
#body
  val merge : L.t, L.t,  L.t*  -> void      -- two in, one out
  ------------------------------------------------------------------------
  void merge ( in1, in2, d ) {
    if      ( in1 == L.nil ) *d = in2;       -- basecase one: in1 empty
    else if ( in2 == L.nil ) *d = in1;       -- basecase two: in2 empty
    else { memo;                             -- memo point, guards recursion
      if( pred( & in1->hd, & in2->hd ) )     -- which input to take?
              merge( in1->tl, in2, L.tlx( d, L.mcp( in1 ) ) ); -- take first
           else merge( in1, in2->tl, L.tlx( d, L.mcp( in2 ) ) ); -- take second
  ------------------------------------------------------------------------
#module_end
```

Figure A.5: The `coin` module.

```
#module_begin COIN -- deterministic random choices
#exports
  type key_t = uintptr_t                    -- keys index determ. coin tosses
  type coin_t                               -- coins have random seeds
  val init     :        coin_t* -> void     -- initialize a new random coin
  val det_toss : key_t, coin_t* -> bool_t   -- determ. toss, on coin + key
#module_end
```

Figure A.6: The `List_reduce` module.

```
#module_begin List_reduce
#imports
  module L = [ LIST ]                    -- list elements have,
  val binop : L.hd_t, L.hd_t -> L.hd_t   -- ..associative binop
  module C = [ COIN ]                    -- coin tosses choose re-associations
#body
  module Lu = [ List_util ] with L

  val reduce : L.t* -> L.hd_t
  ------------------------------------------------------------------------
  L.hd_t reduce ( in ) {                     -- assume: input is non-empty
    if ( cut( Lu.len_lte ( *in, 1 ) ) ) {    -- is it one element?
      assert( in != L.nil );                 -- ..breaks our assumption
      return in->hd ;                        -- return single element.
    } else {                                 -- two or more elements:
      L.t             out ;                    -- out: result of reduce round
      L.t*      d = & out ;                    -- cursor for tlx'ing out
      C.coin_t       coin ;                    -- coin, to determine re-assoc
      C.init( & coin );                        -- ..seed coin randomly
      cut {                                    -- begin reduction round:
       while ( in ) {                              -- loop over input list
         L.hd_t acc = in->hd ;                     -- seed accumlator
         in         = in->tl ;                     -- next input element
         while ( in && C.det_toss( in, coin ) ) {   -- coin flips select breaks:
           acc = binop( acc, in->hd ); in = in->tl; }   -- dont break: accumulate
         L.t acc_cons = memo( in; L.cons( acc ) );  -- break: keys = { in, acc }
         d = L.tlx( d, acc_cons );                  -- extend out with acc_cons
         memo;                                      -- keys = { in, d }
      }}
      return reduce( & out );                -- this round is finished, recur..
    }}
  ------------------------------------------------------------------------
#module_end
```

Figure A.7: The `List_msort` module.

```
#module_begin List_msort
#imports
  module L = [ LIST ]
  module C = [ COIN ]
  val lte : L.hd_t*, L.hd_t* -> bool_t
#body
  module Lu = [ List_util ] with L    -- use List_util.len_lte
  module P1 = [ List_pred1 ] with L   -- divide step: List_pred1.split
    and ( add-arg pred C.coin_t* )     -- add coin arg to pred & its uses
          := C.det_toss                -- use C.det_toss as splitting pred
  module P2 = [ List_pred2 ]          -- combine step: List_pred2.merge
    with pred := lte                   -- use lte pred as merging pred

  val msort : L.t*, L.t* -> void      -- merge sort
  ------------------------------------------------------------------
  void msort ( in, d ) {
    if( cut( L.len_lte( *in, 1 ) ) ) { -- basecase test: len <= 1
      L.tlx( d, in );                  -- basecase: list is sorted
    } else {                           -- recursive, divide-and-conquer:
      L.t m1, m2;                      -- reserve space to hold ptrs
      P1.split ( in, & m1, & m2 );     -- divide: input into m1 and m2
      L.t s1; msort( & m1, & s1 );        -- reserve s1; sort m1 into s1
      L.t s2; msort( & m2, & s2 );        -- reserve s2; sort m2 into s2
      P2.merge ( s1, s2, d );          -- combine: merge sorted lists
    }
  }
  ------------------------------------------------------------------
#module_end
```

Figure A.8: The `List_qsort` module.

```
#module_begin List_qsort
#imports
  module L = [ LIST ]
  val lte : L.hd_t*, L.hd_t* -> bool_t
#body
  module P1 = [ List_pred1 ] with L   -- divide step: List_pred1.split
    and ( add-arg pred L.hd_t* )       -- add pivot arg to pred & its uses
        := lte                         -- use lte to as pivoting pred

  val qsort   : L.t, L.t*             -- quick-sort
  val qsort_r : L.t, L.t*, L.t        -- recursive version
  ---------------------------------------------------------------------
  void qsort ( in, d ) {
    qsort_r ( in, d, L.nil ) ; }       -- accumulator is empty

  void qsort_r ( in, d, rest ) {       -- rest is a sorted accumulator
    if( in == L.nil ) L.tlx( d, rest );   -- basecase: acc is sorted.
    else {
      L.hd_t p      = in->hd ;         -- the first element, p, is the pivot
      memotbl_t* mt ;                  -- ..associate allocation with p, below
      L.t     pcons ;                  -- ..
      L.t*    l     ;                  -- ..
      L.t*    g     ;                  -- ..
      memo {                           -- keys = { p }, allocate:
        mt    = Memotbl.mk();             -- a memo table
        pcons = L.consh( p );             -- cons cell, holding p
        l     = alloc( L.t );             -- list of those <= p
        g     = alloc( L.t );             -- list of those  > p
      }
      memo ( mt ) {                              -- memo table associated with p
        P1.split( in->tl, l, g, & pcons->hd );     -- pivot tail into l and g
      }
      memo { qsort( *l, d,           pcons ); } -- recur on l, in (d, pcons]
      memo { qsort( *g, & pcons->tl, rest  ); } -- recur on g, in (pcons, rest]
    }
  }
  ---------------------------------------------------------------------
#module_end
```

APPENDIX B
RUN-TIME SYSTEM LISTINGS

Figure B.1: The `Cell` module: for building modifiable references[†].

```
#module_begin Cell
#imports
  module V   = [ MODREF_VALUE ]              -- the cell's value
  module Rds = [ NODE_SET, NODE_ITER_NEXT ] -- record of a cell's reads
#body
  type cell_t = { val : V.t ;          -- current value of the cell
                  rds : Rds.t ; }       -- readers; some may be inconsistent

  type rd_t   = { trnd : trnd_t*  ;    -- a rd_t is a trace action (an act_t)
                  rdnd : Rds.nd_t ; } -- node from the Rds node set (unboxed)

  val modify  : cell_t*, V.t    -> void -- modify the value; update readers
  val peek    : cell_t*         -> V.t  -- inspect value, but not as a reader
  val put_rd  : cell_t*, rd_t*  -> V.t  -- inspect value, as a reader
  val rem_rd  : rd_t*           -> void -- remove reader membership
  ---------------------------------------------------------------------------
  void rdnd_req_update ( rdnd ) {
    rd_t* rd = rd_of_rdnd( rdnd ); -- (pointer arith)
    Trnd.req_update( rd->trnd );   -- requests that the trnd be udpated
  }
  ---------------------------------------------------------------------------
  module Rds_req_update = [ Node.Visit ] with Ns := Rds
                            and on_visit := rdnd_req_update
  ---------------------------------------------------------------------------
  void modify ( cell, val ) {
    if( V.eq( cell->val, val ) == FALSE ){ -- has value changed?
      modref->val = val;                    -- save new value
      Rds_req_update.all( & cell->rds );   -- update all reads
    }
  }

  V.t peek( cell ){ return cell->val; }

  V.t put_rd( cell, rd ){
    Rds.nd_t* rdnd = & rd->rdnd;       -- Rds node of the read action
    if( Rds.is_inserted( rdnd ) )      -- is it associated with some cell?
      Rds.remove( rdnd );              -- if so, remove it
    Rds.insert( & cell->rds, rdnd );  -- insert it into cell
    return peek( cell );               -- do the read
  }

  void rem_rd( rd ){ Rds.remove( & rd->rdnd ); }
  ---------------------------------------------------------------------------
#module_end
```

[†] Section B and Section B give single-write ("one-shot") and multiple-write modifiable references, respectively (See the `MODREF` signature, Figure B.2). Each implementation uses a cell to represent a write trace action: a modifiable value, with a set of readers.

Figure B.2: The MODREF signature: An interface to modifiable references.

```
#module_begin MODREF_VALUE
  type t                   -- stored value type
  val eq : t, t -> bool_t -- value type equality relation
#module_end

#module_begin MODREF
#imports
  module V = [ MODREF_VALUE ]
#exports
  type mrf_t -- modref type; replaces qualified occurrences of V.t
  thc read  : mrf_t*      -> V.t   -- reading
  thc write : mrf_t*, V.t -> void  -- writing
#module_end
```

Figure B.3: The `Oneshot_modref` module: Single-write modifiable references

```
#module_begin Oneshot_modref
#exports
  module_open [ MODREF ] -- we implement this signature
#imports
  module C = [ Cell ]    -- modifiable cells that record readers
#body
  type mrf_t = C.cell_t    -- oneshot modrefs are cells
  type rd_t  = C.rd_t      -- oneshot reads are cell reads
  ...
```

```
#module_begin Read
#body
  type clos_t = rd_t

  val foreign : mrf_t*                -> V.t
  val invoke  : trnd_t*, rd_t*, mrf_t* -> V.t
  val consis  : trnd_t*, rd_t*         -> bool_t
  val revoke  : trnd_t*, rd_t*         -> void
  val revinv  : trnd_t*, rd_t*, mrf_t* -> vt
  ---------------------------------------------------------------------
  V.t    foreign ( m )          { return C.peek( m ); }
  V.t    invoke ( trnd, rd, m ) { rd->trnd = trnd; return C.put_rd( rd, m ); }
  bool_t consis ( _, _ )        { return FALSE; }
  void   revoke ( _, rd )       { C.rem_rd( rd ); }
  V.t    revinv ( _, rd, m )    { return C.put_rd( rd, m ); }
  ---------------------------------------------------------------------
#module_end
```

```
#module_begin Write
#body
  type clos_t = empty_t -- sizeof(empty_t) is zero

  val foreign : mrf_t*, V.t                    -> void
  val invoke  : trnd_t*, empty_t*, mrf_t*, V.t -> void
  val consis  : trnd_t*, empty_t*              -> bool_t
  val revinv  : trnd_t*, empty_t*, mrf_t*, V.t -> void
  val revoke  : trnd_t*, empty_t*              -> void
  ---------------------------------------------------------------------
  V.t    foreign ( m, v )       { C.modify( m, v ); }
  V.t    invoke  ( _, _, m, v ) { C.modify( m, v ); }
  bool_t consis  ( _, _ )       { return TRUE;      }
  void   revoke  ( _, _ )       { /*nothing to do*/ }
  V.t    revinv  ( _, _, m, v ) { C.modify( m, v ); }
  ---------------------------------------------------------------------
#module_end
```

[a] Figure B.2 contains the definition of the `MODREF` signature
[b] Figure B.1 contains the definition of the `Cell` module

Figure B.4: The `Multwr_modref` module: Declarations.

```
#module_begin Multwr_modref
#exports
  module_open [ MULTWR_MODREF ]
#imports
  module Wrs = [ NODE_SET, NODE_ORD, NODE_ITER_PREV ] -- write actions
                with key_t       := act_t*
                and  key_compare := act_compare

  module Rds = [ NODE_SET, NODE_ORD, NODE_ITER_NEXT ] -- read actions
                with key_t       := act_t*
                and  key_compare := act_compare

  module C = [ Cell ] -- write actions contain cells
                with V, Rds -- cells have a value and some read actions
#body
  type modref_t = { wrs : Wrs.t ;   -- sequence of zero or more write actions
                    fst : wr_t  ; } -- invariant: if inserted, then is first

  type wr_t = { trnd  : trnd_t*   ;   -- write actions are trace actions
                mrf   : modref_t* ;   -- the modref written
                wrnd  : Wrs.nd_t  ;   -- a node in the write action sequence
                cell  : C.cell_t  ; } -- a cell with write value and readers

  type rd_t = { cell_rd : C.rd_t ;   -- C.rd_t is trace action
                wr      : wr_t*  ;   -- the corresponding write
                val     : V.t    ; } -- for consistency check with wr->cell
```

Figure B.5: The `Multwr_modref.Read` module: Read trace hooks.

```
#module_begin Read
#body
  type clos_t = rd_t
  val foreign : modref_t*                  -> V.t
  val invoke  : trnd_t*, rd_t*, modref_t* -> V.t
  val consis  : trnd_t*, rd_t*             -> bool_t
  val revoke  : trnd_t*, rd_t*             -> void
  val revinv  : trnd_t*, rd_t*, modref_t* -> vt
  ---------------------------------------------------------------------------
  V.t foreign ( m ) { C.peek( mrf_lst_wr( m )->cell ); }

  V.t invoke ( trnd, rd, m ) {
    rd->trnd       = trnd;                                  -- store trnd
    Wrs.nd_t* wrnd = Wrs.key_search( & m->wrs, (act_t*) rd ); -- find write
    wr_t*     wr   = wr_of_wrnd( wrnd );                    -- pointer arith
    rd->val        = C.put_rd( & wr->cell, & rd->cell_rd );   -- inspect cell
    return rd->val; }

  bool_t consis ( _, rd ) {
    return V.eq( C.peek( rd->wr->cell.val ), rd->val ); } -- check against cell

  void revoke ( _, rd ) { C.rem_rd( & cell->rd ); } -- end membership with cell

  V.t revinv ( trnd, rd, m ) { -- reinsert (read modref or cell may be different)
    revoke( trnd, rd );
    return invoke( trnd, rd, m ); }
  ---------------------------------------------------------------------------
#module_end
```

Figure B.6: The `Multwr_modref.Write` module: Write trace hooks.

```
#module_begin Write
  type clos_t = wr_t
  val foreign : modref_t*, V.t                  -> void
  val invoke  : trnd_t*, wr_t*, modref_t*, V.t -> void
  val consis  : trnd_t*, wr_t*                  -> bool_t
  val revoke  : trnd_t*, wr_t*                  -> void
  val revinv  : trnd_t*, wr_t*, modref_t*, V.t -> void
--------------------------------------------------------------------------------
V.t foreign ( m, v ) {
  if( ! Wrs.is_inserted( & m->fst ) ) {
    trnd_t* root = Self_adj_machine.get_root( );
    invoke( root, & m->fst, m, v );
  } else
    C.modify( & m->fst.cell, v ); }

V.t invoke ( trnd, wr, m, v ) {
  wr->trnd = trnd ;
  Wrs.insert( & m->wrs, & wr->wrnd ); -- insert the write node
  wrnd_t* prev_wrnd = Wrs.prev( & m->wrs, & wr->wrnd );
  -- is there is a previous write?
  if( prev_wrnd != NULL ) {
    -- if so, look for reads that occur after previous write
    Rds.t*    rds  = &( wr_of_wrnd( prev_wrnd )->rds );
    Rds.nd_t* rdnd = Rds.key_search( rds, wr );
    if( rdnd && act_compare( wr, rd_of_rdnd( rdnd ) ) == LESS ) {
      -- for all reads after this write, move the reads to this write
      Rds.iter_t i;
      Rds.iter_at_nd( rds, rdnd, & i );
      Rds_put.rest( wr, & i );
    }}
  return wr->val ;
}

bool_t consis ( _, _ ) { return TRUE; }

void revoke ( trnd, wr ) {
  wrnd_t* prev_wrnd = Wrs.prev( & wr->mrf->wrs, & wr->wrnd );
  Rds_put.all( & wr->rds, wr_of_wrnd( prev_wrnd ) ); }

void revinv ( trnd, wr, m, v ) {
  if( wr->mrf != m ) {        -- different modref ?
    revoke( trnd, wr );       -- revoke write on original modref
    invoke( trnd, wr, m, v ); -- invoke on modref m
  } else
    C.modify( wr->cell, v );}  -- modify the write action cell
--------------------------------------------------------------------------------
#module_end
```

APPENDIX C
ABSTRACT MACHINE PROOFS

The proofs presented in this chapter were the result of a collaboration with Georg Neis (Hammer et al., 2011). Georg's insights and hard work were instrumental in the development of these formal results.

C.1 Proofs for Consistency

Definition C.1.1 (SA)**.** We define $\mathsf{SA}(\sigma, \rho, e)$ to mean the following:
If $\sigma, \epsilon, \rho, e \xrightarrow{\mathsf{r}}{}^{*} \_, \_, \rho', \textbf{update}\ e'$,
then there exists $\overline{\nu}$ such that $\overline{w} = \overline{\nu}$ whenever $\_, \epsilon, \rho', e' \xrightarrow{\mathsf{r}}{}^{*} \_, \epsilon, \epsilon, \overline{w}$.

Definition C.1.2 (CSA)**.** We define $\mathsf{CSA}(\sigma, \rho, e)$ to mean the following:
If $\sigma, \varepsilon, \rho, e \xrightarrow{\mathsf{r}}{}^{*} \sigma', \kappa, \rho', e'$, then $\mathsf{SA}(\sigma', \rho', e')$.

Definition C.1.3. We write $\mathsf{noreuse}(\langle \Pi, \mathsf{T} \rangle)$ if and only if

1. $\mathsf{T} = \varepsilon$
2. $\boxminus_{\_} \notin \Pi$
3. $\boxplus_{\_} \notin \Pi$

Definition C.1.4 (From-scratch consistent traces)**.** A trace T is from-scratch consistent, written $\mathsf{T}\ \mathsf{fsc}$, if and only if there exists a closed command $\langle \rho, \alpha_{\mathsf{r}} \rangle$, store σ, and trace context Π such that

1. $\mathsf{CSA}(\sigma, \rho, \alpha_{\mathsf{r}})$
2. $\mathsf{noreuse}(\langle \Pi, \varepsilon \rangle)$
3. $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}} \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
4. $\langle \Pi', \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \Pi, \varepsilon \rangle; \mathsf{T}$

Lemma C.1.1 (From traced to untraced)**.** *If* $\langle \Pi, \mathsf{T} \rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}} \xrightarrow{t}{}^{n} \langle \Pi', \mathsf{T}' \rangle, \sigma', \kappa', \rho', \alpha_{\mathsf{r}}{}'$ *using only* **E.*** *and* **U.***, *then we also have* $\sigma, \kappa, \rho, \alpha_{\mathsf{r}} \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \alpha_{\mathsf{r}}{}'$.

Proof. By inducton on n. When $n = 0$, the claim is obvious. For $n > 0$, we inspect the first step taken. In each possible case, it is easy to verify that the claim follows by induction. □

Definition C.1.5 (Okay traces)**.**

$$\frac{}{\varepsilon\ \mathsf{ok}} \qquad \frac{\mathsf{T}_1\ \mathsf{ok} \qquad \mathsf{T}_2\ \mathsf{ok}}{(\mathsf{T}_1) \cdot \mathsf{T}_2\ \mathsf{ok}} \qquad \frac{\mathsf{T}\ \mathsf{fsc}}{\mathsf{T}\ \mathsf{ok}}$$

Definition C.1.6 (Okay trace contexts)**.**

$$\frac{}{\varepsilon\ \mathsf{ok}} \qquad \frac{\Pi\ \mathsf{ok}}{\Pi{\cdot}\mathsf{t}\ \mathsf{ok}} \qquad \frac{\Pi\ \mathsf{ok}}{\Pi{\cdot}\Box\ \mathsf{ok}} \qquad \frac{\Pi\ \mathsf{ok} \qquad \mathsf{T}\ \mathsf{fsc}}{\Pi{\cdot}\boxplus_{\mathsf{T}}\ \mathsf{ok}} \qquad \frac{\Pi\ \mathsf{ok} \qquad \mathsf{T}\ \mathsf{ok}}{\Pi{\cdot}\boxminus_{\mathsf{T}}\ \mathsf{ok}}$$

Definition C.1.7 (Okay trace zippers)**.**

$$\frac{\Pi\ \mathsf{ok} \qquad \mathsf{T}\ \mathsf{ok}}{\langle \Pi, \mathsf{T}\rangle\ \mathsf{ok}}$$

Lemma C.1.2 (Rewinding is okay)**.** *If* $\langle \Pi, \mathsf{T}\rangle; \_ \circlearrowleft^* \langle \Pi', \mathsf{T}'\rangle; \_$ *and* $\langle \Pi, \mathsf{T}\rangle\ ok$ *then* $\langle \Pi', \mathsf{T}'\rangle\ ok$

Proof. Trivial induction around the following case analysis of a rewind step.

- Case $\langle \Pi_1{\cdot}\mathsf{t}, \mathsf{T}\rangle; \_ \circlearrowleft \langle \Pi_1, \mathsf{T}\rangle; \_$
  - By assumption, $\Pi_1{\cdot}\mathsf{t}$ ok and T ok
  - By inversion, Π_1 ok
  - Hence, $\langle \Pi_1, \mathsf{T}\rangle$ ok.
- Case $\langle \Pi_1{\cdot}\boxminus_{\mathsf{T}_1}, \varepsilon\rangle; \_ \circlearrowleft \langle \Pi_1, \mathsf{T}_1\rangle; \_$
  - By assumption $\Pi_1{\cdot}\boxminus_{\mathsf{T}_1}$ ok
  - By inversion, Π_1 ok and T_1 ok
  - Hence, $\langle \Pi_1, \mathsf{T}_1\rangle$ ok.
- Case $\langle \Pi_1{\cdot}\boxminus_{\mathsf{T}_2}, \mathsf{t}{\cdot}\mathsf{T}_1\rangle; \_ \circlearrowleft \langle \Pi_1, (\mathsf{t}{\cdot}\mathsf{T}_1){\cdot}\mathsf{T}_2\rangle; \_$
  - By assumption $\Pi_1{\cdot}\boxminus_{\mathsf{T}_2}$ and $\mathsf{t}{\cdot}\mathsf{T}_1$ ok
  - By inversion, Π_1 ok and T_2 ok
  - Hence, $(\mathsf{t}{\cdot}\mathsf{T}_1){\cdot}\mathsf{T}_2$ ok
  - And $\langle \Pi_1, (\mathsf{t}{\cdot}\mathsf{T}_1){\cdot}\mathsf{T}_2\rangle$ ok.

□

Lemma C.1.3 (Purity)**.** *If* $\langle \Pi_1, \mathsf{T}_1\rangle, \sigma_1, \kappa_1, \rho, \alpha_r \xrightarrow{t}{}^* \langle \Pi_1, \mathsf{T}_1\rangle, \sigma_1, \kappa_1, \rho', \alpha_r{}'$ *using* **E.0** *only, then for any* Π_2*,* T_2*,* σ_2*,* κ_2 *we have* $\langle \Pi_2, \mathsf{T}_2\rangle, \sigma_2, \kappa_2, \rho, \alpha_r \xrightarrow{t}{}^* \langle \Pi_2, \mathsf{T}_2\rangle, \sigma_2, \kappa_2, \rho', \alpha_r{}'$*.*

Proof. Trivial induction. □

Lemma C.1.4 (Rewinding)**.** *If* $\langle \Pi', \_\rangle; \_ \circlearrowleft^* \langle \Pi, \_\rangle; \_$*, then*

1. $\Pi \in \mathsf{Prefixes}(\Pi')$*,*
2. $\#_{\Box}(\Pi') = \#_{\Box}(\Pi)$*, and*

3. $\#_{\boxplus}(\Pi') = \#_{\boxplus}(\Pi)$.

Proof. By induction on the number n of rewinding steps. If $n = 0$, then $\Pi = \Pi'$ and the claim holds trivially. Suppose $n = 1 + n'$. Case analysis on the first step:

- Case $\langle \Pi', T_1' \rangle; T_2' \circlearrowleft \langle \Pi'', T_1' \rangle; t \cdot T_2' \circlearrowleft^{n'} \langle \Pi, T_1 \rangle ; T_2$ with $\Pi' = \Pi'' \cdot t$:
  - By induction we get $\Pi \in \mathsf{Prefixes}(\Pi'') \wedge \#_{\square}(\Pi'') = \#_{\square}(\Pi) \wedge \#_{\boxplus}(\Pi'') = \#_{\boxplus}(\Pi)$.
  - This implies the claims.
- Case $\langle \Pi', T_1' \rangle; T_2' \circlearrowleft \langle \Pi'', T \rangle; T_2' \circlearrowleft^{n'} \langle \Pi, T_1 \rangle ; T_2$ with $T_1' = \varepsilon$ and $\Pi' = \Pi'' \cdot \boxminus_T$:
  - By induction we get $\Pi \in \mathsf{Prefixes}(\Pi'') \wedge \#_{\square}(\Pi'') = \#_{\square}(\Pi) \wedge \#_{\boxplus}(\Pi'') = \#_{\boxplus}(\Pi)$.
  - This implies the claims.
- Case $\langle \Pi', T_1' \rangle; T_2' \circlearrowleft \langle \Pi'', (T_1') \cdot T \rangle; T_2' \circlearrowleft^{n'} \langle \Pi, T_1 \rangle ; T_2$ with $T_1' = t \cdot T_1''$ and $\Pi' = \Pi'' \cdot \boxminus_T$:
  - By induction we get $\Pi \in \mathsf{Prefixes}(\Pi'') \wedge \#_{\square}(\Pi'') = \#_{\square}(\Pi) \wedge \#_{\boxplus}(\Pi'') = \#_{\boxplus}(\Pi)$.
  - This implies the claims.

□

Lemma C.1.5. *If* $\Pi \in \mathsf{Prefixes}(\Pi')$*, then* $\mathsf{drop}_{\boxminus}(\Pi) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$.

Lemma C.1.6. *If* $\langle \Pi, T_1 \rangle; T \circlearrowleft^* \langle \Pi', \_ \rangle; T'$*, then* $\langle \mathsf{drop}_{\boxminus}(\Pi), T_1 \rangle; T \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi'), \_ \rangle; T'$.

Proof. By induction on the number n of rewinding steps. If $n = 0$, then $\Pi = \Pi'$ and $T = T'$, so the claim holds obviously. Now suppose $n > 0$. We inspect the last step:

- Case $\langle \Pi, T_1 \rangle; T \circlearrowleft^* \langle \Pi' \cdot t, \_ \rangle; T_2 \circlearrowleft \langle \Pi', \_ \rangle ; t \cdot T_2$ with $T' = t \cdot T_2$:
  - By induction, $\langle \mathsf{drop}_{\boxminus}(\Pi), T_1 \rangle; T \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi' \cdot t), \_ \rangle; T_2 \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi'), \_ \rangle ; t \cdot T_2$.
- Case $\langle \Pi, T_1 \rangle; T \circlearrowleft^* \langle \Pi' \cdot \boxminus_{\_}, \_ \rangle; T' \circlearrowleft \langle \Pi', \_ \rangle ; T'$:
  - By induction, $\langle \mathsf{drop}_{\boxminus}(\Pi), T_1 \rangle; T \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi' \cdot \boxminus_{\_}), \_ \rangle; T' = \langle \mathsf{drop}_{\boxminus}(\Pi'), \_ \rangle ; T'$.

□

Lemma C.1.7 (Trace actions stick around (prefix version))**.** *If*

1. $\langle \Pi_1 \cdot T_2 \cdot \Pi_2, T_1 \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^* \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$
2. $\mathsf{drop}_{\boxminus}(\Pi_1) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_3))$

then $\mathsf{drop}_{\boxminus}(\Pi_1 \cdot T_2) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_3))$.

Proof. By induction on the length n of the reduction chain. If $n = 0$, then $\Pi_3 = \Pi_1 \cdot T_2 \cdot \Pi_2$ and thus the claim is obvious. Now consider $n = 1 + n'$. We inspect the first step:

- Case **E.0**, **U.1-2**:
  - Then $\langle \Pi_1 \cdot T_2 \cdot \Pi_2, T_1 \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
  - The claim then follows by induction.
- Case **E.1–5,7**, **E.P**, **P.E,1–5,7**:
  - Then $\langle \Pi_1 \cdot T_2 \cdot \Pi_2 \cdot t, T_1 \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$, for some t.
  - The claim then follows by induction.
- Case **E.6**:
  - Then $\langle \Pi_1 \cdot T_2 \cdot \Pi_2 \cdot \Box, T_1 \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
  - The claim then follows by induction.
- Case **P.6**:
  - Then $\langle \Pi_1 \cdot T_2 \cdot \Pi_2 \cdot \boxplus_T, T_1 \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
  - The claim then follows by induction.
- Case **U.3**:
  - Then $\langle \Pi_1 \cdot T_2 \cdot \Pi_2 \cdot \boxminus_T, T_1 \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
  - The claim then follows by induction.
- Case **E.8**:
  - Subcase $\langle \Pi_1 \cdot T_2 \cdot \Pi_2, T_1 \rangle ; \varepsilon \circlearrowleft^* \langle \Pi_1 \cdot T_2 \cdot \Pi_2' \cdot \Box, T_1' \rangle ; T_3$:
    - Then $\langle \Pi_1 \cdot T_2 \cdot \Pi_2' \cdot (T_3), T_1' \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
    - The claim then follows by induction.
  - Subcase $\langle \Pi_1 \cdot T_2 \cdot \Pi_2, T_1 \rangle ; \varepsilon \circlearrowleft^* \langle \Pi_1, \_ \rangle ; \_ \circlearrowleft^* \langle \Pi_1' \cdot \Box, T_1' \rangle ; T_3$:
    - Then $\langle \Pi_1' \cdot (T_3), T_1' \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
    - By Lemma C.1.4 we have $\Pi_1' \cdot \Box \in \mathsf{Prefixes}(\Pi_1)$.
    - Hence, using Lemma C.1.5, $\mathsf{drop}_{\boxminus}(\Pi_1'), \mathsf{drop}_{\boxminus}(\Pi_1' \cdot \Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_1)) \subseteq \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_3))$.
    - Hence $\mathsf{drop}_{\boxminus}(\Pi_1' \cdot (T_3)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_3))$ by induction, contradicting $\mathsf{drop}_{\boxminus}(\Pi_1' \cdot \Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_3))$.
- Case **P.8**:
  - Subcase $\langle \Pi_1 \cdot T_2 \cdot \Pi_2, T_1 \rangle ; \varepsilon \circlearrowleft^* \langle \Pi_1 \cdot T_2 \cdot \Pi_2' \cdot \boxplus_T, \varepsilon \rangle ; T_3$ where $T_1 = \varepsilon$:

* Then $\langle \Pi_1 \cdot \mathsf{T}_2 \cdot \Pi_2' \cdot (\mathsf{T}_3), \mathsf{T} \rangle, \_, \_, \_, \_ \xrightarrow{\mathrm{t}}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
* The claim then follows by induction.

– Subcase $\langle \Pi_1 \cdot \mathsf{T}_2 \cdot \Pi_2, \mathsf{T}_1 \rangle ; \varepsilon \circlearrowleft^* \langle \Pi_1, \_ \rangle ; \_ \circlearrowleft^* \langle \Pi_1' \cdot \boxplus_{\mathsf{T}}, \varepsilon \rangle ; \mathsf{T}_3$ where $\mathsf{T}_1 = \varepsilon$:

* Then $\langle \Pi_1' \cdot (\mathsf{T}_3), \mathsf{T} \rangle, \_, \_, \_, \_ \xrightarrow{\mathrm{t}}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
* By Lemma C.1.4 we have $\Pi_1' \cdot \boxplus_{\mathsf{T}} \in \mathsf{Prefixes}(\Pi_1)$.
* Hence, using Lemma C.1.5, $\mathsf{drop}_{\boxminus}(\Pi_1'), \mathsf{drop}_{\boxminus}(\Pi_1' \cdot \boxplus_{\mathsf{T}}) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_1)) \subseteq \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_3))$.
* Hence $\mathsf{drop}_{\boxminus}(\Pi_1' \cdot (\mathsf{T}_3)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_3))$ by induction, contradicting $\mathsf{drop}_{\boxminus}(\Pi_1' \cdot \boxplus_{\mathsf{T}}) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_3))$.

- Case **U.4**:

– Subcase $\langle \Pi_1 \cdot \mathsf{T}_2 \cdot \Pi_2, \mathsf{T}_1 \rangle = \langle \Pi_1 \cdot \mathsf{T}_2 \cdot \Pi_2' \cdot \boxminus_{\mathsf{T}}, \varepsilon \rangle$:

* Then $\langle \Pi_1 \cdot \mathsf{T}_2 \cdot \Pi_2', \mathsf{T} \rangle, \_, \_, \_, \_ \xrightarrow{\mathrm{t}}{}^{n'} \langle \Pi_3, \_ \rangle, \_, \_, \_, \_$.
* The claim then follows by induction.

– Subcase $\langle \Pi_1 \cdot \mathsf{T}_2 \cdot \Pi_2, \mathsf{T}_1 \rangle = \langle \Pi_1' \cdot \boxminus_{\mathsf{T}}, \varepsilon \rangle$ where $\mathsf{T}_2 \cdot \Pi_2 = \varepsilon$:

* Then the claim is (2).

□

Lemma C.1.8 (Trace actions stick around (rewinding version))**.** *If*

- $\langle \Pi \cdot \mathrm{t}, \mathsf{T} \rangle, \_, \_, \_, \_ \xrightarrow{t}{}^{*} \langle \Pi', \mathsf{T}' \rangle, \_, \_, \_, \_$
- $\langle \mathsf{drop}_{\boxminus}(\Pi'), \mathsf{T}_1 \rangle ; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi), \mathsf{T}_2 \rangle ; \mathsf{T}_3$

then $\langle \mathsf{drop}_{\boxminus}(\Pi'), \mathsf{T}_1 \rangle ; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi \cdot \mathrm{t}), \mathsf{T}_2 \rangle ; \mathsf{T}_3' \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \mathsf{T}_2 \rangle ; \mathsf{T}_3$.

Proof. Note that the rewinding takes at least one step, otherwise Lemmas C.1.4 and C.1.7 would yield $\Pi \cdot \mathrm{t} \in \mathsf{Prefixes}(\Pi)$, a contradiction. We inspect the last step:

- Case $\langle \mathsf{drop}_{\boxminus}(\Pi'), \mathsf{T}_1 \rangle ; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi) \cdot \mathrm{t}', \mathsf{T}_2 \rangle ; \mathsf{T}_3' \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \mathsf{T}_2 \rangle ; \mathsf{T}_3$ with $\mathsf{T}_3 = \mathrm{t}' \cdot \mathsf{T}_3'$:

– Lemmas C.1.4 and C.1.7 yield $\mathsf{drop}_{\boxminus}(\Pi \cdot \mathrm{t}) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$.
– Lemma C.1.4 also yields $\mathsf{drop}_{\boxminus}(\Pi \cdot \mathrm{t}') \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$.
– Hence $\mathrm{t} = \mathrm{t}'$ and we are done.

- Case $\langle \mathsf{drop}_{\boxminus}(\Pi'), \mathsf{T}_1 \rangle ; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi) \cdot \boxminus_{\widehat{\mathsf{T}}}, \mathsf{T}_2' \rangle ; \mathsf{T}_3 \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \mathsf{T}_2 \rangle ; \mathsf{T}_3$:

– Lemma C.1.4 yields $\mathsf{drop}_{\boxminus}(\Pi) \boxminus_{\widehat{\mathsf{T}}} \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$, which is a contradiction.

□

Lemma C.1.9. *If* $\langle \Pi, T\rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t}{}^{*} \langle \Pi', T'\rangle, \sigma', \kappa', \rho', \alpha_r{}'$ *and* $\mathsf{noreuse}(\langle \Pi, T\rangle)$*, then* $\mathsf{noreuse}(\langle \Pi', T'\rangle)$.

Proof. Easy induction on the length of the reduction. □

Lemma C.1.10. *If* $\sigma, \kappa, \rho, \alpha_r \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \alpha_r{}'$*, then* $\sigma, \kappa_0@\kappa, \rho, \alpha_r \xrightarrow{r}{}^{*} \sigma', \kappa_0@\kappa', \rho', \alpha_r{}'$ *for any* κ_0.

Proof. Easy induction on the length of the reduction. □

Lemma C.1.11 (CSA preservation (untraced))**.** *If* $\sigma, \varepsilon, \rho, e \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', e'$ *and* $\mathsf{CSA}(\sigma, \rho, e)$*, then* $\mathsf{CSA}(\sigma', \rho', e')$.

Proof.

- Suppose $\sigma', \varepsilon, \rho', e' \xrightarrow{r}{}^{*} \sigma'', \kappa'', \rho'', e''$.
- We must show $\mathsf{SA}(\sigma'', \rho'', e'')$.
- By Lemma C.1.10 we get $\sigma', \kappa', \rho', e' \xrightarrow{r}{}^{*} \sigma'', \kappa'\cdot\kappa'', \rho'', e''$.
- Hence $\sigma, \varepsilon, \rho, e \xrightarrow{r}{}^{*} \sigma'', \kappa'\cdot\kappa'', \rho'', e''$.
- The goal then follows from $\mathsf{CSA}(\sigma, \rho, e)$.

□

Lemma C.1.12. *Suppose the following:*

- $\mathsf{drop}_{\boxminus}(\Pi) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$*,*
- $\mathsf{drop}_{\boxminus}(\Pi\cdot\Box) \notin \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$*,*
- $|\widetilde{\kappa}| = \#_{\Box}(\widetilde{\Pi})$.

1. *If* $\langle \Pi\cdot\Box\cdot\widetilde{\Pi}, T_0\rangle, \sigma, \kappa\cdot\lfloor\rho_f, f\rfloor\cdot\widetilde{\kappa}, \rho, \alpha_r \xrightarrow{t}{}^{n} \langle \Pi', T_0'\rangle, \sigma', \kappa\cdot\widehat{\kappa}, \rho', \alpha_r{}'$*, then:*
   - $\langle \Pi\cdot\Box\cdot\widetilde{\Pi}, T_0\rangle, \sigma, \kappa\cdot\lfloor\rho_f, f\rfloor\cdot\widetilde{\kappa}, \rho, \alpha_r \xrightarrow{t}{}^{n_1} \langle \Pi\cdot\Box\cdot\widetilde{\Pi}', T_0''\rangle, \sigma'', \kappa\cdot\lfloor\rho_f, f\rfloor, \varepsilon, \overline{\omega}$
   - $\langle \Pi\cdot\Box\cdot\widetilde{\Pi}', T_0''\rangle; \varepsilon \circlearrowleft^{*} \langle \Pi\cdot\Box, T_0'''\rangle; \widetilde{T}$
   - $\rho_f(f) = \mathbf{fun}\ f\mathbf{(}\overline{x}\mathbf{)}.e_f$
   - $\langle \Pi\cdot\Box\cdot\widetilde{\Pi'}, T_0''\rangle, \sigma'', \kappa\cdot\lfloor\rho_f, f\rfloor, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi\cdot(\widetilde{T}), T_0'''\rangle, \sigma'', \kappa, \rho_f[\overline{x \mapsto \omega}], e_f$
   - $\langle \Pi\cdot(\widetilde{T}), T_0'''\rangle, \sigma'', \kappa, \rho_f[\overline{x \mapsto \omega}], e_f \xrightarrow{t}{}^{n_2} \langle \Pi', T_0'\rangle, \sigma', \kappa\cdot\widehat{\kappa}, \rho', \alpha_r{}'$
   - $n = n_1 + 1 + n_2$
2. *If* $\langle \Pi\cdot\Box\cdot\widetilde{\Pi}, T_0\rangle, \sigma, \kappa\cdot\lfloor\rho_f, f\rfloor\cdot\widetilde{\kappa}, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n} \langle \Pi', T_0'\rangle, \sigma', \kappa\cdot\widehat{\kappa}, \rho', \alpha_r{}'$*, then:*

- $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}, T_0 \rangle, \sigma, \kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_1} \langle \Pi \cdot \Box \cdot \widetilde{\Pi}', T_0'' \rangle, \sigma'', \kappa \cdot \lfloor \rho_f, f \rfloor, \varepsilon, \overline{\omega}$
- $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}', T_0'' \rangle; \varepsilon \circlearrowleft^* \langle \Pi \cdot \Box, T_0''' \rangle; \widetilde{T}$
- $\rho_f(f) = \mathbf{fun}\ f(\overline{x}).e_f$
- $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}', T_0'' \rangle, \sigma'', \kappa \cdot \lfloor \rho_f, f \rfloor, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi \cdot (\widetilde{T}), T_0''' \rangle, \sigma'', \kappa, \rho_f[\overline{x \mapsto \omega}], e_f$
- $\langle \Pi \cdot (\widetilde{T}), T_0''' \rangle, \sigma'', \kappa, \rho_f[\overline{x \mapsto \omega}], e_f \xrightarrow{t}{}^{n_2} \langle \Pi', T_0' \rangle, \sigma', \kappa \cdot \widehat{\kappa}, \rho', \alpha_r{}'$
- $n = n_1 + 1 + n_2$

Proof. By mutual induction on n. If $n = 0$, then we obtain a contradiction to $\mathsf{drop}_{\boxminus}(\Pi \cdot \Box) \notin \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$. So consider $n > 0$. In each part we inspect the first step of the reduction.

1.
   - Case **E.0–7**: Straightforward, using the inductive hypothesis.
   - Case **E.8**:
     - Subcase $\#_{\Box}(\widetilde{\Pi}) = 0$:
       - Then $\kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}, \rho, \alpha_r = \kappa \cdot \lfloor \rho_f, f \rfloor, \varepsilon, \overline{\omega}$ and $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}, T_0 \rangle; \varepsilon \circlearrowleft^* \langle \Pi \cdot \Box, T_2 \rangle; T_1$.
       - Hence $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}, T_0 \rangle, \sigma, \kappa \cdot \lfloor \rho_f, f \rfloor, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi \cdot (T_1), T_2 \rangle, \sigma, \kappa, \rho_f[\overline{x \mapsto \omega}], e_f$ with $\langle \Pi \cdot (T_1), T_2 \rangle, \sigma, \kappa, \rho_f[\overline{x \mapsto \omega}], e_f \xrightarrow{t}{}^{n-1} \langle \Pi', T_0' \rangle, \sigma', \kappa \cdot \widehat{\kappa}, \rho', \alpha_r{}'$.
       - Thus the claim holds for $n_1 = 0$, $n_2 = n - 1$.
     - Subcase $\#_{\Box}(\widetilde{\Pi}) > 0$:
       - Then $\kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}, \rho, \alpha_r = \kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}' \cdot \lfloor \widehat{\rho}, \widehat{f} \rfloor, \varepsilon, \overline{\omega}$ and $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}, T_0 \rangle; \varepsilon \circlearrowleft^* \langle \Pi \cdot \Box \cdot \widetilde{\Pi}' \cdot \Box, T_0'' \rangle; \widehat{T}$.
       - So $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}, T_0 \rangle, \sigma, \kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi \cdot \Box \cdot \widetilde{\Pi}' \cdot (\widehat{T}), T_0'' \rangle, \sigma, \kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}', \widehat{\rho}, \widehat{e}$.
       - And $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}' \cdot (\widehat{T}), T_0'' \rangle, \sigma, \kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}', \widehat{\rho}, \widehat{e} \xrightarrow{t}{}^{n-1} \langle \Pi', T_0' \rangle, \sigma', \kappa \cdot \widehat{\kappa}, \rho', \alpha_r{}'$.
       - Hence the claim holds by induction.
   - Case **E.P**: By induction (part 2).
   - Case **P.E,1–7**: Not possible.
   - Case **P.8**:
     - Then $\rho, \alpha_r = \varepsilon, \overline{\omega}$ and $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}, \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi \cdot \Box \cdot \widetilde{\Pi}' \cdot \boxplus_{T_2}, \varepsilon \rangle; T_1$.
     - So $\langle \Pi \cdot \Box \cdot \widetilde{\Pi}, T_0 \rangle, \sigma, \kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi \cdot \Box \cdot \widetilde{\Pi} \cdot (T_1), T_2 \rangle, \sigma, \kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}, \varepsilon, \mathbf{prop}$.
     - And $\langle \Pi \cdot \Box \cdot \widetilde{\Pi} \cdot (T_1), T_2 \rangle, \sigma, \kappa \cdot \lfloor \rho_f, f \rfloor \cdot \widetilde{\kappa}, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n-1} \langle \Pi', T_0' \rangle, \sigma', \kappa \cdot \widehat{\kappa}, \rho', \alpha_r{}'$.

– Hence the claim holds by induction (part 2).

- Case **U.1–4**: Straightforward by induction.

2.
- Case **E.0–8,P**: Not possible.
- Case **P.E**: By induction (part 1).
- Case **P.1–7**: Straightforward by induction.
- Case **U.1–4**: Not possible.
- Case **P.8**: Not possible.

□

Lemma C.1.13. *Suppose the following:*

- $\mathsf{drop}_{\boxminus}(\Pi) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$
- $\mathsf{drop}_{\boxminus}(\Pi \cdot \boxplus_{T'}) \notin \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$
- $|\widetilde{\kappa}| = \#_{\square}(\widetilde{\Pi})$

1. *If* $\langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}, T_0 \rangle, \sigma, \kappa \cdot \widetilde{\kappa}, \rho, \alpha_r \xrightarrow{t}{}^{n} \langle \Pi', T_0' \rangle, \sigma', \kappa \cdot \widehat{\kappa}, \rho', \alpha_r{}', $ *then:*
   - $\langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}, T_0 \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t}{}^{n_1} \langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}', \varepsilon \rangle, \sigma'', \kappa, \varepsilon, \overline{\omega}$
   - $\langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi \cdot \boxplus_{T'}, \varepsilon \rangle; \widetilde{T}$
   - $\langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}', \varepsilon \rangle, \sigma'', \kappa, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi \cdot (\widetilde{T}), T' \rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop}$
   - $\langle \Pi \cdot (\widetilde{T}), T' \rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_2} \langle \Pi', T_0' \rangle, \sigma', \kappa \cdot \widehat{\kappa}, \rho', \alpha_r{}'$
   - $n = n_1 + 1 + n_2$
2. *If* $\langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}, T_0 \rangle, \sigma, \kappa \cdot \widetilde{\kappa}, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n} \langle \Pi', T_0' \rangle, \sigma', \kappa \cdot \widehat{\kappa}, \rho', \alpha_r{}', $ *then:*
   - $\langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}, T_0 \rangle, \sigma, \kappa \cdot \widetilde{\kappa}, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_1} \langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}', \varepsilon \rangle, \sigma'', \kappa, \varepsilon, \overline{\omega}$
   - $\langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi \cdot \boxplus_{T'}, \varepsilon \rangle; \widetilde{T}$
   - $\langle \Pi \cdot \boxplus_{T'} \cdot \widetilde{\Pi}', \varepsilon \rangle, \sigma'', \kappa, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi \cdot (\widetilde{T}), T' \rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop}$
   - $\langle \Pi \cdot (\widetilde{T}), T' \rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_2} \langle \Pi', T_0' \rangle, \sigma', \kappa \cdot \widehat{\kappa}, \rho', \alpha_r{}'$
   - $n = n_1 + 1 + n_2$

Proof. By mutual induction on n. If $n = 0$, then we obtain a contradiction to the following:

$$\mathsf{drop}_{\boxminus}(\Pi \cdot \boxplus_{T'}) \notin \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$$

So consider $n > 0$. In each part we inspect the first step of the reduction.

1.
   - Case **E.0–7**: Straightforward, using the inductive hypothesis.
   - Case **E.8**:
     - Then $\kappa{\cdot}\widetilde{\kappa}, \rho, \alpha_r = \kappa{\cdot}\widetilde{\kappa}'{\cdot}\lfloor\widehat{\rho}, \widehat{f}\rceil, \varepsilon, \overline{\omega}$ and $\langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}, T_0\rangle; \varepsilon \circlearrowleft^* \langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}'{\cdot}\square, T_0''\rangle; \widehat{T}$.
     - So $\langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}, T_0\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}'{\cdot}(\widehat{T}), T_0''\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}', \widehat{\rho}, \widehat{f}$.
     - And $\langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}'{\cdot}(\widehat{T}), T_0''\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}', \widehat{\rho}, \widehat{f} \xrightarrow{t}^{n-1} \langle \Pi', T_0'\rangle, \sigma', \kappa{\cdot}\widehat{\kappa}, \rho', {\alpha_r}'$.
     - Hence the claim holds by induction.
   - Case **E.P**: By part (2).
   - Case **P.E,1–7**: Not possible.
   - Case **P.8**:
     - Subcase $\#_{\boxplus}(\widetilde{\Pi}) = 0$:
       - Then $\langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}, \varepsilon\rangle; \varepsilon \circlearrowleft^* \langle \Pi{\cdot}\boxplus_{T'}, \varepsilon\rangle; T_1$ and $\#_{\square}(\widetilde{\Pi}) = 0$ and thus $\widetilde{\kappa} = \varepsilon$.
       - So $\langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}, T_0\rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t} \langle \Pi{\cdot}(T_1), T_2\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop}$.
       - And $\langle \Pi{\cdot}(T_1), T_2\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}^{n-1} \langle \Pi', T_0'\rangle, \sigma', \kappa{\cdot}\widehat{\kappa}, \rho', {\alpha_r}'$.
       - Thus the claim holds for $n_1 = 0$, $n_2 = n - 1$.
     - Subcase $\#_{\boxplus}(\widetilde{\Pi}) > 0$:
       - Then $\langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}, \varepsilon\rangle; \varepsilon \circlearrowleft^* \langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}'{\cdot}\boxplus_{T_2}, \varepsilon\rangle; T_1$ with $\#_{\square}(\widetilde{\Pi}') = \#_{\square}(\widetilde{\Pi})$.
       - So $\langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}, T_0\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}, \rho, \alpha_r \xrightarrow{t} \langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}'{\cdot}(T_1), T_2\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}, \varepsilon, \mathbf{prop}$.
       - And $\langle \Pi{\cdot}\boxplus_{T'}{\cdot}\widetilde{\Pi}'{\cdot}(T_1), T_2\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}, \varepsilon, \mathbf{prop} \xrightarrow{t}^{n-1} \langle \Pi', T_0'\rangle, \sigma', \kappa{\cdot}\widehat{\kappa}, \rho', {\alpha_r}'$.
       - Hence the claim holds by induction (part 2).
   - Case **U.1–4**: Straightforward, using the inductive hypothesis.
2.
   - Case **E.0–8,P**: Not possible.
   - Case **P.E**: By part (1).
   - Case **P.1–7**: Straightforward, using the inductive hypothesis.
   - Case **P.8**: Not possible.
   - Case **U.1–4**: Not possible.

□

Lemma C.1.14. *Suppose* $\mathsf{drop}_{\boxplus}(\Pi{\cdot}\square) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxplus}(\Pi'))$ *and* $\#_{\square}(\widetilde{\Pi}) = |\widetilde{\kappa}|$.

1. *If* $\langle \Pi{\cdot}\square{\cdot}\widetilde{\Pi}, T\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}, \rho, \alpha_r \xrightarrow{t}^{n} \langle \Pi', T'\rangle, \sigma', \kappa', \rho', {\alpha_r}'$, *then* $\kappa \in \mathsf{Prefixes}(\kappa')$.
2. *If* $\langle \Pi{\cdot}\square{\cdot}\widetilde{\Pi}, T\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}, \varepsilon, \mathbf{prop} \xrightarrow{t}^{n} \langle \Pi', T'\rangle, \sigma', \kappa', \rho', {\alpha_r}'$, *then* $\kappa \in \mathsf{Prefixes}(\kappa')$.

Proof. By mutual induction on n. If $n = 0$, then we obtain a contradiction to $\mathsf{drop}_{\boxminus}(\Pi{\cdot}\Box) \notin \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$. So consider $n > 0$. In each part we inspect the first step of the reduction.

1.
   - Case **E.0–7**: Straightforward, using the inductive hypothesis.
   - Case **E.8**:
     - Subcase $\#_{\Box}(\widetilde{\Pi}) = 0$:
       - Then $\langle \Pi{\cdot}(T_1), T_2\rangle, \sigma, \kappa_1, \rho_f[\overline{x \mapsto \omega}], e_f \xrightarrow{t}{}^{n-1} \langle \Pi', T'\rangle, \sigma', \kappa', \rho', {\alpha_r}'$.
       - Lemma C.1.7 yields $\mathsf{drop}_{\boxminus}(\Pi{\cdot}(T_1)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$, which contradicts the first assumption.
     - Subcase $\#_{\Box}(\widetilde{\Pi}) > 0$:
       - Then $\langle \Pi{\cdot}\Box{\cdot}\widetilde{\Pi}'{\cdot}(\widehat{T}), T\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}', \widehat{\rho}, \widehat{e} \xrightarrow{t}{}^{n-1} \langle \Pi', T'\rangle, \sigma', \kappa', \rho', {\alpha_r}'$ with $|\widetilde{\kappa}'| = |\widetilde{\kappa}| - 1 = \#_{\Box}(\widetilde{\Pi}) - 1 = \#_{\Box}(\widetilde{\Pi}') = \#_{\Box}(\widetilde{\Pi}'{\cdot}(\widehat{T}))$.
       - Hence the claim holds by induction.
   - Case **E.P**: By induction (part 2).
   - Case **P.E,1–7**: Not possible.
   - Case **P.8**:
     - Then $\langle \Pi{\cdot}\Box{\cdot}\widetilde{\Pi}'{\cdot}(\widehat{T}), T\rangle, \sigma, \kappa{\cdot}\widetilde{\kappa}, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n-1} \langle \Pi', T'\rangle, \sigma', \kappa', \rho', {\alpha_r}'$ with $|\widetilde{\kappa}| = \#_{\Box}(\widetilde{\Pi}) = \#_{\Box}(\widetilde{\Pi}') = \#_{\Box}(\widetilde{\Pi}'{\cdot}(\widehat{T}))$.
     - Hence the claim holds by induction (part 2).
   - Case **U.1–4**: Straightforward by induction.
2.
   - Case **E.0–8,P**: Not possible.
   - Case **P.E**: By induction (part 1).
   - Case **P.1–7**: Straightforward by induction.
   - Case **U.1–4**: Not possible.
   - Case **P.8**: Not possible.

□

Lemma C.1.15. *Suppose* $\mathsf{drop}_{\boxminus}(\Pi) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$.

1. *If* $\langle \Pi, T\rangle, \sigma, \kappa_1, \rho, \alpha_r \xrightarrow{t}{}^{n} \langle \Pi', T'\rangle, \sigma', \kappa_1{\cdot}\kappa, \rho', {\alpha_r}'$, *then* $\langle \Pi, T\rangle, \sigma, \kappa_2, \rho, \alpha_r \xrightarrow{t}{}^{n} \langle \Pi', T'\rangle, \sigma', \kappa_2{\cdot}\kappa, \rho', {\alpha_r}'$ *for any* κ_2.
2. *If* $\langle \Pi, T\rangle, \sigma, \kappa_1, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n} \langle \Pi', T'\rangle, \sigma', \kappa_1{\cdot}\kappa, \rho', {\alpha_r}'$, *then* $\langle \Pi, T\rangle, \sigma, \kappa_2, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n} \langle \Pi', T'\rangle, \sigma', \kappa_2{\cdot}\kappa, \rho', {\alpha_r}'$ *for any* κ_2.

Proof. Mutually, by induction on n. If $n = 0$, both parts hold trivially. Now suppose $n > 0$. We inspect the first step of each reduction.

1.
   - Case **E.0–5,7**: By Lemma C.1.7 (except E.0), induction, and application of the corresponding rule.
   - Case **E.6**:
     - Subcase $\mathsf{drop}_{\boxminus}(\Pi{\cdot}\Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$:
       - We know $\alpha_r = \textbf{push}\ f\ \textbf{do}\ e$ and $\langle \Pi{\cdot}\Box, T\rangle, \sigma, \kappa_1{\cdot}\lfloor\rho, f\rfloor, \rho, e \xrightarrow{t}{}^{n-1} \langle \Pi', T'\rangle, \sigma', \kappa_1{\cdot}\kappa, \rho', {\alpha_r}'$.
       - By Lemma C.1.14 we know $\kappa_1{\cdot}\lfloor\rho, f\rfloor \in \mathsf{Prefixes}(\kappa_1{\cdot}\kappa)$.
       - Hence $\kappa_1{\cdot}\kappa = \kappa_1'{\cdot}\kappa'$ for $\kappa_1' = \kappa_1{\cdot}\lfloor\rho, f\rfloor$ and some κ'.
       - The claim then follows by induction and application of rule **E.6**.
     - Subcase $\mathsf{drop}_{\boxminus}(\Pi{\cdot}\Box) \notin \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$:
       By Lemma C.1.12, Lemma C.1.6, Lemma C.1.4, Lemma C.1.7, induction (twice), and rule **E.6**.
   - Case **E.8**: Lemmas C.1.4, C.1.6 and C.1.7 yield both the following:
     $$\mathsf{drop}_{\boxminus}(\Pi''{\cdot}\Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$$
     $$\mathsf{drop}_{\boxminus}(\Pi''{\cdot}(\_)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$$
     This is a contradiction.
   - Case **E.P**: By Lemma C.1.7, induction (part 2), and application of **E.P**.
   - Case **P.8**: Lemmas C.1.4, C.1.6 and C.1.7 yield both
     $$\mathsf{drop}_{\boxminus}(\Pi''{\cdot}\boxplus\_) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$$
     and $\mathsf{drop}_{\boxminus}(\Pi''{\cdot}(\_)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$, which is a contradiction.
   - Case **U.1–4**: By induction and application of the corresponding rule.
   - Case **P.E,1–7**: Not possible.

2.
   - Case **E.0–8,P**: Not possible.
   - Case **P.1–5**: By Lemma C.1.7, induction, and application of the corresponding rule.
   - Case **P.6**:
     - Subcase $\mathsf{drop}_{\boxminus}(\Pi{\cdot}\boxplus_{\widehat{\mathsf{T}}}) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$: By induction and application of rule **P.6**.
     - Subcase $\mathsf{drop}_{\boxminus}(\Pi{\cdot}\boxplus_{\widehat{\mathsf{T}}}) \notin \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$:
       By Lemma C.1.13, Lemma C.1.6, Lemma C.1.4, Lemma C.1.7, induction (twice), and rule **P.6**.
   - Case **P.7,E**: By Lemma C.1.7, induction (part 1), and application of the corresponding rule.

- Case **P.8**: Not possible.
- Case **U.1–4**: Not possible.

□

Lemma C.1.16 (CSA preservation (traced))**.** *If*

1. $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, e \xrightarrow{t}{}^{*} \langle \Pi', \_ \rangle, \sigma', \kappa \cdot \kappa', \rho', e'$
2. $\mathsf{drop}_{\boxminus}(\Pi) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$
3. $\mathsf{noreuse}(\langle \Pi, \varepsilon \rangle)$
4. $\mathsf{CSA}(\sigma, \rho, e)$

then $\mathsf{CSA}(\sigma', \rho', e')$.

Proof.

- By Lemma C.1.15 we get $\langle \Pi, \varepsilon \rangle, \sigma, \varepsilon, \rho, e \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi', \_ \rangle, \sigma', \kappa', \rho', e'$.
- By Lemma C.1.9 that reduction does not use rules other than **E.*** and **U.***.
- Hence by Lemma C.1.1 we get $\sigma, \varepsilon, \rho, e \xrightarrow{\mathsf{r}}{}^{*} \sigma', \kappa', \rho', e'$.
- The claim then follows by Lemma C.1.11.

□

Lemma C.1.17 (Decomposition)**.** *Suppose* T *fsc, from initial configuration* $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}}$ *and producing* $\overline{\nu}$.

1. *If* $\mathsf{T} = A_{\ell,m} \cdot \mathsf{T}'$*, then:*
   (a) $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}} \xrightarrow{t}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho', \mathbf{let}\ x = \mathbf{alloc}(y)\ \mathbf{in}\ e$ *using* **E.0** *only*
   (b) T' *fsc from* $\langle \Pi \cdot A_{\ell,m}, \varepsilon \rangle, \sigma', \kappa, \rho'[x \mapsto \ell], e$*, producing* $\overline{\nu}$
   (c) $\sigma, \rho', \mathbf{alloc}(y) \xrightarrow{s} \sigma', \ell$
   (d) $\rho'(y) = m$
2. *If* $\mathsf{T} = R^{\nu}_{\ell[m]} \cdot \mathsf{T}'$*, then:*
   (a) $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_{\mathsf{r}} \xrightarrow{t}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho', \mathbf{let}\ x = \mathbf{read}(y[z])\ \mathbf{in}\ e$ *using* **E.0** *only*
   (b) T' *fsc from* $\langle \Pi \cdot R^{\nu}_{\ell[m]}, \varepsilon \rangle, \sigma, \kappa, \rho'[x \mapsto \nu], e$*, producing* $\overline{\nu}$
   (c) $\sigma, \rho', \mathbf{read}(y[z]) \xrightarrow{s} \sigma, \nu$
   (d) $\rho'(y) = \ell$

(e) $\rho'(z) = m$

3. *If* $T = W^{\nu}_{\ell[m]} \cdot T'$*, then:*

   (a) $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho', \textbf{let } \_ \textbf{ = write } (x[y],z) \textbf{ in } e$ *using* **E.0** *only*
   (b) T' *fsc from* $\langle \Pi \cdot W^{\nu}_{\ell[m]}, \varepsilon \rangle, \sigma', \kappa, \rho', e$*, producing* $\overline{\nu}$
   (c) $\sigma, \rho', \textbf{write } (x[y],z) \xrightarrow{s} \sigma', 0$
   (d) $\rho'(x) = \ell$
   (e) $\rho'(y) = m$
   (f) $\rho'(z) = \nu$

4. *If* $T = M_{\rho',e} \cdot T'$*, then:*

   (a) $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho', \textbf{memo } e$ *using* **E.0** *only*
   (b) T' *fsc from* $\langle \Pi \cdot U_{\rho',e}, \varepsilon \rangle, \sigma, \kappa, \rho', e$*, producing* $\overline{\nu}$

5. *If* $T = U_{\rho',e} \cdot T'$*, then:*

   (a) $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho', \textbf{update } e$ *using* **E.0** *only*
   (b) T' *fsc from* $\langle \Pi \cdot U_{\rho',e}, \varepsilon \rangle, \sigma, \kappa, \rho', e$*, producing* $\overline{\nu}$

6. *If* $T = (T_1) \cdot T_2$*, then:*

   (a) $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho', \textbf{push } f \textbf{ do } e$ *using* **E.0** *only*
   (b) T_1 *fsc from* $\langle \Pi \cdot \Box, \varepsilon \rangle, \sigma, \kappa \cdot \lfloor \rho', f \rfloor, \rho', e$*, producing* $\overline{\omega}$
   (c) T_2 *fsc from* $\langle \Pi \cdot (T_1), \varepsilon \rangle, \_, \kappa, \rho'[\overline{x \mapsto \omega}], e'$*, producing* $\overline{\nu}$
   (d) $\rho'(f) = \textbf{fun } f\textbf{(}\overline{x}\textbf{)}.e'$

7. *If* $T = \overline{\omega} \cdot T'$*, then:*

   (a) $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t}{}^{*} \langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho', \textbf{pop } \overline{x}$ *using only* **E.0**
   (b) $\rho'(\overline{x}) = \overline{\omega}$
   (c) $T' = \varepsilon$
   (d) $\overline{\omega} = \overline{\nu}$

Proof. From the assumption we know that:

(i) $\mathsf{CSA}(\sigma, \rho, \alpha_r)$

(ii) $\mathsf{noreuse}(\langle \Pi, \varepsilon \rangle)$

(iii) $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \alpha_r \xrightarrow{t}{}^{n} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$

(iv) $\langle \Pi', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi, \varepsilon \rangle; \mathsf{T}$

The proof is by induction on n. We are only interested in cases where T is nonempty and thus $n > 0$. In each part we inspect the first step of the reduction in (iii).

1. $\mathsf{T} = \mathsf{A}_{\ell,m} \cdot \mathsf{T}'$
   - Case **E.0**: By Lemma C.1.16 and induction.
   - Case **E.1**:
     - Then:
       (a) $\alpha_r = \textbf{let } x \textbf{ = alloc}(y) \textbf{ in } e$
       (b) $\langle \Pi \cdot \mathsf{A}_{\ell',m'}, \varepsilon \rangle, \sigma'', \kappa, \rho[x \mapsto \ell'], e \xrightarrow{t}{}^{n-1} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
       (c) $\sigma, \rho, \textbf{alloc}(y) \xrightarrow{s} \sigma'', \ell'$
       (d) $\rho(y) = m'$
     - By (iv), Lemma C.1.6 and Lemma C.1.8 we get
       $$\langle \mathsf{drop}_{\boxminus}(\Pi'), \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi \cdot \mathsf{A}_{\ell',m'}), \varepsilon \rangle; \mathsf{T}' \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \varepsilon \rangle ; \mathsf{T}$$
       with $\mathsf{T} = \mathsf{A}_{\ell',m'} \cdot \mathsf{T}'$, hence $\ell' = \ell$ and $m' = m$.
     - By Lemma C.1.9 we know $\mathsf{drop}_{\boxminus}(\Pi') = \Pi'$ and $\mathsf{drop}_{\boxminus}(\Pi \cdot \mathsf{A}_{\ell,m}) = \Pi \cdot \mathsf{A}_{\ell,m}$.
     - Finally, Lemma C.1.16 yields $\mathsf{CSA}(\sigma'', \rho[x \mapsto \ell], e)$ and therefore T' fsc from $\langle \Pi \cdot \mathsf{A}_{\ell',m'}, \varepsilon \rangle, \sigma'', \kappa, \rho[x \mapsto \ell'], e$, producing $\overline{\nu}$.
   - Case **E.2–8**:
     - Then $\langle \Pi \cdot t, \varepsilon \rangle, \sigma'', \kappa, \rho'', \alpha_r{}'' \xrightarrow{t}{}^* \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$ with $t \neq \mathsf{A}_{\ell,m}$ (using Lemma C.1.12 in case **E.6**).
     - By (iv), Lemma C.1.6 and Lemma C.1.8 we get $\langle \mathsf{drop}_{\boxminus}(\Pi'), \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi \cdot t), \varepsilon \rangle; \mathsf{T}' \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \varepsilon \rangle ; \mathsf{T}$ with $\mathsf{T} = t \cdot \mathsf{T}'$.
     - This is a contradiction.
   - Case **E.P,P.E,P.1–8,U.1–4**: Impossible due to (ii).
2. $\mathsf{T} = \mathsf{R}^{\nu}_{\ell[m]} \cdot \mathsf{T}'$
   - Case **E.0**: By Lemma C.1.16 and induction.
   - Case **E.2**: base case
   - Case **E.1,3–8**: contradiction
   - Case **E.P,P.E,P.1–8,U.1–4**: Impossible due to (ii).
3. $\mathsf{T} = \mathsf{W}^{\nu}_{\ell[m]} \cdot \mathsf{T}'$
   - Case **E.0**: By Lemma C.1.16 and induction.

- Case **E.3**: base case
- Case **E.1,2,4–8**: contradiction
- Case **E.P,P.E,P.1–8,U.1–4**: Impossible due to (ii).

4. $T = M_{\rho',e} \cdot T'$
   - Case **E.0**: By Lemma C.1.16 and induction.
   - Case **E.4**: base case
   - Case **E.1–3,5–8**: contradiction
   - Case **E.P,P.E,P.1–8,U.1–4**: Impossible due to (ii).
5. $T = U_{\rho',e} \cdot T'$
   - Case **E.0**: By Lemma C.1.16 and induction.
   - Case **E.5**: base case
   - Case **E.1–4,6–8**: contradiction
   - Case **E.P,P.E,P.1–8,U.1–4**: Impossible due to (ii).
6. $T = (T_1) \cdot T_2$
   - Case **E.0**: By Lemma C.1.16 and induction.
   - Case **E.6**:
     - Then $\alpha_r = \textbf{push}\ f\ \textbf{do}\ e$ and $\langle \Pi \cdot \Box, \varepsilon \rangle, \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \rho, e \xrightarrow{t}{}^{n-1} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{v}$.
     - By (iv), Lemma C.1.6, Lemma C.1.12 and Lemma C.1.9 we get:
       - $\langle \Pi \cdot \Box, \varepsilon \rangle, \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \rho, e \xrightarrow{t}{}^{n_1} \langle \Pi'', \varepsilon \rangle, \sigma'', \kappa \cdot \lfloor \rho, f \rfloor, \varepsilon, \overline{\omega}$
       - $\langle \Pi'', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi \cdot \Box, \varepsilon \rangle; \widetilde{T}$
       - $\rho(f) = \textbf{fun}\ f(x).e_f$
       - $\langle \Pi'', \varepsilon \rangle, \sigma'', \kappa \cdot \lfloor \rho, f \rfloor, \varepsilon, \overline{\omega} \xrightarrow{t} \langle \Pi \cdot (\widetilde{T}), \varepsilon \rangle, \sigma'', \kappa, \rho[\overline{x \mapsto \omega}], e_f$
       - $\langle \Pi \cdot (\widetilde{T}), \varepsilon \rangle, \sigma'', \kappa, \rho[\overline{x \mapsto \omega}], e_f \xrightarrow{t}{}^{n_2} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{v}$
       - $n - 1 = n_1 + 1 + n_2$
     - By (iv), Lemma C.1.6 and Lemma C.1.8 we get:

       $$\langle \mathsf{drop}_{\boxminus}(\Pi'), \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi \cdot (\widetilde{T})), \varepsilon \rangle; T' \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \varepsilon \rangle ; T$$

       with $T = (\widetilde{T}) \cdot T'$ and thus $T_1 = \widetilde{T}$ and $T_2 = T'$.
     - Hence by Lemma C.1.9 and Lemma C.1.16 we know:
       - T_1 fsc from $\langle \Pi \cdot \Box, \varepsilon \rangle, \sigma, \kappa \cdot \lfloor \rho, f \rfloor, \rho, e$
       - T_2 fsc from $\langle \Pi \cdot (\widetilde{T}), \varepsilon \rangle, \sigma'', \kappa, \rho[\overline{x \mapsto \omega}], e_f$

- Case **E.1–5,7,8**: contradiction
- Case **E.P,P.E,P.1–8,U.1–4**: Impossible due to (ii).

7. $T = \overline{\omega} \cdot T'$

   - Case **E.0**: By Lemma C.1.16 and induction.
   - Case **E.7**:
     - Then $\alpha_r = \textbf{pop}\ \overline{x}$ and $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, \textbf{pop}\ \overline{x} \xrightarrow{t} \langle \Pi \cdot \overline{\omega}', \varepsilon \rangle, \sigma, \kappa, \varepsilon, \overline{\omega}' \xrightarrow{t}^{n-1} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$, where $\overline{\omega}' = \rho(\overline{x})$.
     - We show that $n - 1 = 0$:
       * For a contradiction, suppose that $n - 1 > 0$.
       * Note that then the next reduction step must be either **P.8** or **E.8**.
       * In either case, using Lemmas C.1.4, C.1.5, and C.1.7, we would get a contradiction to $\langle \Pi', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi, \varepsilon \rangle; T$.
     - Hence $\overline{\omega}' = \overline{\nu}$.
     - Furthermore, Lemmas C.1.6 and C.1.8 yield $T = \overline{\omega}'$ and thus $\overline{\omega} = \overline{\nu}$ and $T' = \varepsilon$.
   - Case **E.1–6,8**: contradiction
   - Case **E.P,P.E,P.1–8,U.1–4**: Impossible due to (ii).

□

Definition C.1.8 (Last element of a trace)**.**

$$\begin{array}{lcll} \mathsf{last}(t \cdot T) & = & \mathsf{last}(T) & T \neq \varepsilon \\ \mathsf{last}(t \cdot \varepsilon) & = & t & \\ \mathsf{last}(\varepsilon) & & \textit{undefined} & \end{array}$$

Lemma C.1.18 (Evaluation values)**.** *If* T *fsc producing values* $\overline{\nu}$ *then* $\mathsf{last}(T) = \overline{\nu}$.

Proof. By induction over the structure of T.

- Case $T = \varepsilon$:
  - Not possible.
- Case $T = \overline{\omega} \cdot T'$: By Lemma C.1.17.
- Case $T = t \cdot T'$ with t not a value: By Lemma C.1.17 and induction.

□

Lemma C.1.19 (Propagation values)**.** *If*

(a) $\langle \Pi, T_1 \rangle, \sigma, \kappa, \epsilon, \textbf{prop} \xrightarrow{t}^{n} \langle \Pi', \_ \rangle, \sigma', \kappa, \epsilon, \overline{\nu}$

(b) $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle; \epsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle; \_$

(c) reduction (a) does not contain a use of **P.E**

Then $\mathsf{last}(T_1) = \overline{\nu}$

Proof. By induction on the number of reduction steps n. Note necessarily that $n > 0$. We inspect the first reduction step of (a).

- Case **E.0**-**E.8**,**U.1**-**U.4**,**P.8**,**E** not possible, due to (c).
- Case **P.1**-**P.5**
  - Then $\langle \Pi, t \cdot \widehat{T_1} \rangle, \sigma, \kappa, \epsilon, \mathbf{prop}$
    $\xrightarrow{t} \langle \Pi \cdot t, \widehat{T_1} \rangle, \sigma', \kappa, \epsilon, \mathbf{prop}$
    $\xrightarrow{t}{}^{n-1} \langle \Pi', \_ \rangle, \sigma', \kappa, \epsilon, \overline{\nu}$ with $T_1 = t \cdot \widehat{T_1}$
  - By Lemma C.1.8, we have that $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle; \epsilon \circlearrowleft^*$
    $\langle \mathsf{drop}_{\boxminus}(\Pi \cdot t), \widehat{T_1} \rangle \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle; \_$
  - The claim follows by induction.
- Case **P.6**
  - Then $\langle \Pi, (T_2) \cdot T_3 \rangle, \sigma, \kappa, \epsilon, \mathbf{prop}$
    $\xrightarrow{t} \langle \Pi \cdot \boxplus_{T_3}, T_2 \rangle, \sigma, \kappa, \epsilon, \mathbf{prop}$
    $\xrightarrow{t}{}^{n-1} \langle \Pi', \_ \rangle, \sigma', \kappa, \epsilon, \overline{\nu}$ with $T_1 = (T_2) \cdot T_3$
  - From Lemma C.1.13 we have $\langle \Pi \cdot \boxplus_{T_3}, T_2 \rangle, \sigma, \kappa, \epsilon, \mathbf{prop}$
    $\xrightarrow{t}{}^{n_1} \langle \Pi \cdot (T_2'), T_3 \rangle, \widehat{\sigma}, \kappa, \varepsilon, \mathbf{prop}$
    $\xrightarrow{t}{}^{n_2} \langle \Pi', \_ \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
  - From Lemma C.1.8, we have that $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi \cdot (T_2')), \_\rangle; \_ \circlearrowleft$
    $\langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle; \_$
  - The claim follows by induction.
- Case **P.7**:
  - Then $\langle \Pi, \overline{\omega} \cdot \varepsilon \rangle, \sigma, \kappa, \epsilon, \mathbf{prop} \xrightarrow{t} \langle \Pi \cdot \overline{\omega}, \varepsilon \rangle, \sigma, \kappa, \epsilon, \overline{\omega} \xrightarrow{t}{}^{n-1} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \epsilon, \overline{\nu}$ with $T_1 = \overline{\omega} \cdot \varepsilon$
  - We show that $n - 1 = 0$:
    - For a contradiction, suppose that $n - 1 > 0$.
    - We inspect the next step in $n - 1$, which must be either **P.8** or **E.8**. We assume **E.8**; **P.8** is analogous.

* Hence $\langle \Pi \cdot \overline{\omega}, \varepsilon \rangle, \sigma, \kappa, \varepsilon, \overline{\omega}$
  $\xrightarrow{t} \langle \Pi'' \cdot (T), T' \rangle, \sigma, \kappa', \rho_f, e_f$
  $\xrightarrow{t}{}^{n-2} \langle \Pi', \_ \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
  where $\langle \mathsf{drop}_{\boxminus}(\Pi \cdot \overline{\omega}), \varepsilon \rangle ; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi'' \cdot \Box), T' \rangle ; T$.
* By Lemma C.1.4 we know $\Pi'' \cdot \Box \in \mathsf{Prefixes}(\Pi \cdot \overline{\omega})$, i.e., $\Pi'' \cdot \Box \in \mathsf{Prefixes}(\Pi)$.
* Hence $\mathsf{drop}_{\boxminus}(\Pi''), \mathsf{drop}_{\boxminus}(\Pi'' \cdot \Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi)) \subseteq \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$ using Lemma C.1.4, (d), and Lemma C.1.5.
* Using Lemma C.1.7 we get $\mathsf{drop}_{\boxminus}(\Pi'' \cdot (T)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$, contradicting $\mathsf{drop}_{\boxminus}(\Pi'' \cdot \Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi'))$.

– Hence, since $n = 1$ we have that
  * $\Pi \cdot \overline{\nu} = \Pi'$
  * $\overline{\omega} = \overline{\nu}$
– Moreover, $\mathsf{last}(T_1) = \mathsf{last}(\overline{\nu} \cdot \varepsilon) = \overline{\nu}$

□

Lemma C.1.20 (Case analysis: waking up before push action)**.** *If*

(a) $\langle \Pi, T \rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$

(b) $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_ \rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi), \_ \rangle ; \_$

(c) Π *ok*

(d) T *fsc*

(e) $T = T_1 \cdot (T_2) \cdot T_3$

(f) T_1 *contains no parenthesis (i.e.,* $(\_) \notin T_1$*)*

Then either:

1. • $\langle \Pi, T \rangle, \sigma, \kappa, \varepsilon, \mathbf{prop}$
     $\xrightarrow{t}{}^{n_1} \langle \Pi'', U_{\rho,e} \cdot T_1' \cdot (T_2) \cdot T_3 \rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop}$
     $\xrightarrow{t} \langle \Pi'' \cdot U_{\rho,e}, T_1' \cdot (T_2) \cdot T_3 \rangle, \sigma'', \kappa, \rho, e$
     $\xrightarrow{t}{}^{n_2} \langle \Pi', \varepsilon \rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
   • $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_ \rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi'' \cdot U_{\rho,e}), \_ \rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi), \_ \rangle ; \_$
   • Π'' *ok*
   • $U_{\rho,e} \cdot T_1' \cdot (T_2) \cdot T_3$ *fsc*
   • $\mathsf{last}(T_1' \cdot (T_2) \cdot T_3) = \mathsf{last}(T)$
   • $n = n_1 + 1 + n_2$

2.
   - $\langle \Pi, T\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_1} \langle \Pi'', T_3\rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_2} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
   - $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi''), \_\rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle ; \_$
   - Π'' *ok*
   - T_3 *fsc*
   - $\mathsf{last}(T_3) = \mathsf{last}(T)$
   - $n = n_1 + n_2$, $n_1 > 0$

Proof. By induction on the number of reduction steps n. Note necessarily that $n > 0$. We inspect the first step taken.

- Cases **E.0**-**E.8**, **E.P**, **U.1**-**U.4**: not possible.
- Case **P.1**-**P.5**:
  - Then $T = t \cdot T'$ and $\langle \Pi, t\cdot T'\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t} \langle \Pi\cdot t, T'\rangle, \widehat{\sigma}, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n-1} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
  - Hence, $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi\cdot t), \_\rangle; \_ \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle; \_$ by Lemma C.1.8
  - From Π ok, we have $\Pi\cdot t$ ok
  - Note that $T' = T_1'\cdot(T_2)\cdot T_3$ where $T_1 = t\cdot T_1'$
  - Hence, from (f) we have that T_1' contains no paranthesis
  - From T fsc we have T' fsc using Lemma C.1.17
  - The claim then follows by induction.
- Case **P.E**: We show claim (1) as follows:
  - Then $T = U_{\rho,e}\cdot T_1'\cdot(T_2)T_3$ and $\langle \Pi, U_{\rho,e}\cdot T'\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t} \langle \Pi\cdot U_{\rho,e}, T'\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n-1} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
  - Hence, $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi\cdot U_{\rho,e}), \_\rangle; \_ \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle; \_$ by Lemma C.1.8
  - With $n_1 = 0$, claim (1) follows immediately by assumptions (c), (d) and (e).
- Case **P.6**: We show claim (2) as follows:
  - Then $T = (T_2)\cdot T_3$, $T_1 = \varepsilon$, and $\langle \Pi, (T_2)\cdot T_3\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t} \langle \Pi\cdot \boxplus_{T_3}, T_2\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n-1} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$

- From Lemma C.1.13 we have:
  (i) $\langle \Pi, (T_2)\cdot T_3\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop}$
  $\xrightarrow{t} \langle \Pi\cdot\boxplus_{T_3}, T_2\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop}$
  $\xrightarrow{t}{}^{m_1} \langle \Pi\cdot(T_2'), T_3\rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop}$
  $\xrightarrow{t}{}^{m_2} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{v}$
  (ii) $n = 1 + m_1 + m_2$
- Hence $\langle \mathsf{drop}_{\boxminus}(\Pi'), \varepsilon\rangle\,;\_\circlearrowleft^*\langle \mathsf{drop}_{\boxminus}(\Pi\cdot(T_2')), \_\rangle\,;\_\circlearrowleft\langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle\,;\_$ using (b) and Lemma C.1.8
- From Π ok we have $\Pi\cdot(T_2')$ ok
- From Lemma C.1.17 and (d), we have T_3 fsc
- Finally, by definition $\mathsf{last}(T_3) = \mathsf{last}((T_2)\cdot T_3) = \mathsf{last}(T)$.
- This completes the case, showing claim (2) with $n_1 = 1 + m_1$ and $n_2 = m_2$.

- Case **P.7**: not possible; it contradicts assumption (e).
- Case **P.8**: not possible; it contradicts assumption (a).

□

Lemma C.1.21 (Case analysis: final (non-nested) awakening)**.** *If*

(a) $\langle \Pi, T\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{v}$

(b) $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle;\_\circlearrowleft^*\langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle;\_$

(c) Π *ok*

(d) T *fsc*

Then either:

1. - $\langle \Pi, T\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop}$
     $\xrightarrow{t}{}^{n_1} \langle \Pi'', U_{\rho,e}\cdot T'\rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop}$
     $\xrightarrow{t} \langle \Pi''\cdot U_{\rho,e}, T'\rangle, \sigma'', \kappa, \rho, e$
     $\xrightarrow{t}{}^{n_2} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{v}$
   - $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle\,;\_\circlearrowleft^*\langle \mathsf{drop}_{\boxminus}(\Pi''\cdot U_{\rho,e}), \_\rangle\,;\_\circlearrowleft^*\langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle\,;\_$
   - Π'' *ok*
   - $U_{\rho,e}\cdot T'$ *fsc*
   - $\mathsf{last}(T) = \mathsf{last}(T')$
   - $n = n_1 + 1 + n_2$

2.
   - $\langle \Pi, T\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_1} \langle \Pi'', T'\rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_2} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
   - $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi''), \_\rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle ; \_$
   - Π'' *ok*
   - T' *fsc*
   - $\mathsf{last}(T) = \mathsf{last}(T')$
   - $n = n_1 + n_2$
   - *Reduction* n_2 *contains no use of* **P.E**

Proof. Case analysis on the shape of trace T:

- Case: $\exists T_1, T_2, T_3$ such that $T = T_1 \cdot (T_2) \cdot T_3$ and T_1 contains no parenthesis.
  - Applying lemma C.1.20, we get subcases (i) and (ii):

  (i)
  - $\langle \Pi, T\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_1} \langle \Pi'', U_{\rho,e} \cdot T_1' \cdot (T_2) \cdot T_3\rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t} \langle \Pi'' \cdot U_{\rho,e}, T_1' \cdot (T_2) \cdot T_3\rangle, \sigma'', \kappa, \rho, e \xrightarrow{t}{}^{n_2} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
  - $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi'' \cdot U_{\rho,e}), \_\rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle ; \_$
  - Π'' ok
  - $U_{\rho,e} \cdot T_1' \cdot (T_2) \cdot T_3$ fsc
  - $\mathsf{last}(T_1' \cdot (T_2) \cdot T_3) = \mathsf{last}(T)$
  - $n = n_1 + 1 + n_2$

  – This immediately shows claim (1).

  (ii)
  - $\langle \Pi, T\rangle, \sigma, \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_1} \langle \Pi'', T_3\rangle, \sigma'', \kappa, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n_2} \langle \Pi', \varepsilon\rangle, \sigma', \kappa, \varepsilon, \overline{\nu}$
  - $\langle \mathsf{drop}_{\boxminus}(\Pi'), \_\rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi''), \_\rangle ; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi), \_\rangle ; \_$
  - Π'' ok
  - T_3 fsc
  - $\mathsf{last}(T_3) = \mathsf{last}(T)$
  - $n = n_1 + n_2, n_1 > 0$

  – Since we have that $n_2 < n$, we continue by induction on reduction n_2, which shows the claim.

- Case: *Otherwise*: Note necessarily that $T = t_1 \cdot \ldots \cdot t_m$ such that $\forall i.\ t_i \neq (\_)$
  - Subcase: reduction (a) contains a use of **P.E**:
    - Hence, $\exists t_i = U_{\rho,e}$ such that $T = t_1 \cdot \ldots \cdot t_i \cdot \ldots \cdot t_m$ and $\Pi'' = \Pi \cdot t_1 \cdot \ldots \cdot t_{i-1}$
    - Then, since Π ok we have Π'' ok

* Moreover,

$$\begin{aligned}\mathsf{last}(\mathsf{T}) &= \mathsf{last}(t_1 \cdot \ldots \cdot t_i \cdot \ldots \cdot t_m) \\ &= \mathsf{last}(t_2 \cdot \ldots \cdot t_i \cdot \ldots \cdot t_m) \\ &= \mathsf{last}(t_i \cdot \ldots \cdot t_m) \\ &= \mathsf{last}(t_{i+1} \cdot \ldots \cdot t_m)\end{aligned}$$

* Since $\mathsf{T}' = t_{i+1} \cdot \ldots \cdot t_m$, we have that $\mathsf{last}(\mathsf{T}) = \mathsf{last}(\mathsf{T}')$
* We get the rest of claim (1) from repeated use of Lemmas C.1.8 and C.1.17. (the number of required uses is $i - 1$).

– Subcase: reduction (a) contains no use of **P.E**:

* Then we have claim (2) immediately, with $n_1 = 0$.

□

Theorem C.1.22 (Consistency)**.**

1. *If*

   (a) $\langle \Pi_2, \mathsf{T}_1' \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop} \xrightarrow{t}{}^{n} \langle \Pi_2', \mathsf{T}_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$

   (b) Π_2 *ok*

   (c) T_1' *fsc, from initial configuration* $\langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1}$

   (d) $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle; \mathsf{T}_2'$

   then for any Π_3 *there is* Π_3' *such that*

   (i) $\langle \Pi_2', \mathsf{T}_1'' \rangle$ *ok*

   (ii) $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_1, \alpha_{r1} \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$

   (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3, \varepsilon \rangle; \mathsf{T}_2'$

2. *If*

   (a) $\langle \Pi_2, \mathsf{T}_1' \rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2} \xrightarrow{t}{}^{n} \langle \Pi_2', \mathsf{T}_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$

   (b) $\langle \Pi_2, \mathsf{T}_1' \rangle$ *ok*

   (c) $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle; \mathsf{T}_2'$

   then for any Π_3 *there is* Π_3' *such that*

   (i) $\langle \Pi_2', \mathsf{T}_1'' \rangle$ *ok*

   (ii) $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_2, \alpha_{r2} \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$

   (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3, \varepsilon \rangle; \mathsf{T}_2'$

Proof. By simultaneous induction on n.

1. Note that necessarily $n > 0$. We inspect the first reduction step of (a).

   - Case **P.1**:
     - Then $T_1' = A_{\ell,m} \cdot \widehat{T_1}$ and $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
       $\xrightarrow{t} \langle \Pi_2 \cdot A_{\ell,m}, \widehat{T_1} \rangle, \widehat{\sigma_2}, \kappa_2, \varepsilon, \mathbf{prop}$
       $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$,
       where $\sigma_2, \varepsilon, \mathbf{alloc}(m) \xrightarrow{s} \widehat{\sigma_2}, \ell$.
     - Hence

       $$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot A_{\ell,m}), \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle ; A_{\ell,m} \cdot \widehat{T_2}$$

       with $T_2' = A_{\ell,m} \cdot \widehat{T_2}$ by Lemma C.1.8 and (d).
     - By Lemma C.1.17 and (c) we get:
       * $\langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{t}{}^* \langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1', \mathbf{let}\ x = \mathbf{alloc}(y)\ \mathbf{in}\ e$ using **E.0** only
       * $\widehat{T_1}$ fsc from $\langle \Pi_1 \cdot A_{\ell,m}, \varepsilon \rangle, \sigma_1', \kappa_1, \rho_1'[x \mapsto \ell], e$
       * $\sigma_1, \rho_1', \mathbf{alloc}(y) \xrightarrow{s} \sigma_1', \ell$
       * $\rho_1'(y) = m$
     - From (b) we get $\Pi_2 \cdot A_{\ell,m}$ ok.
     - By induction then:
       (i) $\langle \Pi_2', T_1'' \rangle$ ok
       (ii) $\langle \Pi_3 \cdot A_{\ell,m}, \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3, \rho_1'[x \mapsto \ell], e \xrightarrow{t}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$
       (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot A_{\ell,m}, \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; T_2'$
     - From $\sigma_2, \varepsilon, \mathbf{alloc}(m) \xrightarrow{s} \widehat{\sigma_2}, \ell$ and the knowledge about ρ_1' follows

       $$\sigma_2|_{gc}, \rho_1', \mathbf{alloc}(y) \xrightarrow{s} \widehat{\sigma_2}|_{gc}, \ell$$

       .
     - Hence, using Lemma C.1.3, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1, \alpha_{r1}$
       $\xrightarrow{t}{}^* \langle \Pi_3 \cdot A_{\ell,m}, \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3, \rho_1'[x \mapsto \ell], e$
       $\xrightarrow{t}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$
   - Case **P.2**:
     - Then $T_1' = R^{\nu}_{\ell[m]} \cdot \widehat{T_1}$ and $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
       $\xrightarrow{t} \langle \Pi_2 \cdot R^{\nu}_{\ell[m]}, \widehat{T_1} \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
       $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$,
       where $\sigma_2, \varepsilon, \mathbf{read}(\ell[m]) \xrightarrow{s} \sigma_2, \nu$.

– Hence

$$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_\rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \mathsf{R}^{\nu}_{\ell[m]}), \_\rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_\rangle ; \mathsf{R}^{\nu}_{\ell[m]} \cdot \widehat{\mathsf{T}_2}$$

with $\mathsf{T}_2' = \mathsf{R}^{\nu}_{\ell[m]} \cdot \widehat{\mathsf{T}_2}$ by Lemma C.1.8 and (d).

– By Lemma C.1.17 and (c) we get:
  * $\langle \Pi_1, \varepsilon\rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1}$ $\xrightarrow{t}{}^{*}$ $\langle \Pi_1, \varepsilon\rangle, \sigma_1, \kappa_1, \rho_1', \mathbf{let}\ x = \mathbf{read}(y[z])\ \mathbf{in}\ e$ using **E.0** only
  * $\widehat{\mathsf{T}_1}$ fsc from $\langle \Pi_1 \cdot \mathsf{R}^{\nu}_{\ell[m]}, \varepsilon\rangle, \sigma_1, \kappa_1, \rho_1'[x \mapsto \nu], e$
  * $\sigma_1, \rho_1', \mathbf{read}(y[z]) \xrightarrow{s} \sigma_1, \nu$
  * $\rho_1'(y) = \ell$
  * $\rho_1'(z) = m$
– From (b) we get $\Pi_2 \cdot \mathsf{R}^{\nu}_{\ell[m]}$ ok.
– By induction then:
  (i) $\langle \Pi_2', \mathsf{T}_1''\rangle$ ok
  (ii) $\langle \Pi_3 \cdot \mathsf{R}^{\nu}_{\ell[m]}, \varepsilon\rangle, \sigma_2|_{gc}, \kappa_3, \rho_1'[x \mapsto \nu], e \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon\rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$
  (iii) $\langle \Pi_3', \varepsilon\rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot \mathsf{R}^{\nu}_{\ell[m]}, \varepsilon\rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \Pi_3, \varepsilon\rangle ; \mathsf{T}_2'$
– From $\sigma_2, \varepsilon, \mathbf{read}(\ell[m]) \xrightarrow{s} \sigma_2, \nu$ and the knowledge about ρ_1' follows

$$\sigma_2|_{gc}, \rho_1', \mathbf{read}(y[z]) \xrightarrow{s} \sigma_2|_{gc}, \nu$$

– Hence, by Lemma C.1.3, $\langle \Pi_3, \varepsilon\rangle, \sigma_2|_{gc}, \kappa_3, \rho_1, \alpha_{r1} \xrightarrow{t}{}^{*}$ $\langle \Pi_3 \cdot \mathsf{R}^{\nu}_{\ell[m]}, \varepsilon\rangle, \sigma_2|_{gc}, \kappa_3, \rho_1'[x \mapsto \nu], e \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon\rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$

- Case **P.3**:
  – Then $\mathsf{T}_1' = \mathsf{W}^{\nu}_{\ell[m]} \cdot \widehat{\mathsf{T}_1}$ and $\langle \Pi_2, \mathsf{T}_1'\rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$ $\xrightarrow{t} \langle \Pi_2 \cdot \mathsf{W}^{\nu}_{\ell[m]}, \widehat{\mathsf{T}_1}\rangle, \widehat{\sigma_2}, \kappa_2, \varepsilon, \mathbf{prop}$ $\xrightarrow{t}{}^{n-1}$ $\langle \Pi_2', \mathsf{T}_1''\rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$, where $\sigma_2, \varepsilon, \mathbf{write}(\ell[m],\nu) \xrightarrow{s} \widehat{\sigma_2}, 0$.
  – Hence

$$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_\rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \mathsf{W}^{\nu}_{\ell[m]}), \_\rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_\rangle ; \mathsf{W}^{\nu}_{\ell[m]} \cdot \widehat{\mathsf{T}_2}$$

with $\mathsf{T}_2' = \mathsf{W}^{\nu}_{\ell[m]} \cdot \widehat{\mathsf{T}_2}$ by Lemma C.1.8 and (d).

  – By Lemma C.1.17 and (c) we get:
    * $\langle \Pi_1, \varepsilon\rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{t}{}^{*} \langle \Pi_1, \varepsilon\rangle, \sigma_1, \kappa_1, \rho_1', \mathbf{let}\ \_ = \mathbf{write}(x[y],z)\ \mathbf{in}\ e$ using **E.0** only
    * $\widehat{\mathsf{T}_1}$ fsc from $\langle \Pi_1 \cdot \mathsf{W}^{\nu}_{\ell[m]}, \varepsilon\rangle, \sigma_1', \kappa_1, \rho_1', e$

  - $\sigma_1, \rho_1', \mathbf{write}(x[y],z) \xrightarrow{s} \sigma_1', 0$
  - $\rho_1'(x) = \ell$
  - $\rho_1'(y) = m$
  - $\rho_1'(z) = \nu$
- From (b) we get $\Pi_2 \cdot W^{\nu}_{\ell[m]}$ ok.
- By induction then:
  (i) $\langle \Pi_2', T_1'' \rangle$ ok
  (ii) $\langle \Pi_3 \cdot W^{\nu}_{\ell[m]}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1', e \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$.
  (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \Pi_3 \cdot W^{\nu}_{\ell[m]}, \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; T_2'$
- From $\sigma_2, \varepsilon, \mathbf{write}(\nu[\ell],m) \xrightarrow{s} \sigma_2', 0$ and the knowledge about ρ_1' follows $\sigma_2|_{gc}, \rho_1', \mathbf{write}(x[y],z) \xrightarrow{s} \sigma_2'|_{gc}, 0$.
- Hence, using Lemma C.1.3, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1, \alpha_{r1}$
  $\xrightarrow{t}{}^{*} \langle \Pi_3 \cdot W^{\nu}_{\ell[m]}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1', e$
  $\xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$

- Case **P.4**:
  - Then $T_1' = M_{\rho,e} \cdot \widehat{T_1}$ and $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
    $\xrightarrow{t} \langle \Pi_2 \cdot M_{\rho,e}, \widehat{T_1} \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$.
  - Hence $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^{*} \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot M_{\rho,e}), \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle ; M_{\rho,e} \cdot \widehat{T_2}$ with $T_2' = M_{\rho,e} \cdot \widehat{T_2}$ by Lemma C.1.8 and (d).
  - By Lemma C.1.17 and (c) we get:
    - $\langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{t}{}^{*} \langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho, \mathbf{memo}\ e$ using **E.0** only
    - $\widehat{T_1}$ fsc from $\langle \Pi_1 \cdot M_{\rho,e}, \varepsilon \rangle, \sigma_1, \kappa_1, \rho, e$
  - From (b) we get $\Pi_2 \cdot M_{\rho,e}$ ok.
  - By induction then:
    (i) $\langle \Pi_2', T_1'' \rangle$ ok
    (ii) $\langle \Pi_3 \cdot M_{\rho,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho, e \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$
    (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \Pi_3 \cdot M_{\rho,e}, \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; T_2'$
  - Finally, using Lemma C.1.3, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1, \alpha_{r1}$
    $\xrightarrow{t}{}^{*} \langle \Pi_3 \cdot M_{\rho,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho, e$
    $\xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$

- Case **P.5**:
  - Then $T_1' = U_{\rho,e} \cdot \widehat{T_1}$ and $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
    $\xrightarrow{t} \langle \Pi_2 \cdot U_{\rho,e}, \widehat{T_1} \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$.

- Hence $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_\rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \mathsf{U}_{\rho,e}), \_\rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_\rangle ; \mathsf{U}_{\rho,e} \cdot \widehat{\mathsf{T}_2}$ with $\mathsf{T}_2' = \mathsf{U}_{\rho,e} \cdot \widehat{\mathsf{T}_2}$ by Lemma C.1.8 and (d).
- By Lemma C.1.17 and (c) we get:
  - $\langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{t}{}^* \langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho,$ **update** e using **E.0** only
  - $\widehat{\mathsf{T}_1}$ fsc from $\langle \Pi_1 \cdot \mathsf{U}_{\rho,e}, \varepsilon \rangle, \sigma_1, \kappa_1, \rho, e$
- From (b) we get $\Pi_2 \cdot \mathsf{U}_{\rho,e}$ ok.
- By induction then:
  (i) $\langle \Pi_2', \mathsf{T}_1'' \rangle$ ok
  (ii) $\langle \Pi_3 \cdot \mathsf{U}_{\rho,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho, e \xrightarrow{t}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$
  (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot \mathsf{U}_{\rho,e}, \varepsilon \rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; \mathsf{T}_2'$
- Finally, using Lemma C.1.3, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1, \alpha_{r1}$
  $\xrightarrow{t}{}^* \langle \Pi_3 \cdot \mathsf{U}_{\rho,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho, e$
  $\xrightarrow{t}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_2, \varepsilon, \overline{\nu_2}$

- Case **P.6**:
  - Then $\mathsf{T}_1' = (\widetilde{\mathsf{T}_1}) \cdot \widehat{\mathsf{T}_1}$ and $\langle \Pi_2, \mathsf{T}_1' \rangle, \sigma_2, \kappa_2, \varepsilon,$ **prop**
    $\xrightarrow{t} \langle \Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}, \widetilde{\mathsf{T}_1} \rangle, \sigma_2, \kappa_2, \varepsilon,$ **prop**
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', \mathsf{T}_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}.$
  - By Lemmas C.1.4 and C.1.13 we get:
    - $\langle \Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}, \widetilde{\mathsf{T}_1} \rangle, \sigma_2, \kappa_2, \varepsilon,$ **prop** $\xrightarrow{t}{}^{n_1} \langle \widehat{\Pi_2}, \varepsilon \rangle, \widehat{\sigma_2}, \kappa_2, \varepsilon, \overline{\nu}$
    - $\langle \widehat{\Pi_2}, \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}, \varepsilon \rangle; \mathsf{T}$
    - $\langle \widehat{\Pi_2}, \varepsilon \rangle, \widehat{\sigma_2}, \kappa_2, \varepsilon, \overline{\nu} \xrightarrow{t} \langle \Pi_2 \cdot (\mathsf{T}), \widehat{\mathsf{T}_1} \rangle, \widehat{\sigma_2}, \kappa_2, \varepsilon,$ **prop**
    - $\langle \Pi_2 \cdot (\mathsf{T}), \widehat{\mathsf{T}_1} \rangle, \widehat{\sigma_2}, \kappa_2, \varepsilon,$ **prop** $\xrightarrow{t}{}^{n_2} \langle \Pi_2', \mathsf{T}_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
    - $n - 1 = n_1 + 1 + n_2$
  - By Lemma C.1.17 and (c) we get:
    - $\langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{t}{}^* \langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho,$ **push** f **do** e using **E.0** only
    - $\widetilde{\mathsf{T}_1}$ fsc from $\langle \Pi_1 \cdot \square, \varepsilon \rangle, \sigma_1, \kappa_1 \cdot \lfloor \rho, f \rfloor, \rho, e,$ producing value $\overline{\omega}$
    - $\widehat{\mathsf{T}_1}$ fsc from $\langle \Pi_1 \cdot (\widetilde{\mathsf{T}_1}), \varepsilon \rangle, \sigma_1', \kappa_1, \rho', e'$
    - $\rho(f) =$ **fun** $f(\overline{x}).e'$
    - $\rho' = \rho[\overline{x \mapsto \omega}]$
  - Since Π_2 ok and $\widehat{\mathsf{T}_1}$ fsc, we know $\Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}$ ok.
  - Furthermore, $\langle \widehat{\Pi_2}, \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}, \varepsilon \rangle; \mathsf{T}$
    implies $\langle \mathsf{drop}_{\boxminus}(\widehat{\Pi_2}), \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}), \varepsilon \rangle; \mathsf{T}$ by Lemma C.1.6.
  - Induction with n_1 then yields:
    - $\langle \widehat{\Pi_2}, \varepsilon \rangle$ ok
    - $\langle \Pi_3 \cdot \square, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3 \cdot \lfloor \rho, f \rfloor, \rho, e \xrightarrow{t}{}^* \langle \widehat{\Pi_3}, \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3 \cdot \lfloor \rho, f \rfloor, \varepsilon, \overline{\nu}$

  * $\langle \widehat{\Pi_3}, \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot \Box, \varepsilon \rangle; \mathsf{T}$
- Since Π_2 ok we get $\Pi_2 \cdot (\mathsf{T})$ ok.
- We get $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot (\mathsf{T})), \_ \rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle; (\mathsf{T}) \cdot \widehat{\mathsf{T}_2}$ with $\mathsf{T}_2' = (\mathsf{T}) \cdot \widehat{\mathsf{T}_2}$ by Lemma C.1.8 and (d).
- Induction with n_2 then yields:
  * $\langle \Pi_2', \mathsf{T}_1'' \rangle$ ok
  * $\langle \Pi_3 \cdot (\mathsf{T}), \varepsilon \rangle, \widehat{\sigma_2}|_{\mathsf{gc}}, \kappa_3, \rho', e' \xrightarrow{\mathsf{t}}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{v_2}$
  * $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot (\mathsf{T}), \varepsilon \rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle; \mathsf{T}_2'$
- Finally, using Lemma C.1.3, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_1, \alpha_{r1}$ $\xrightarrow{\mathsf{t}}{}^* \langle \Pi_3 \cdot \Box, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3 \cdot \lfloor \rho, \mathsf{f} \rfloor, \rho, e$
- $\langle \Pi_3 \cdot \Box, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3 \cdot \lfloor \rho, \mathsf{f} \rfloor, \rho, e \xrightarrow{\mathsf{t}}{}^* \langle \widehat{\Pi_3}, \varepsilon \rangle, \widehat{\sigma_2}|_{\mathsf{gc}}, \kappa_3 \cdot \lfloor \rho, \mathsf{f} \rfloor, \varepsilon, \overline{v}$
- $\langle \widehat{\Pi_3}, \varepsilon \rangle, \widehat{\sigma_2}|_{\mathsf{gc}}, \kappa_3 \cdot \lfloor \rho, \mathsf{f} \rfloor, \varepsilon, \overline{v}$ $\xrightarrow{\mathsf{t}} \langle \Pi_3 \cdot (\mathsf{T}), \varepsilon \rangle, \widehat{\sigma_2}|_{\mathsf{gc}}, \kappa_3, \rho'', e'$, where $\rho'' = \rho[\overline{x \mapsto v}]$

- It remains to show that $\overline{w} = \overline{v}$ and thus $\rho' = \rho''$.
- Recall that we have:
  * $\langle \Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}, \widetilde{\mathsf{T}_1} \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop} \xrightarrow{\mathsf{t}}{}^{n_1} \langle \widehat{\Pi_2}, \varepsilon \rangle, \widehat{\sigma_2}, \kappa_2, \varepsilon, \overline{v}$
  * $\langle \widehat{\Pi_2}, \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}, \varepsilon \rangle; \mathsf{T}$
  * $\Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}$ ok
  * $\widetilde{\mathsf{T}_1}$ fsc
- By Lemmas C.1.6 and C.1.21, we get two subcases:
  (a) First subcase (of two)
    * In this subcase, we have that:
      · $\langle \Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}, \widetilde{\mathsf{T}_1} \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop} \xrightarrow{\mathsf{t}}{}^{m_1} \langle \Pi'', \mathsf{U}_{\widehat{\rho},\widehat{e}} \cdot \mathsf{T}' \rangle, \widehat{\sigma_2'}, \kappa_2, \varepsilon, \mathbf{prop}$
      · $\langle \Pi'', \mathsf{U}_{\widehat{\rho},\widehat{e}} \cdot \mathsf{T}' \rangle, \widehat{\sigma_2'}, \kappa_2, \varepsilon, \mathbf{prop} \xrightarrow{\mathsf{t}} \langle \Pi'' \cdot \mathsf{U}_{\widehat{\rho},\widehat{e}}, \mathsf{T}' \rangle, \widehat{\sigma_2'}, \kappa_2, \widehat{\rho}, \widehat{e}$
      · $\langle \Pi'' \cdot \mathsf{U}_{\widehat{\rho},\widehat{e}}, \mathsf{T}' \rangle, \widehat{\sigma_2'}, \kappa_2, \widehat{\rho}, \widehat{e} \xrightarrow{\mathsf{t}}{}^{m_2} \langle \widehat{\Pi_2}, \varepsilon \rangle, \widehat{\sigma_2}, \kappa_2, \varepsilon, \overline{v}$
      · $\langle \mathsf{drop}_{\boxminus}(\widehat{\Pi_2}), \_ \rangle; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi'' \cdot \mathsf{U}_{\rho,e}), \_ \rangle; \_ \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \boxplus_{\widehat{\mathsf{T}_1}}), \_ \rangle; \_$
      · Π'' ok
      · $\mathsf{U}_{\widehat{\rho},\widehat{e}} \cdot \mathsf{T}'$ fsc
      · $\mathsf{last}(\widetilde{\mathsf{T}_1}) = \mathsf{last}(\mathsf{T}')$
      · $n_1 = m_1 + 1 + m_2$
    * By Lemma C.1.17 we get $\langle \Pi'' \cdot \mathsf{U}_{\widehat{\rho},\widehat{e}}, \mathsf{T}' \rangle$ ok.
    * From induction on reduction m_2 using part (2), we get:
      · $\langle \varepsilon, \varepsilon \rangle, \widehat{\sigma_2'}|_{\mathsf{gc}}, \kappa_2, \widehat{\rho}, \widehat{e} \xrightarrow{\mathsf{t}}{}^* \langle \Pi_4, \varepsilon \rangle, \widehat{\sigma_2}|_{\mathsf{gc}}, \kappa_2, \varepsilon, \overline{v}$
      · $\langle \Pi_4, \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \varepsilon, \varepsilon \rangle; \_$

* From Lemma C.1.15, we have that $\langle \varepsilon, \varepsilon \rangle, \widehat{\sigma_2'|_{gc}}, \varepsilon, \widehat{\rho}, \widehat{e} \xrightarrow{t}{}^{*} \langle \Pi_4, \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \varepsilon, \varepsilon, \overline{\nu}$
* From Lemma C.1.9 and Lemma C.1.1 we have that: $\widehat{\sigma_2'|_{gc}}, \varepsilon, \widehat{\rho}, \widehat{e} \xrightarrow{r}{}^{*} \widehat{\sigma_2}|_{gc}, \varepsilon, \varepsilon, \overline{\nu}$
* Next, since $U_{\widehat{\rho},\widehat{e}} \cdot T'$ fsc, with the help of Lemma C.1.17 there exists Π_5, σ_5, κ_5, ρ_5, e_5, Π_5', σ_5' and $\overline{\omega'}$ such that
  - $\mathsf{CSA}(\sigma_5, \rho_5, e_5)$
  - $\mathsf{noreuse}(\langle \Pi_5, \varepsilon \rangle)$
  - $\langle \Pi_5, \varepsilon \rangle, \sigma_5, \kappa_5, \rho_5, e_5 \xrightarrow{t}{}^{*} \langle \Pi_5, \varepsilon \rangle, \sigma_5, \kappa_5, \widehat{\rho}, \textbf{update } \widehat{e}$ using **E.0** only
  - T' fsc from $\langle \Pi_5 \cdot U_{\widehat{\rho},\widehat{e}}, \varepsilon \rangle, \sigma_5, \kappa_5, \widehat{\rho}, \widehat{e}$ producing $\overline{\omega'}$
  - $\langle \Pi_5 \cdot U_{\widehat{\rho},\widehat{e}}, \varepsilon \rangle, \sigma_5, \kappa_5, \widehat{\rho}, \widehat{e} \xrightarrow{t}{}^{*} \langle \Pi_5', \varepsilon \rangle, \sigma_5', \kappa_5, \varepsilon, \overline{\omega'}$
  - $\langle \Pi_5', \varepsilon \rangle ; \varepsilon \circlearrowleft^{*} \langle \Pi_5 \cdot U_{\widehat{\rho},\widehat{e}}, \varepsilon \rangle ; T'$
* From T' fsc, $\widetilde{T_1}$ fsc and $\mathsf{last}(T') = \mathsf{last}(\widetilde{T_1})$ we have $\overline{\omega}' = \mathsf{last}(T') = \mathsf{last}(\widetilde{T_1}) = \overline{\omega}$ by Lemma C.1.18.
* By Lemma C.1.16 we get

$$\mathsf{CSA}(\sigma_5, \widehat{\rho}, \textbf{update } \widehat{e})$$

  and thus

$$\mathsf{SA}(\sigma_5, \widehat{\rho}, \textbf{update } \widehat{e})$$

* From Lemmas C.1.4, C.1.5, C.1.9, C.1.15, and C.1.1 we have

$$\sigma_5, \varepsilon, \widehat{\rho}, \widehat{e} \xrightarrow{r}{}^{*} \sigma_5', \varepsilon, \varepsilon, \overline{\omega}$$

* Since also $\widehat{\sigma_2'|_{gc}}, \varepsilon, \widehat{\rho}, \widehat{e} \xrightarrow{r}{}^{*} \widehat{\sigma_2}|_{gc}, \varepsilon, \varepsilon, \overline{\nu}$ we have that $\overline{\nu} = \overline{\omega}$ by definition of SA.

(b) Second (and last) subcase:

* In this subcase, we have that:
  - $\langle \Pi_2 \cdot \boxplus_{\widehat{T_1}}, \widetilde{T_1} \rangle, \sigma_2, \kappa_2, \varepsilon, \textbf{prop} \xrightarrow{t}{}^{m_1} \langle \Pi'', T' \rangle, \widehat{\sigma_2'}, \kappa_2, \varepsilon, \textbf{prop} \xrightarrow{t}{}^{m_2} \widehat{\langle \Pi_2, \varepsilon \rangle}, \widehat{\sigma_2}, \kappa_2, \varepsilon, \overline{\nu}$
  - $\langle \mathsf{drop}_{\boxminus}(\widehat{\Pi_2}), \_ \rangle ; \_ \circlearrowleft^{*} \langle \mathsf{drop}_{\boxminus}(\Pi'')\_, ; \rangle \_ \circlearrowleft^{*} \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \boxplus_{\widehat{T_1}}), \_ \rangle ; \_$
  - $\mathsf{last}(\widetilde{T_1}) = \mathsf{last}(T')$
  - Reduction m_2 contains no use of **P.E**
* Applying Lemma C.1.19 to the reduction m_2 we have that $\mathsf{last}(T') = \overline{\nu}$.
* Putting this together, we have that $\mathsf{last}(\widetilde{T_1}) = \mathsf{last}(T') = \overline{\nu}$.
* Finally, by applying Lemma C.1.18 to "$\widetilde{T_1}$ fsc from ... producing $\overline{\omega}$", we have that $\mathsf{last}(\widetilde{T_1}) = \overline{\omega}$.

* Hence, $\overline{\omega} = \overline{\nu}$.

- Case **P.7**:
  - Then $T_1' = \overline{\nu_1}$ and $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
    $\xrightarrow{t} \langle \Pi_2 \cdot \overline{\nu_1}, \varepsilon \rangle, \sigma_2, \kappa_2, \varepsilon, \overline{\nu_1}$
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$.
  - We show that $n - 1 = 0$:
    * Assume the contrary. The only reduction rules that apply to

      $$\langle \Pi_2 \cdot \overline{\nu_1}, \varepsilon \rangle, \sigma_2, \kappa_2, \varepsilon, \overline{\nu_1}$$

      are **E.8** and **P.8**. We consider only the former case; the latter is analogous.
    * Hence $\langle \Pi_2 \cdot \overline{\nu_1}, \varepsilon \rangle, \sigma_2, \kappa_2, \varepsilon, \overline{\nu_1}$
      $\xrightarrow{t} \langle \Pi_2'' \cdot (T), T' \rangle, \sigma_2, \kappa_2', \rho_f, e_f$
      $\xrightarrow{t}{}^{n-2} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$ where

      $$\langle \Pi_2 \cdot \overline{\nu_1}, \varepsilon \rangle ; \varepsilon \circlearrowleft^* \langle \Pi_2'' \cdot \Box, T' \rangle ; T$$

    * By Lemma C.1.4 we know $\Pi_2'' \cdot \Box \in \mathsf{Prefixes}(\Pi_2 \cdot \overline{\nu_1})$, i.e., $\Pi_2'' \cdot \Box \in \mathsf{Prefixes}(\Pi_2)$.
    * Hence

      $$\mathsf{drop}_{\boxminus}(\Pi_2''), \mathsf{drop}_{\boxminus}(\Pi_2'' \cdot \Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_2)) \subseteq \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_2'))$$

      using Lemma C.1.4, (d), and Lemma C.1.5.
    * Using Lemma C.1.7 we get $\mathsf{drop}_{\boxminus}(\Pi_2'' \cdot (T)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_2'))$, contradicting $\mathsf{drop}_{\boxminus}(\Pi_2'' \cdot \Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_2'))$.
  - Hence $\overline{\nu_1} = \overline{\nu_2}$ and $\sigma_2' = \sigma_2$ and $\Pi_2' = \Pi_2 \cdot \overline{\nu_2}$.
  - By inversion on (d) we get $T_2' = \overline{\nu_2}$.
  - $\langle \Pi_2 \cdot \overline{\nu_2}, \varepsilon \rangle$ ok follows from (b).
  - By Lemma C.1.17 and (c) we get

    $$\langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{t}{}^* \langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1', \mathbf{pop}\ \overline{x}$$

    using only **E.0**, where $\rho_1'(\overline{x}) = \overline{\nu_1}$.
  - Hence, using Lemma C.1.3, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1, \alpha_{r1}$
    $\xrightarrow{t}{}^* \langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1', \mathbf{pop}\ \overline{x}$
    $\xrightarrow{t} \langle \Pi_3 \cdot \overline{\nu_2}, \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \nu_2$.
  - Finally, $\langle \Pi_3 \cdot \overline{\nu_2}, \varepsilon \rangle ; \varepsilon \circlearrowleft^* \langle \Pi_3, \varepsilon \rangle ; \overline{\nu_2}$.

- Case **P.E**:

- Then $T_1' = U_{\rho,e} \cdot \widehat{T_1}$ and $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \varepsilon,$ **prop**
  $\xrightarrow{t} \langle \Pi_2 \cdot U_{\rho,e}, \widehat{T_1} \rangle, \sigma_2, \kappa_2, \rho, e$
  $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$.
- Hence $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot U_{\rho,e}), \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle ; U_{\rho,e} \cdot \widehat{T_2}$ with $T_2' = U_{\rho,e} \cdot \widehat{T_2}$ by Lemma C.1.8 and (d).
- By Lemma C.1.17 and (a) we get:
  * $\langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{t}{}^* \langle \Pi_1, \varepsilon \rangle, \sigma_1, \kappa_1, \rho,$ **update** e using **E.0** only
  * $\widehat{T_1}$ fsc from $\langle \Pi_1 \cdot U_{\rho,e}, \varepsilon \rangle, \sigma_1, \kappa_1, \rho, e$ and thus $\widehat{T_1}$ ok
- $\Pi_2 \cdot U_{\rho,e}$ ok follows from Π_2 ok.
- Induction and part (2) then yield:
  (i) $\langle \Pi_2', T_1'' \rangle$ ok
  (ii) $\langle \Pi_3 \cdot U_{\rho,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho, e \xrightarrow{t}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{\nu_2}$
  (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot U_{\rho,e}, \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; T_2'$
- Finally, using Lemma C.1.3, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_1, \alpha_{r1}$
  $\xrightarrow{t}{}^* \langle \Pi_3 \cdot U_{\rho,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho, e$.

- Cases **E.0–8**, **E.P**, **P.8**, **U.1–4**: not possible

2. Case analysis on n. First, we handle the simple case when $n = 0$:

   - Since $n = 0$, we have that:
     - $\langle \Pi_2, T_1' \rangle = \langle \Pi_2', T_1'' \rangle$
     - $\sigma_2 = \sigma_2'$
     - $\rho_2 = \epsilon$
     - $\alpha_{r2} = \overline{\nu}_2$
     - $T_2' = \varepsilon$, by inversion on (c)
   - $\langle \Pi_2', T_1'' \rangle$ ok is given.
   - Pick $\Pi_3' = \Pi_3$.
   - Then reflexively we have that

     $$\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, \alpha_{r2} \xrightarrow{t}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{\nu_2}$$

   - Similarly, reflexively we have that $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3, \varepsilon \rangle; T_2'$.

   For $n > 0$ we inspect the first reduction step of (a):

   - Case **E.0**.
     - Then $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, e^u$
       $\xrightarrow{t} \langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2', \alpha_{r2}'$
       $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$

- Induction yields:
  (i) $\langle \Pi_2', T_1'' \rangle$ ok
  (ii) $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2', \alpha_{r_2}' \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{v_2}$
  (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \Pi_3, \varepsilon \rangle; T_2'$
- Finally, using Lemma C.1.3 and (ii) we have that
  $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, e^{u}$
  $\xrightarrow{t}{}^{*} \langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2', \alpha_{r_2}'$
  $\xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{v_2}$

- Case **E.1**:
  - Then $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, \textbf{let } x = \textbf{alloc}(y) \textbf{ in } e$
    $\xrightarrow{t} \langle \Pi_2 \cdot A_{\ell,m}, T_1' \rangle, \widehat{\sigma_2}, \kappa_2, \rho_2', e$
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{v_2}$
    Where:
    * $\sigma_2, \rho_2, \textbf{alloc}(y) \xrightarrow{s} \widehat{\sigma_2}, \ell$
    * $\rho_2(y) = m$
    * $\rho_2' = \rho_2[x \mapsto \ell]$
  - From Π_2 ok we have that $\Pi_2 \cdot A_{\ell,m}$ ok
  - By Lemma C.1.8 and (c) we have

    $$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^{*} \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot A_{\ell,m}), \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle; T_2'$$

    with $T_2' = A_{\ell,m} \cdot \widehat{T_2}$
  - By induction then:
    (i) $\langle \Pi_2', T_1'' \rangle$ ok
    (ii) $\langle \Pi_3 \cdot A_{\ell,m}, \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3, \rho_2', e \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{v_2}$
    (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \Pi_3 \cdot A_{\ell,m}, \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle; T_2'$
  - Hence, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, \textbf{let } x = \textbf{alloc}(y) \textbf{ in } e$
    $\xrightarrow{t} \langle \Pi_3 \cdot A_{\ell,m}, \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3, \rho_2', e$
    $\xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{v_2}$

- Case **E.2**:
  - Then $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, \textbf{let } x = \textbf{read}(y[z]) \textbf{ in } e$
    $\xrightarrow{t} \langle \Pi_2 \cdot R^{v}_{\ell[m]}, T_1' \rangle, \sigma_2, \kappa_2, \rho_2', e$
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{v_2}$
    Where:
    * $\sigma_2, \rho_2, \textbf{read}(y[z]) \xrightarrow{s} \sigma_2, v$
    * $\rho_2(y) = \ell$

  * $\rho_2(z) = m$
  * $\rho_2' = \rho_2[x \mapsto \nu]$
- From Π_2 ok we have that $\Pi_2 \cdot \mathsf{R}^{\nu}_{\ell[m]}$ ok
- By Lemma C.1.8 and (c) we have

$$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_\rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \mathsf{R}^{\nu}_{\ell[m]}), \_\rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_\rangle ; \mathsf{T}_2'$$

  with $\mathsf{T}_2' = \mathsf{R}^{\nu}_{\ell[m]} \cdot \widehat{\mathsf{T}_2}$
- By induction then:
  (i) $\langle \Pi_2', \mathsf{T}_1'' \rangle$ ok
  (ii) $\langle \Pi_3 \cdot \mathsf{R}^{\nu}_{\ell[m]}, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_2', e \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$
  (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot \mathsf{R}^{\nu}_{\ell[m]}, \varepsilon \rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; \mathsf{T}_2'$
- Hence, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_2, \mathbf{let}\ x = \mathbf{read}(y[z])\ \mathbf{in}\ e$
  $\xrightarrow{t} \langle \Pi_3 \cdot \mathsf{R}^{\nu}_{\ell[m]}, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_2', e$
  $\xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$

- Case **E.3**:
  - Then $\langle \Pi_2, \mathsf{T}_1' \rangle, \sigma_2, \kappa_2, \rho_2, \mathbf{let}\ \_ = \mathbf{write}(x[y],z)\ \mathbf{in}\ e$
    $\xrightarrow{t} \langle \Pi_2 \cdot \mathsf{W}^{\nu}_{\ell[m]}, \mathsf{T}_1' \rangle, \widehat{\sigma_2}, \kappa_2, \rho_2, e$
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', \mathsf{T}_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
    Where:
    * $\sigma_2, \rho_2, \mathbf{write}(x[y],z) \xrightarrow{s} \widehat{\sigma_2}, 0$
    * $\rho_2(x) = \ell$
    * $\rho_2(y) = m$
    * $\rho_2(z) = \nu$
  - From Π_2 ok we have that $\Pi_2 \cdot \mathsf{W}^{\nu}_{\ell[m]}$ ok
  - By Lemma C.1.8 and (c) we have

$$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_\rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \mathsf{W}^{\nu}_{\ell[m]}), \_\rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_\rangle ; \mathsf{T}_2'$$

    with $\mathsf{T}_2' = \mathsf{W}^{\nu}_{\ell[m]} \cdot \widehat{\mathsf{T}_2}$
  - By induction then:
    (i) $\langle \Pi_2', \mathsf{T}_1'' \rangle$ ok
    (ii) $\langle \Pi_3 \cdot \mathsf{W}^{\nu}_{\ell[m]}, \varepsilon \rangle, \widehat{\sigma_2}|_{\mathsf{gc}}, \kappa_3, \rho_2, e \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$
    (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot \mathsf{W}^{\nu}_{\ell[m]}, \varepsilon \rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; \mathsf{T}_2'$
  - Hence, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_2, \mathbf{let}\ \_ = \mathbf{write}(x[y],z)\ \mathbf{in}\ e$
    $\xrightarrow{t} \langle \Pi_3 \cdot \mathsf{W}^{\nu}_{\ell[m]}, \varepsilon \rangle, \widehat{\sigma_2}|_{\mathsf{gc}}, \kappa_3, \rho_2, e$
    $\xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$

- Case **E.4**:
  - Then $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, \mathbf{memo}\ e \xrightarrow{t} \langle \Pi_2 \cdot M_{\rho_2,e}, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, e \xrightarrow{t}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
  - From Π_2 ok we have that $\Pi_2 \cdot M_{\rho_2,e}$ ok
  - By Lemma C.1.8 and (c) we have

    $$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot M_{\rho_2,e}), \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle ; T_2'$$

    with $T_2' = M_{\rho_2,e} \cdot \widehat{T_2}$
  - By induction then:

    (i) $\langle \Pi_2', T_1'' \rangle$ ok

    (ii) $\langle \Pi_3 \cdot M_{\rho_2,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, e \xrightarrow{t}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{\nu_2}$

    (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot M_{\rho_2,e}, \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; T_2'$
  - Hence, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, \mathbf{memo}\ e \xrightarrow{t} \langle \Pi_3 \cdot M_{\rho_2,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, e \xrightarrow{t}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{\nu_2}$
- Case **E.5**:
  - Then $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, \mathbf{update}\ e \xrightarrow{t} \langle \Pi_2 \cdot U_{\rho_2,e}, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, e \xrightarrow{t}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
  - From Π_2 ok we have that $\Pi_2 \cdot U_{\rho_2,e}$ ok
  - By Lemma C.1.8 and (c) we have

    $$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot U_{\rho_2,e}), \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle ; T_2'$$

    with $T_2' = U_{\rho_2,e} \cdot \widehat{T_2}$
  - By induction then:

    (i) $\langle \Pi_2', T_1'' \rangle$ ok

    (ii) $\langle \Pi_3 \cdot U_{\rho_2,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, e \xrightarrow{t}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{\nu_2}$

    (iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot U_{\rho_2,e}, \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; T_2'$
  - Hence, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, \mathbf{update}\ e \xrightarrow{t} \langle \Pi_3 \cdot U_{\rho_2,e}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, e \xrightarrow{t}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{\nu_2}$
- Case **E.6**:
  - Then $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, \mathbf{push}\ f\ \mathbf{do}\ e$
    $\xrightarrow{t} \langle \Pi_2 \cdot \Box, T_1' \rangle, \sigma_2, \kappa_2 \cdot \lfloor \rho_2, f \rfloor, \rho_2, e$
    $\xrightarrow{t}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
  - Note that also $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, \mathbf{push}\ f\ \mathbf{do}\ e$
    $\xrightarrow{t} \langle \Pi_3 \cdot \Box, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3 \cdot \lfloor \rho_2, f \rfloor, \rho_2, e$.
  - By Lemma C.1.12 the $n-1$ reduction above decomposes as follows:

  - $\langle \Pi_2 \cdot \Box, T_1' \rangle, \sigma_2, \kappa_2 \cdot \lfloor \rho_2, f \rfloor, \rho_2, e \xrightarrow{t}{}^{n_1} \langle \widehat{\Pi_2}, \widehat{T_1'} \rangle, \widehat{\sigma_2}, \kappa_2 \cdot \lfloor \rho_2, f \rfloor, \varepsilon, \overline{v}$
  - $\langle \widehat{\Pi_2}, \widehat{T_1'} \rangle; \varepsilon \circlearrowleft^* \langle \Pi_2 \cdot \Box, \widetilde{T_1'} \rangle; T$
  - $\langle \widehat{\Pi_2}, \widehat{T_1'} \rangle, \widehat{\sigma_2}, \kappa_2 \cdot \lfloor \rho_2, f \rfloor, \varepsilon, \overline{v} \xrightarrow{t} \langle \Pi_2 \cdot (T), \widetilde{T_1'} \rangle, \widehat{\sigma_2}, \kappa_2, \widehat{\rho_2}, e_f$
  - $\langle \Pi_2 \cdot (T), \widetilde{T_1'} \rangle, \widehat{\sigma_2}, \kappa_2, \widehat{\rho_2}, e_f \xrightarrow{t}{}^{n_2} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{v_2}$
  - $n = n_1 + 1 + n_2$
- From $\langle \Pi_2, T_1' \rangle$ ok we get $\langle \Pi_2 \cdot \Box, T_1' \rangle$ ok.
- From $\langle \widehat{\Pi_2}, \widetilde{T_1'} \rangle; \varepsilon \circlearrowleft^* \langle \Pi_2 \cdot \Box, \widetilde{T_1'} \rangle; T$ follows

$$\langle \mathsf{drop}_{\boxminus}(\widehat{\Pi_2}), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \Box), \_ \rangle; T$$

  by Lemma C.1.6.
- Hence induction with n_1 yields:
  - $\langle \widehat{\Pi_2}, \widehat{T_1'} \rangle$ ok
  - $\langle \Pi_3 \cdot \Box, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3 \cdot \lfloor \rho_2, f \rfloor, \rho_2, e \xrightarrow{t}{}^* \langle \Pi_3'', \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3 \cdot \lfloor \rho_2, f \rfloor, \varepsilon, \overline{v}$
  - $\langle \Pi_3'', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot \Box, \varepsilon \rangle; T$
- Note that $\langle \Pi_3'', \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3 \cdot \lfloor \rho_2, f \rfloor, \varepsilon, \overline{v} \xrightarrow{t} \langle \Pi_3 \cdot (T), \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3, \widehat{\rho_2}, e_f$.
- $\langle \Pi_2 \cdot (T), \widetilde{T_1'} \rangle$ ok follows from $\langle \widehat{\Pi_2}, \widehat{T_1'} \rangle$ ok by Lemma C.1.2.
- By Lemma C.1.8 and (c) we have

$$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot (T)), \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle ; T_2'$$

  with $T_2' = (T) \cdot \widehat{T_2}$
- So induction with n_2 yields:
  - $\langle \widehat{\Pi_2'}, T_1'' \rangle$ ok
  - $\langle \Pi_3 \cdot (T), \varepsilon \rangle, \widehat{\sigma_2}|_{gc}, \kappa_3, \widehat{\rho_2}, e_f \xrightarrow{t}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{v_2}$
  - $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \Pi_3 \cdot (T), \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; T_2'$
- Finally, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, \textbf{push } f \textbf{ do } e$
  $\xrightarrow{t}{}^* \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{v_2}$ by putting the pieces together.

- Case **E.7**
  - Then $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, \textbf{pop } \overline{x}$
    $\xrightarrow{t} \langle \Pi_2 \cdot \overline{v}, T_1' \rangle, \sigma_2, \kappa_2, \varepsilon, \overline{v}$
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{v_2}$
    Where: $\rho_2(x_i)_1^{|\overline{x}|} = \overline{v}$
  - From Π_2 ok we have that $\Pi_2 \cdot \overline{v}$ ok
  - By Lemma C.1.8 and (c) we have $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \overline{v}), \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle ; T_2'$ with $T_2' = \overline{v} \cdot \widehat{T_2}$
  - By induction then:

(i) $\langle \Pi_2', T_1'' \rangle$ ok

(ii) $\langle \Pi_3 \cdot \overline{\nu}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \varepsilon, \overline{\nu} \xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{\nu_2}$

(iii) $\langle \Pi_3', \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \Pi_3 \cdot \overline{\nu}, \varepsilon \rangle; \widehat{T_2} \circlearrowleft \langle \Pi_3, \varepsilon \rangle ; T_2'$

– Hence, $\langle \Pi_3, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \rho_2, \mathbf{pop}\ \overline{x}$
$\xrightarrow{t} \langle \Pi_3 \cdot \overline{\nu}, \varepsilon \rangle, \sigma_2|_{gc}, \kappa_3, \varepsilon, \overline{\nu}$
$\xrightarrow{t}{}^{*} \langle \Pi_3', \varepsilon \rangle, \sigma_2'|_{gc}, \kappa_3, \varepsilon, \overline{\nu_2}$

- Case **E.8**: We show that this case does not arise.
  – Then
    * $\langle \Pi_2, T_1' \rangle, \sigma_2, \widehat{\kappa_2} \cdot \lfloor \widehat{\rho}, f \rfloor, \epsilon, \overline{\nu}$
      $\xrightarrow{t} \langle \widehat{\Pi_2} \cdot (T_3), \widehat{T_1'} \rangle, \sigma_2, \widehat{\kappa_2}, \_, \_$
      $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \widehat{\kappa_2} \cdot \lfloor \widehat{\rho}, f \rfloor, \varepsilon, \overline{\nu_2}$
    * $\langle \mathsf{drop}_{\boxminus}(\Pi_2), T_1' \rangle; \varepsilon \circlearrowleft^{*} \langle \mathsf{drop}_{\boxminus}(\widehat{\Pi_2} \cdot \Box), \widehat{T_1'} \rangle; T_3$
  – By Lemmas C.1.7 and C.1.4 we get both
    * $\mathsf{drop}_{\boxminus}(\widehat{\Pi_2} \cdot \Box) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_2'))$
    * $\mathsf{drop}_{\boxminus}(\widehat{\Pi_2} \cdot (T_3)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_2'))$.
  – This is a contradiction and thus rules out this case.

- Case **P.8**: We show that this case does not arise.
  – Then
    * $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2}$
      $\xrightarrow{t} \langle \widehat{\Pi_2} \cdot (T_4), T_3 \rangle, \sigma_2, \kappa_2, \varepsilon, \mathbf{prop}$
      $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
    * $\langle \mathsf{drop}_{\boxminus}(\Pi_2), \varepsilon \rangle; \varepsilon \circlearrowleft^{*} \langle \mathsf{drop}_{\boxminus}(\widehat{\Pi_2} \cdot \boxplus_{T_3}), \varepsilon \rangle; T_4$
  – By Lemmas C.1.7 and C.1.4 we get both
    * $\mathsf{drop}_{\boxminus}(\widehat{\Pi_2} \cdot \boxplus_{T_3}) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_2'))$
    * $\mathsf{drop}_{\boxminus}(\widehat{\Pi_2} \cdot (T_4)) \in \mathsf{Prefixes}(\mathsf{drop}_{\boxminus}(\Pi_2'))$.
  – This is a contradiction and thus rules out this case.

- Case **E.P**
  – Then $T_1' = M_{\rho_2,e} \cdot \widehat{T_1}$ and $\langle \Pi_2, T_1' \rangle, \sigma_2, \kappa_2, \rho_2, \mathbf{memo}\ e$
    $\xrightarrow{t} \langle \Pi_2 \cdot M_{\rho_2,e}, \widehat{T_1} \rangle, \sigma_2, \kappa_2, \rho_2, e$
    $\xrightarrow{t}{}^{n-1} \langle \Pi_2', T_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
  – Hence

$$\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^{*} \langle \mathsf{drop}_{\boxminus}(\Pi_2) \cdot M_{\rho_2,e}, \_ \rangle; \widehat{T_2} \circlearrowleft \langle \mathsf{drop}_{\boxminus}(\Pi_2), \_ \rangle ; T_2'$$

  with $T_2' = M_{\rho,e} \cdot \widehat{T_2}$ by Lemma C.1.8 and (c).

- From (b) we have that $\mathsf{M}_{\rho_2,e}\widehat{\mathsf{T}_1}$ ok, and by inversion we have that $\mathsf{M}_{\rho_2,e}\widehat{\mathsf{T}_1}$ fsc.
- Hence, by Lemma C.1.17 we know there exists some components Π_1, σ_1, κ_1, ρ_1, α_{r1} such that
  - $\langle \Pi_1, \varepsilon\rangle, \sigma_1, \kappa_1, \rho_1, \alpha_{r1} \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi_1, \varepsilon\rangle, \sigma_1, \kappa_1, \rho_2, \mathbf{memo}\ e$ using **E.0** only
  - $\widehat{\mathsf{T}_1}$ fsc from $\langle \Pi_1 \cdot \mathsf{M}_{\rho_2,e}, \varepsilon\rangle, \sigma_1, \kappa_1, \rho_2, e$
- $\Pi_2 \cdot \mathsf{M}_{\rho_2,e}$ ok follows from Π_2 ok.
- Induction and part (1) then yield:
  (i) $\langle \Pi_2', \mathsf{T}_1''\rangle$ ok
  (ii) $\langle \Pi_3 \cdot \mathsf{M}_{\rho_2,e}, \varepsilon\rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_2, e \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi_3', \varepsilon\rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$
  (iii) $\langle \Pi_3', \varepsilon\rangle; \varepsilon \circlearrowleft^{*} \langle \Pi_3 \cdot \mathsf{M}_{\rho_2,e}, \varepsilon\rangle; \widehat{\mathsf{T}_2} \circlearrowleft \langle \Pi_3, \varepsilon\rangle ; \mathsf{T}_2'$
- Finally, using Lemma C.1.3,
  $\langle \Pi_3, \varepsilon\rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_1, \alpha_{r1}$
  $\xrightarrow{\mathsf{t}}{}^{*} \langle \Pi_3 \cdot \mathsf{M}_{\rho_2,e}, \varepsilon\rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_2, e.$

- Case **U.1**
  - Then $\langle \Pi_2, \mathsf{A}_{\ell,m} \cdot \widehat{\mathsf{T}_1'}\rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2}$
    $\xrightarrow{\mathsf{t}} \langle \Pi_2, \widehat{\mathsf{T}_1'}\rangle, \sigma_2[\ell \mapsto \diamond], \kappa_2, \rho_2, \alpha_{r2}$
    $\xrightarrow{\mathsf{t}}{}^{n-1} \langle \Pi_2', \mathsf{T}_1''\rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
  - By inversion on (b) we have that $\mathsf{A}_{\ell,m} \cdot \widehat{\mathsf{T}_1'}$ fsc
  - By Lemma C.1.17, we have that $\widehat{\mathsf{T}_1'}$ fsc
  - Hence, $\widehat{\mathsf{T}_1'}$ ok
  - Induction yields:
    (i) $\langle \Pi_2', \mathsf{T}_1''\rangle$ ok
    (ii) $\langle \Pi_3, \varepsilon\rangle, \sigma_2[\diamond \mapsto \ell]|_{\mathsf{gc}}, \kappa_3, \rho_2, \alpha_{r2} \xrightarrow{\mathsf{t}}{}^{*} \langle \Pi_3', \varepsilon\rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$
    (iii) $\langle \Pi_3', \varepsilon\rangle; \varepsilon \circlearrowleft^{*} \langle \Pi_3, \varepsilon\rangle; \mathsf{T}_2'$
  - By definition, $\sigma_2[\ell \mapsto \diamond]|_{\mathsf{gc}} = \sigma_2|_{\mathsf{gc}}$
  - Hence, $\langle \Pi_3, \varepsilon\rangle, \sigma_2|_{\mathsf{gc}}, \kappa_3, \rho_2, \alpha_{r2}$
    $\xrightarrow{\mathsf{t}}{}^{*} \langle \Pi_3', \varepsilon\rangle, \sigma_2'|_{\mathsf{gc}}, \kappa_3, \varepsilon, \overline{\nu_2}$

- Case **U.2**
  - Then $\langle \Pi_2, \mathsf{t} \cdot \widehat{\mathsf{T}_1'}\rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2}$
    $\xrightarrow{\mathsf{t}} \langle \Pi_2, \widehat{\mathsf{T}_1'}\rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2}$
    $\xrightarrow{\mathsf{t}}{}^{n-1} \langle \Pi_2', \mathsf{T}_1''\rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
  - By inversion on (b), with the knowledge that $\mathsf{t} \neq (\_)$, we have that $\mathsf{t} \cdot \widehat{\mathsf{T}_1'}$ fsc
  - By Lemma C.1.17, we have that $\widehat{\mathsf{T}_1'}$ fsc
  - Hence, $\widehat{\mathsf{T}_1'}$ ok

- The claim then follows by induction.

- Case **U.3**
  - Then $\langle \Pi_2, (\widehat{\mathsf{T}_1'}) \cdot \widehat{\mathsf{T}_2'} \rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2} \xrightarrow{t} \langle \Pi_2 \cdot \boxminus_{\widehat{\mathsf{T}_2'}}, \widehat{\mathsf{T}_1'} \rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2} \xrightarrow{t}^{n-1} \langle \Pi_2', \mathsf{T}_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
  - By inversion of (b) we show that $\widehat{\mathsf{T}_1'}$ ok and $\widehat{\mathsf{T}_2'}$ ok:
    - Subcase: $\widehat{\mathsf{T}_1'}$ ok and $\widehat{\mathsf{T}_2'}$ ok.
      - Immediate.
    - Subcase: $(\widehat{\mathsf{T}_1'}) \cdot \widehat{\mathsf{T}_2'}$ fsc.
      - From Lemma C.1.17 we have $\widehat{\mathsf{T}_1'}$ fsc and $\widehat{\mathsf{T}_2'}$ fsc.
      - The claim then follows immediately.
  - Hence from (b), we have $\Pi_2 \cdot \boxminus_{\widehat{\mathsf{T}_2'}}$ ok
  - Note that $\mathsf{drop}_{\boxminus}(\Pi_2 \cdot \boxminus_{\widehat{\mathsf{T}_2'}}) = \mathsf{drop}_{\boxminus}(\Pi_2)$
  - Hence, from (c) we have $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\Pi_2 \cdot \boxminus_{\widehat{\mathsf{T}_2'}}), \_ \rangle; \mathsf{T}_2'$
  - The claim then follows by induction.
- Case **U.4**
  - Then $\langle \widehat{\Pi_2} \cdot \boxminus_{\widehat{\mathsf{T}}}, \varepsilon \rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2} \xrightarrow{t} \langle \widehat{\Pi_2}, \widehat{\mathsf{T}} \rangle, \sigma_2, \kappa_2, \rho_2, \alpha_{r2} \xrightarrow{t}^{n-1} \langle \Pi_2', \mathsf{T}_1'' \rangle, \sigma_2', \kappa_2, \varepsilon, \overline{\nu_2}$
  - From (b) we have both $\widehat{\Pi_2}$ ok and $\widehat{\mathsf{T}}$ ok
  - Note that $\mathsf{drop}_{\boxminus}(\widehat{\Pi_2} \cdot \boxminus_{\widehat{\mathsf{T}}}) = \mathsf{drop}_{\boxminus}(\widehat{\Pi_2})$
  - Hence, from (c) we have $\langle \mathsf{drop}_{\boxminus}(\Pi_2'), \_ \rangle; \varepsilon \circlearrowleft^* \langle \mathsf{drop}_{\boxminus}(\widehat{\Pi_2}), \_ \rangle; \mathsf{T}_2'$
  - The claim then follows by induction.

□

Definition C.1.9 (Big-step Sugar)**.**

$$\frac{\langle \varepsilon, \mathsf{T} \rangle, \sigma, \epsilon, \rho, \alpha_r \xrightarrow{t}^* \langle \Pi, \varepsilon \rangle, \sigma', \epsilon, \epsilon, \overline{\nu} \qquad \langle \Pi, \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \varepsilon, \varepsilon \rangle; \mathsf{T}'}{\mathsf{T}, \sigma, \rho, \alpha_r \Downarrow \mathsf{T}', \sigma', \overline{\nu}} \textsc{BigEval}$$

$$\frac{\langle \varepsilon, \mathsf{T} \rangle, \sigma, \epsilon, \epsilon, \mathbf{prop} \xrightarrow{t}^* \langle \Pi, \varepsilon \rangle, \sigma', \epsilon, \epsilon, \overline{\nu} \qquad \langle \Pi, \varepsilon \rangle; \varepsilon \circlearrowleft^* \langle \varepsilon, \varepsilon \rangle; \mathsf{T}'}{\mathsf{T}, \sigma \curvearrowright \mathsf{T}', \sigma', \overline{\nu}} \textsc{BigProp}$$

Corollary (Big-step Consistency)**.** *Suppose* $\varepsilon, \sigma_1, \rho_1, \alpha_{r1} \Downarrow \mathsf{T}_1, \sigma_1', \overline{\nu}_1$ *and* $\mathsf{CSA}(\sigma_1, \rho_1, \alpha_{r1})$.

1. *If* $\mathsf{T}_1, \sigma_2, \rho_2, \alpha_{r2} \Downarrow \mathsf{T}_2, \sigma_2', \overline{\nu}_2$ *then* $\varepsilon, \sigma_2|_{\mathsf{gc}}, \rho_2, \alpha_{r2} \Downarrow \mathsf{T}_2, \sigma_2'|_{\mathsf{gc}}, \overline{\nu}_2$

2. *If* $T_1, \sigma_2 \curvearrowright T_2, \sigma_2', \overline{v}_2$ *then* $\varepsilon, \sigma_2|_{gc}, \rho_1, \alpha_{r1} \Downarrow T_2, \sigma_2'|_{gc}, \overline{v}_2$

Proof. Immediate corollary of Theorem C.1.22 □

C.2 Proofs for DPS Conversion

In this section, let D_f^x denote the auxiliary function that is used in the DPS translation of a **push** command (where $n = \mathsf{Arity}(f)$):

$$\begin{aligned} D_f^x = (&\textbf{fun } f'(y).\textbf{update} \\ &\quad \textbf{let } y_1 = \textbf{read}(y[1]) \textbf{ in } \cdots \\ &\quad \textbf{let } y_n = \textbf{read}(y[n]) \textbf{ in} \\ &\quad f\,(y_1, \ldots, y_n, x)) \end{aligned}$$

Furthermore, we write $\mathsf{FF}(X)$ to denote the set of function names free in the syntactic object X.

C.2.1 DPS Conversion Preserves Extensional Semantics

Definition C.2.1.

$$\frac{}{\varepsilon \sim^{\epsilon \mapsto \epsilon} \varepsilon} \qquad \frac{\kappa_1 \sim^{\overline{x} \mapsto \overline{\ell}} \kappa_2 \qquad [\![\rho_f]\!] \subseteq \rho_f' \wedge \rho_f'(x_f) = \ell_f \wedge \rho_f'(f') = D_f^{x_f}}{\kappa_1 \cdot \lfloor \rho_f, f \rfloor \sim^{\overline{x} @ x_{f'} \mapsto \overline{\ell} @ \ell_f} \kappa_2 \cdot \lfloor \rho_f', f' \rfloor}$$

Theorem C.2.1. *If*

- $\sigma_1, \kappa_1, \rho_1, e \xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{v}$
- $\mathsf{dom}(\sigma_2) = \{\overline{\ell}, \ell\}$
- $\overline{\ell}, \ell \notin \mathsf{dom}(\sigma_1')$
- $\kappa_1 \sim^{\overline{x} \mapsto \overline{\ell}} \kappa_2$
- $[\![\rho_1]\!] \subseteq \rho_2$
- $\rho_2(x) = \ell$

then $\sigma_1 \uplus \sigma_2, \kappa_2, \rho_2, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma_1' \uplus \sigma_2', \varepsilon, \varepsilon, \ell'$ *where* $\ell' = \mathsf{head}(\overline{\ell} @ \ell)$ *and* $\sigma_2'(\ell', i) = v_i$ *for all* i.

Proof. By induction on the length of the reduction chain.

- Case $\boxed{e = \textbf{let fun } f(\overline{z}).e_1 \textbf{ in } e_2}$:

- Then $\sigma_1, \kappa_1, \rho_1, e \xrightarrow{r} \sigma_1, \kappa_1, \rho_1', e_2$
  $\xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{v}$,
  where $\rho_1' = \rho_1[f \mapsto \textbf{fun } f(\overline{z}).e_1]$.
- We know $\sigma_1 \uplus \sigma_2, \kappa_2, \rho_2, [\![e]\!]_x \xrightarrow{r} \sigma_1 \uplus \sigma_2, \kappa_2, \rho_2', [\![e_2]\!]_x$, where $\rho_2' = \rho_2[f \mapsto \textbf{fun } f(\overline{z} @ y).[\![e_1]\!]_y]$.
- It is easy to see that $[\![\rho_1']\!] \subseteq \rho_2'$ follows from $[\![\rho_1]\!] \subseteq \rho_2$.
- The claim then follows by induction.

- Case $\boxed{e = \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2}$:
  - Suppose $\rho_1(x) = 0$ (the other case is analogous).
  - Then $\sigma_1, \kappa_1, \rho_1, e \xrightarrow{r} \sigma_1, \kappa_1, \rho_1, e_1 \xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{v}$.
  - $[\![\rho_1]\!] \subseteq \rho_2$ implies $\rho_2(x) = 0$.
  - Hence we know $\sigma_1 \uplus \sigma_2, \kappa_2, \rho_2, [\![e]\!]_x \xrightarrow{r} \sigma_1 \uplus \sigma_2, \kappa_2, \rho_2, [\![e_1]\!]_x$.
  - The claim then follows by induction.

- Case $\boxed{e = f\ (\overline{z})}$:
  - Then $\sigma_1, \kappa_1, \rho_1, e \xrightarrow{r} \sigma_1, \kappa_1, \rho_1', e_f \xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{v}$,
    where $\rho_1' = \rho_1[y_i \mapsto \rho_1(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$
    and $\rho_1(f) = \textbf{fun } f(\overline{y}).e_f$.
  - From $[\![\rho_1]\!] \subseteq \rho_2$ we know $\rho_2(f) = \textbf{fun } f(\overline{y} @ x).[\![e_f]\!]_x$.
  - Hence $\sigma_1 \uplus \sigma_2, \kappa_2, \rho_2, [\![e]\!]_x \xrightarrow{r} \sigma_1 \uplus \sigma_2, \kappa_2, \rho_2', [\![e_f]\!]_x$,
    where $\rho_2' = \rho_2[y_i \mapsto \rho_2(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$.
  - It is easy to see that $[\![\rho_1']\!] \subseteq \rho_2'$ follows from $[\![\rho_1]\!] \subseteq \rho_2$.
  - The claim then follows by induction.

- Case $\boxed{e = \textbf{let } y = \iota \textbf{ in } e'}$:
  - Then $\sigma_1, \kappa_1, \rho_1, e \xrightarrow{r} \sigma_1'', \kappa_1, \rho_1', e' \xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{v}$, where $\rho_1' = \rho_1[y \mapsto v']$ and $\sigma_1, \rho_1, \iota \xrightarrow{s} \sigma_1'', v'$.
  - Since $\mathsf{dom}(\sigma_2) = \{\overline{\ell}, \ell\}$ and $\overline{\ell}, \ell \notin \mathsf{dom}(\sigma_1') \supseteq \mathsf{dom}(\sigma_1'')$ and $[\![\rho_1]\!] \subseteq \rho_2$, we get $\sigma_1 \uplus \sigma_2, \rho_2, \iota \xrightarrow{s} \sigma_1'' \uplus \sigma_2, v'$.
  - Hence we know $\sigma_1 \uplus \sigma_2, \kappa_2, \rho_2, [\![e]\!]_x \xrightarrow{r} \sigma_1'' \uplus \sigma_2, \kappa_2, \rho_2', [\![e']\!]_x$,
    where $\rho_2' = \rho_2[y \mapsto v']$.
  - It is easy to see that $[\![\rho_1']\!] \subseteq \rho_2'$ follows from $[\![\rho_1]\!] \subseteq \rho_2$.

– The claim then follows by induction.

- Case $\boxed{e = \textbf{push}\ f\ \textbf{do}\ e'}$:
  - Then $\sigma_1, \kappa_1, \rho_1, e \xrightarrow{r} \sigma_1, \kappa_1', \rho_1, e' \xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{\nu}$, where $\kappa_1' = \kappa_1 \cdot \lfloor \rho_1, f \rfloor$.
  - We know $\sigma_2, \kappa_2, \rho_2, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma_2', \kappa_2', \rho_2', [\![e']\!]_{x'}$, where
    - $\sigma_2' = \sigma_2[(\ell', i) \mapsto \bot]_{i=1}^{n}$, so $\mathsf{dom}(\sigma_2') = \{\bar{\ell}, \ell, \ell'\}$
    - $\ell' \notin \mathsf{dom}(\sigma_2) \cup \mathsf{dom}(\sigma_1')$
    - $\kappa_2' = \kappa_2 \cdot \lfloor \rho_f', f' \rfloor$
    - $\rho_f' = \rho_2[f' \mapsto D_f^x]$
    - $\rho_2' = \rho_f'[x' \mapsto \ell']$
  - We show $\kappa_1' \sim^{\overline{x} @ x \mapsto \overline{\ell} @ \ell} \kappa_2'$:
    - $\kappa_1 \sim^{\overline{x} \mapsto \overline{\ell}} \kappa_2$ is given.
    - $[\![\rho_1]\!] \subseteq \rho_f'$ follows from $[\![\rho_1]\!] \subseteq \rho_2$.
    - $\rho_f'(x) = \ell$ follows from $\rho_2(x) = \ell$.
    - $\rho_f'(f') = D_f^x$ is obvious.
  - Also, $[\![\rho_1]\!] \subseteq \rho_2'$ follows from $[\![\rho_1]\!] \subseteq \rho_2$
  - Finally, $\rho_2'(x') = \ell'$.
  - The claim then follows by induction (note that $\mathsf{head}(\overline{\ell} @ \ell) = \mathsf{head}(\overline{\ell} @ \ell @ \ell')$).
- Case $\boxed{e = \textbf{pop}\ \overline{z}}$ and $\kappa_1 = \varepsilon$:
  - Then $\sigma_1, \kappa_1, \rho_1, e \xrightarrow{r} \sigma_1, \varepsilon, \varepsilon, \overline{\nu}$ and $\sigma_1' = \sigma_1$ and $\nu_i = \rho_1(z_i)$ for all i.
  - From $\kappa_1 \sim^{\overline{x} \mapsto \overline{\ell}} \kappa_2$ we get $\kappa_2 = \varepsilon$ and $\overline{\ell} = \varepsilon$.
  - Thus we know $\sigma_1 \uplus \sigma_2, \kappa_2, \rho_2, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma_1 \uplus \sigma_2', \varepsilon, \varepsilon, \ell$,
    where $\sigma_2' = \sigma_2[(\ell, i) \mapsto \rho_2(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$.
  - Note that $\sigma_2'(\ell, i) = \rho_2(z_i) = \rho_1(z_i) = \nu_i$, for any i.
  - Finally, note that $\ell = \mathsf{head}(\ell) = \mathsf{head}(\overline{\ell} @ \ell)$.
- Case $\boxed{e = \textbf{pop}\ \overline{z}}$ and $\kappa_1 = \kappa_1' \cdot \lfloor \rho_f, f \rfloor$:
  - Then $\sigma_1, \kappa_1, \rho_1, e \xrightarrow{r}{}^{*} \sigma_1, \kappa_1', \rho_1', e_f \xrightarrow{r}{}^{*} \sigma_1', \varepsilon, \varepsilon, \overline{\nu}$,
    where $\rho_1' = \rho_f[y_i \mapsto \rho_1(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$
    and $\rho_f(f) = \textbf{fun}\ f(\overline{y}).e_f$.
  - From $\kappa_1 \sim^{\overline{x} \mapsto \overline{\ell}} \kappa_2$ we know
    - $\kappa_2 = \kappa_2' \cdot \lfloor \rho_f', f' \rfloor$

  - $\overline{x} = \overline{x}' \mathbin{@} x_f$ and $\overline{\ell} = \overline{\ell}' \mathbin{@} \ell_f$
  - $[\![\rho_f]\!] \subseteq \rho'_f \wedge \rho'_f(x_f) = \ell_f$
  - $\rho'_f(f') = D_f^{x_f}$
- Therefore $\sigma_1 \uplus \sigma_2, \kappa_2, \rho_2, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma_1 \uplus \sigma'_2, \kappa_2, \varepsilon, \ell$, where $\sigma'_2 = \sigma_2[(\ell, i) \mapsto \rho_2(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$.
- And $\sigma_1 \uplus \sigma'_2, \kappa_2, \varepsilon, \ell \xrightarrow{r}{}^{*} \sigma_1 \uplus \sigma'_2, \kappa'_2, \rho'_2, f\,(y_1, \ldots, y_n, x_f)$, where $\rho'_2 = \rho'_f[y \mapsto \ell][y_i \mapsto \rho_2(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$.
- And $\sigma_1 \uplus \sigma'_2, \kappa'_2, \rho'_2, f\,(y_1, \ldots, y_n, x_f) \xrightarrow{r}{}^{*} \sigma_1 \uplus \sigma'_2, \kappa'_2, \rho'_2, [\![e_f]\!]_{x_f}$.
- Note that $[\![\rho'_1]\!] \subseteq \rho'_2$ follows from $[\![\rho_f]\!] \subseteq \rho'_f$ and $[\![\rho_1]\!] \subseteq \rho_2$.
- The claim then follows by induction.

□

Corollary. *If* $\sigma_1, \varepsilon, \rho, e \xrightarrow{r}{}^{*} \sigma'_1, \varepsilon, \varepsilon, \overline{v}$,
then $\sigma_1, \varepsilon, [\![\rho]\!], \mathbf{let}\ x = \mathbf{alloc}(n)\ \mathbf{in}\ [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma'_1 \uplus \sigma'_2, \varepsilon, \varepsilon, \ell$
with $\sigma'_2(\ell, i) = v_i$ *for all* i.

C.2.2 DPS Conversion Produces CSA Programs

Definition C.2.2.

$$\rho' \propto \rho \iff [\![\rho']\!] \subseteq \rho \wedge \forall f \in \mathsf{dom}(\rho).\ \mathsf{FF}(\rho(f)) \subseteq \mathsf{dom}(\rho')$$

Definition C.2.3.

$$\frac{}{\varepsilon\heartsuit} \qquad \frac{\kappa\heartsuit \qquad \rho'_f(x_f) = \ell_f \wedge \rho'_f(f') = D_f^{x_f} \wedge \exists \rho_1.\ \rho_1 \propto \rho'_f}{\kappa \cdot \lfloor \rho'_f, f' \rfloor \heartsuit}$$

Lemma C.2.2. *If*

1. $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \mathbf{update}\ e'$
2. $\exists \rho_1.\ \rho_1 \propto \rho \wedge \mathit{FF}([\![e]\!]_x) \subseteq \mathsf{dom}(\rho_1)$
3. $\kappa\heartsuit$
4. $\rho(x) = \ell$

then:

- $e' = [\![e'']\!]_y$

- $\rho'(y) = \ell'$
- $\exists \rho_2.\ \rho_2 \propto \rho' \wedge FF([\![e'']\!]_y) \subseteq \mathsf{dom}(\rho_2)$

Proof. By induction on the length of the reduction chain in (1).

- Case $\boxed{e = \textbf{let fun } f(\overline{z}).e_1 \textbf{ in } e_2}$:
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma, \kappa, \hat{\rho}, [\![e_2]\!]_x \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \textbf{update } e'$, where $\hat{\rho} = \rho[f \mapsto \textbf{fun } f(\overline{z} \mathbin{@} y).[\![e_1]\!]_y]$.
  - Note that (2) has been preserved.
  - The claim then follows by induction.
- Case $\boxed{e = \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2}$:
  - Suppose $\rho(x) = 0$ (the other case is analogous).
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma, \kappa, \rho, [\![e_1]\!]_x \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \textbf{update } e'$.
  - Note that (2) has been preserved.
  - The claim then follows by induction.
- Case $\boxed{e = f\ (\overline{z})}$:
  - From (2) we know $\rho(f) = \textbf{fun } f(\overline{y} \mathbin{@} x).[\![e_f]\!]_x$.
  - Hence $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma, \kappa, \hat{\rho}, [\![e_f]\!]_x \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', e'$, where $\hat{\rho} = \rho[y_i \mapsto \rho(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$.
  - Note that (2) has been preserved.
  - The claim then follows by induction.
- Case $\boxed{e = \textbf{let } z = \iota \textbf{ in } \hat{e}}$:
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \hat{\sigma}, \kappa, \hat{\rho}, [\![\hat{e}]\!]_x \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \textbf{update } e'$, where $\hat{\rho} = \rho[z \mapsto v]$.
  - Note that (2) has been preserved.
  - The claim then follows by induction.
- Case $\boxed{e = \textbf{memo } \hat{e}}$:
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma, \kappa, \rho, [\![\hat{e}]\!]_x \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \textbf{update } e'$.
  - Note that (2) has been preserved.
  - The claim then follows by induction.
- Case $\boxed{e = \textbf{update } \hat{e}}$:

- If the length of the reduction is 0, then $e' = [\![\hat{e}]\!]_x$ and we are done.
- Otherwise we know $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma, \kappa, \rho, [\![\hat{e}]\!]_x \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \textbf{update}\ e'$.
- Note that (2) has been preserved.
- The claim then follows by induction.

- Case $\boxed{e = \textbf{push}\ f\ \textbf{do}\ \hat{e}}$:
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma, \kappa, \rho'_f, \textbf{push}\ f'\ \textbf{do memo let}\ z = \textbf{alloc}(n)\ \textbf{in}\ [\![\hat{e}]\!]_z$, where $\rho'_f = \rho[f' \mapsto D_f^x]$.
  - Now $\sigma, \kappa, \rho'_f, \textbf{push}\ f'\ \textbf{do memo let}\ z = \textbf{alloc}(n)\ \textbf{in}\ [\![\hat{e}]\!]_z \xrightarrow{r}{}^{*} \tilde{\sigma}, \tilde{\kappa}, \tilde{\rho}, [\![\hat{e}]\!]_z$, where:
    - $\tilde{\kappa} = \kappa \cdot \lfloor \rho'_f, f' \rfloor$
    - $\tilde{\rho} = \rho'_f[z \mapsto \ell']$
  - Note that $[\![\rho_1]\!] \subseteq \tilde{\rho} \wedge \mathsf{FF}([\![\hat{e}]\!]_z) \subseteq \mathsf{dom}(\rho_1) \wedge \forall f \in \mathsf{dom}(\tilde{\rho}).\ \mathsf{FF}(\tilde{\rho}(f)) \subseteq \mathsf{dom}(\rho_1)$ follows from (2).
  - Furthermore we know $\tilde{\sigma}, \tilde{\kappa}, \tilde{\rho}, [\![\hat{e}]\!]_z \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \textbf{update}\ e'$.
  - The claim thus follows by induction if we can show $\tilde{\kappa}\heartsuit$.
  - And yes, we can!

- Case $\boxed{e = \textbf{pop}\ \overline{z}}$ and $\kappa = \varepsilon$: impossible due to (1)

- Case $\boxed{e = \textbf{pop}\ \overline{z}}$ and $\kappa = \tilde{\kappa} \cdot \lfloor \rho'_f, f' \rfloor$:
  - From $\kappa\heartsuit$ we know:
    1. $[\![\rho_2]\!] \subseteq \rho'_f \wedge \forall g \in \mathsf{dom}(\rho'_f).\ \mathsf{FF}(\rho'_f(g)) \subseteq \mathsf{dom}(\rho_2)$
    2. $\rho'_f(f') = D_f^{x_f}$
    3. $\tilde{\kappa}\heartsuit$
  - Hence we know $\rho'_f(f) = \textbf{fun}\ f(\overline{y} \mathbin{@} x_f).[\![e_f]\!]_{x_f}$.
  - So $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r}{}^{*} \tilde{\sigma}, \tilde{\kappa}, \tilde{\rho}, f\ (y_1, \ldots, y_n, x_f) \xrightarrow{r}{}^{*} \tilde{\sigma}, \tilde{\kappa}, \tilde{\rho}, [\![e_f]\!]_{x_f}$, where:
    - $\tilde{\rho} = \rho'_f[y \mapsto \ell][y_i \mapsto \rho(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$
  - Note that $[\![\rho_2]\!] \subseteq \rho'_f \wedge \forall g \in \mathsf{dom}(\rho'_f).\ \mathsf{FF}(\rho'_f(g)) \subseteq \mathsf{dom}(\rho_2)$ implies $[\![\rho_2]\!] \subseteq \tilde{\rho} \wedge \mathsf{FF}([\![e_f]\!]_{x_f}) \subseteq \mathsf{dom}(\rho_2) \wedge \forall f \in \mathsf{dom}(\tilde{\rho}).\ \mathsf{FF}(\tilde{\rho}(f)) \subseteq \mathsf{dom}(\rho_2)$.
  - Furthermore we know $\tilde{\sigma}, \tilde{\kappa}, \tilde{\rho}, [\![e_f]\!]_{x_f} \xrightarrow{r}{}^{*} \sigma', \kappa', \rho', \textbf{update}\ e'$ and the claim thus follows by induction.

□

Definition C.2.4.

$$\frac{}{\varepsilon \rhd^{\ell} \ell}$$

$$\frac{\kappa \rhd^{\ell_f} \ell' \qquad \rho'_f(x_f) = \ell_f \wedge \rho'_f(f') = D_f^{x_f} \wedge \exists \rho_1.\ \rho_1 \propto \rho'_f}{\kappa \cdot \lfloor \rho'_f, f' \rfloor \rhd^{\ell} \ell'}$$

Lemma C.2.3. *If*

1. $\kappa \rhd^{\ell} \ell'$
2. $\rho(x) = \ell$
3. $\exists \rho_1.\ \rho_1 \propto \rho \wedge FF([\![e]\!]_x) \subseteq \mathsf{dom}(\rho_1)$
4. $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \overline{v}$

then $\overline{v} = \ell'$.

Proof. By induction on the length of the reduction chain.

- Case $\boxed{e = \textbf{let fun } f(\overline{z}).e_1 \textbf{ in } e_2}$:
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma, \kappa, \rho', [\![e_2]\!]_x \xrightarrow{r}{}^{*} \_, \varepsilon, \varepsilon, \overline{v}$, where $\rho' = \rho[f \mapsto \textbf{fun } f(\overline{z} \mathbin{@} y).[\![e_1]\!]_y]$.
  - Note that (3) has been preserved.
  - The claim then follows by induction.
- Case $\boxed{e = \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2}$:
  - Suppose $\rho(x) = 0$ (the other case is analogous).
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma, \kappa, \rho, [\![e_1]\!]_x \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \overline{v}$.
  - The claim then follows by induction.
- Case $\boxed{e = f\ (\overline{z})}$:
  - From (3) we know $\rho(f) = \textbf{fun } f(\overline{y} \mathbin{@} x').[\![e_f]\!]_{x'}$.
  - Hence $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma, \kappa, \rho', [\![e_f]\!]_x \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \overline{v}$, where $\rho' = \rho[y_i \mapsto \rho(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$.
  - Note that (3) has been preserved.
  - The claim then follows by induction.
- Case $\boxed{e = \textbf{let } y = \iota \textbf{ in } e'}$:
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r} \sigma', \kappa, \rho', [\![e']\!]_x \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \overline{v}$, where $\rho' = \rho[y \mapsto v']$.

- Note that (3) has been preserved.
- The claim then follows by induction.

- Case $\boxed{e = \textbf{push}\ f\ \textbf{do}\ e'}$:
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma'', \kappa', \rho', [\![e']\!]_{x_f} \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \overline{v}$, where
    - $\kappa' = \kappa \cdot \lfloor \rho'_f, f' \rfloor$
    - $\rho'_f = \rho[f' \mapsto D^x_f]$
    - $\rho' = \rho'_f[x_f \mapsto \ell_f]$
  - We show $\kappa' \rhd^{\ell_f} \ell'$:
    - $\exists \rho_1.\ [\![\rho_1]\!] \subseteq \rho'_f \wedge \forall g \in \mathsf{dom}(\rho'_f).\ \mathsf{FF}(\rho'_f(g)) \subseteq \mathsf{dom}(\rho_1)$ follows from (3).
    - $\rho'_f(f') = D^x_f$ is obvious.
    - $\rho'_f(x) = \ell$ follows from $\rho(x) = \ell$.
    - $\kappa \rhd^{\ell} \ell'$ is given.
  - Note that (3) has been preserved.
  - The claim then follows by induction.

- Case $\boxed{e = \textbf{pop}\ \overline{z}}$ and $\kappa = \varepsilon$:
  - Then $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \ell$ and thus $\overline{v} = \ell$.
  - From $\kappa \rhd^{\ell} \ell'$ we know $\ell = \ell'$.

- Case $\boxed{e = \textbf{pop}\ \overline{z}}$ and $\kappa = \kappa' \cdot \lfloor \rho'_f, f' \rfloor$:
  - From $\kappa \rhd^{\ell} \ell'$ we know
    1. $\exists \rho_1.\ [\![\rho_1]\!] \subseteq \rho'_f \wedge \forall g \in \mathsf{dom}(\rho'_f).\ \mathsf{FF}(\rho'_f(g)) \subseteq \mathsf{dom}(\rho_1)$
    2. $\rho'_f(f') = D^{x_f}_f$
    3. $\rho'_f(x_f) = \ell_f$
    4. $\kappa' \rhd^{\ell_f} \ell'$
  - So $\sigma, \kappa, \rho, [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma'', \kappa, \varepsilon, \ell \xrightarrow{r}{}^{*} \sigma'', \kappa', \rho', f\,(y_1, \ldots, y_n, x_f)$, where $\rho' = \rho'_f[y \mapsto \ell][y_i \mapsto \rho(z_i)]_{i=1}^{\mathsf{length}(\overline{z})}$.
  - From (2) and (1) we know $\rho'(f) = \textbf{fun}\ f(\overline{y} \mathbin{@} x_f).[\![e_f]\!]_{x_f}$.
  - Thus $\sigma'', \kappa', \rho', f\,(y_1, \ldots, y_n, x_f) \xrightarrow{r} \sigma'', \kappa', \rho', [\![e_f]\!]_{x_f} \xrightarrow{r}{}^{*} \sigma', \varepsilon, \varepsilon, \overline{v}$.
  - The claim then follows by induction.

□

Theorem C.2.4. $\mathsf{CSA}(\sigma, [\![\rho]\!], \textbf{let}\ x = \textbf{alloc}(n)\ \textbf{in}\ [\![e]\!]_x)$

Proof. Suppose $\sigma, \varepsilon, [\![\rho]\!], \textbf{let } x = \textbf{alloc}(n) \textbf{ in } [\![e]\!]_x \xrightarrow{r}{}^{m} \sigma', \kappa, \rho', e'$.
We must show $\mathsf{SA}(\sigma', \rho', e')$.
We distinguish two cases:

- Case $m = 0$:
  - So suppose $\sigma', \varepsilon, [\![\rho]\!], \textbf{let } x = \textbf{alloc}(n) \textbf{ in } [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma'', \kappa', \rho'', \textbf{update } e''$.
  - Hence $\sigma'[(\ell, i) \mapsto \bot]_{i=1}^{n}, \varepsilon, [\![\rho]\!][x \mapsto \ell], [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma'', \kappa', \rho'', \textbf{update } e''$.
  - Since $\varepsilon\heartsuit$, Lemma C.2.2 yields:
    - $e'' = [\![\widehat{e}]\!]_y$
    - $\rho''(y) = \ell'$
    - $[\![\rho_2]\!] \subseteq \rho'' \wedge \mathsf{FF}([\![\widehat{e}]\!]_y) \subseteq \mathsf{dom}(\rho_2) \wedge \forall f \in \mathsf{dom}(\rho''). \mathsf{FF}(\rho''(f)) \subseteq \mathsf{dom}(\rho_2)$
  - Now suppose $\_, \varepsilon, \rho'', [\![\widehat{e}]\!]_y \xrightarrow{r}{}^{*} \_, \varepsilon, \varepsilon, \overline{v}$.
  - Since $\varepsilon \triangleright^{\ell'} \ell'$, Lemma C.2.3 yields $\overline{v} = \ell'$.
- Case $m > 0$:
  - Then $\sigma[(\ell, i) \mapsto \bot]_{i=1}^{n}, \varepsilon, [\![\rho]\!][x \mapsto \ell], [\![e]\!]_x \xrightarrow{r}{}^{m-1} \sigma', \kappa, \rho', e'$.
  - So suppose $\sigma', \varepsilon, \rho', e' \xrightarrow{r}{}^{*} \sigma'', \kappa', \rho'', \textbf{update } e''$.
  - By Lemma C.1.10, $\sigma', \kappa, \rho', e' \xrightarrow{r}{}^{*} \sigma'', \kappa@\kappa', \rho'', \textbf{update } e''$.
  - Hence $\sigma[(\ell, i) \mapsto \bot]_{i=1}^{n}, \varepsilon, [\![\rho]\!][x \mapsto \ell], [\![e]\!]_x \xrightarrow{r}{}^{*} \sigma'', \kappa@\kappa', \rho'', \textbf{update } e''$.
  - The rest goes as in the first case.

□

C.3 Cost Semantics Proofs

Lemma C.3.1. *If* $\langle \Pi, \varepsilon \rangle ; \varepsilon \circlearrowleft^{*} \langle \Pi' \cdot \square, T_2 \rangle ; T_1$ *and* $\boxminus \notin \Pi$*, then* $T_2 = \varepsilon$.

Theorem C.3.2. *If*

- $\sigma, \kappa, \rho, e \xrightarrow{r}{}^{*} \_, \varepsilon, \varepsilon, \overline{v}_1$*, described by* $\overline{s_1}$
- $\langle \Pi, \varepsilon \rangle, \sigma, \kappa, \rho, e \xrightarrow{t}{}^{*} \_, \_, \varepsilon, \varepsilon, \overline{v}_2$*, described by* $\overline{s_2}$
- $\boxplus, \boxminus \notin \Pi$

then:

- $\gamma_s\ \overline{s_1} = \gamma_s\ \overline{s_2}$

- $\gamma_\sigma\ \overline{s_1} = \gamma_\sigma\ \overline{s_2}$
- $\gamma_\kappa\ \overline{s_1} = \gamma_\kappa\ \overline{s_2}$

Proof. By induction on the length of $\overline{s_1}$. Note that $\overline{s_1} = \langle\rangle$ is not possible, so $\overline{s_1} = s^u :: \overline{s_3}$. We analyze s^u and in each case observe that $\overline{s_2}$ must start with the step s^t that is associated with the corresponding **E** rule. Note that:

- Rules **E.P** and **P.E** never apply because the reuse trace is empty.
- Rules **P.1–7** never apply because **prop** is not an expression.
- Rule **P.8** never applies because its premise would imply $\boxplus \in \Pi$.
- Rules **U.1–3** never apply because the reuse trace is empty.
- Rule **U.4** never applies because $\boxminus \notin \Pi$.

In each case, we find that s^t has the same cost as s^u in the three models, i.e., $\gamma\ s^u = \gamma\ s^t$ for $\gamma \in \{\gamma_s, \gamma_\sigma, \gamma_\kappa\}$. Also, each step preserves the assumptions. In particular, in rule **E.8**, $T_2 = \varepsilon$ implies $T_2' = \varepsilon$ by Lemma C.3.1.

□

Corollary. *If*

- $\sigma, \varepsilon, \rho, e \xrightarrow{r}{}^* \_, \varepsilon, \varepsilon, \overline{v}_1$*, described by* $\overline{s_1}$
- $\langle\varepsilon, \varepsilon\rangle, \sigma, \varepsilon, \rho, e \xrightarrow{t}{}^* \_, \_, \varepsilon, \varepsilon, \overline{v}_2$*, described by* $\overline{s_2}$

then:

- $\gamma_s\ \overline{s_1}\ \mathbf{0}_s = \gamma_s\ \overline{s_2}\ \mathbf{0}_s$
- $\gamma_\sigma\ \overline{s_1}\ \mathbf{0}_\sigma = \gamma_\sigma\ \overline{s_2}\ \mathbf{0}_\sigma$
- $\gamma_\kappa\ \overline{s_1}\ \mathbf{0}_\kappa = \gamma_\kappa\ \overline{s_2}\ \mathbf{0}_\kappa$

Theorem C.3.3. *Suppose the following:*

- $\sigma_1, \varepsilon, \rho, e \xrightarrow{r}{}^* \sigma_1', \varepsilon, \varepsilon, \overline{v}$*, described by* $\overline{s_1}$
- $\sigma_1, \varepsilon, [\![\rho]\!], \textbf{let } x \textbf{ = alloc}(n) \textbf{ in } [\![e]\!]_x \xrightarrow{r}{}^* \sigma_1' \uplus \sigma_2', \varepsilon, \varepsilon, \ell$*, described by* $s_{alloc} :: \overline{s_2}$
- $\langle u, d, \_, \_\rangle = \gamma_\kappa\ \overline{s_1}\ \mathbf{0}_\kappa$
- $\langle a_1, r_1, w_1\rangle = \gamma_\sigma\ \overline{s_1}\ \mathbf{0}_\sigma$
- $\langle a_2, r_2, w_2\rangle = \gamma_\sigma\ \overline{s_2}\ \mathbf{0}_\sigma$
- N *is the maximum arity of any* **pop** *taken in* $\overline{s_1}$

Then:

1. $\gamma_\kappa\ \overline{s_1}\ \mathbf{0}_\kappa = \gamma_\kappa\ \overline{s_2}\ \mathbf{0}_\kappa$
2. $a_2 - a_1 = u$
3. $r_2 - r_1 \leq N * d$
4. $w_2 - w_1 \leq N * (d + 1)$
5. $\gamma_s\ \overline{s_2}\ \mathbf{0}_s - \gamma_s\ \overline{s_1}\ \mathbf{0}_s \leq (2N + 5) * u + N$

Proof. Informally, this is easy to see from the definition of DPS conversion as explained below. The only interesting cases are **push**s and **pop**s. Note that since both computations start and end with an empty stack, we know $u = d$.

1. Observe that the conversion preserves the number and order of **push**s and **pop**s and that both computations end in an empty stack.
2. Observe that the translation of a **push** introduces a single additional **alloc**.
3. Observe that the translation of a **push** introduces a function containing at most N additional **read**s. Each time the stack is popped, such a function is executed. This happens d times.
4. Observe that the translation of a **pop** introduces at most N additional **write**s. We know that $d + 1$ **pop**s are executed (the last one when the stack is already empty, thereby terminating the program).
5. Observe that: executing the translation of a **push** takes 3 additional steps (function definition, **memo**, **alloc**) to reach its body; executing the translation of a **pop** (of which $d + 1$ are executed) takes at most N steps before actually doing the **pop**; in the d cases where the stack is popped, the function generated by the corresponding **push** is executed, which takes at most $1 + N + 1$ steps. In total, this adds up to $3 * u + N * (d + 1) + (1 + N + 1) * d = (2N + 5) * u + N$ additional steps.

Formally, this can be proven by a very tedious induction, similar to—but much more space consuming than—the proof of Theorem C.2.1. □